AN INVESTIGATION INTO THE CONTROL OF AUDIO STREAMING ACROSS NETWORKS HAVING DIVERSE QUALITY OF SERVICE MECHANISMS

A thesis submitted in fulfilment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

of

RHODES UNIVERSITY

by

PHILIP JAMES FOULKES

September 2011

Abstract

The transmission of realtime audio data across digital networks is subject to strict quality of service requirements. These networks need to be able to guarantee network resources (e.g., bandwidth), ensure timely and deterministic data delivery, and provide time synchronisation mechanisms to ensure successful transmission of this data. Two open standards-based networking technologies, namely IEEE 1394 and the recently standardised Ethernet AVB, provide distinct methods for achieving these goals.

Audio devices that are compatible with IEEE 1394 networks exist, and audio devices that are compatible with Ethernet AVB networks are starting to come onto the market. There is a need for mechanisms to provide compatibility between the audio devices that reside on these disparate networks such that existing IEEE 1394 audio devices are able to communicate with Ethernet AVB audio devices, and vice versa. The audio devices that reside on these networks may be remotely controlled by a diverse set of incompatible command and control protocols. It is desirable to have a common network-neutral method of control over the various parameters of the devices that reside on these networks.

As part of this study, two Ethernet AVB systems were developed. One system acts as an Ethernet AVB audio endpoint device and another system acts as an audio gateway between IEEE 1394 and Ethernet AVB networks. These systems, along with existing IEEE 1394 audio devices, were used to demonstrate the ability to transfer audio data between the networking technologies. Each of the devices is remotely controllable via a network neutral command and control protocol, XFN. The IEEE 1394 and Ethernet AVB devices are used to demonstrate the use of the XFN protocol to allow

for network neutral connection management to take place between IEEE 1394 and Ethernet AVB networks. User control over these diverse devices is achieved via the use of a graphical patchbay application, which aims to provide a consistent user interface to a diverse range of devices.

Acknowledgments

First of all, I would like to thank my project supervisor, Professor Richard Foss, for continually guiding me throughout the project. Many thanks also go to my colleagues for their assistance and support in reaching the goals of the project. To my family and friends, thank you for your support and encouragement.

Thank you to the departmental sponsors for providing financial assistance to undertake this work: Telkom, Comverse, Tellabs, Stortech, Easttel, Bright Ideas 39, THRIP through the Telkom Centre of Excellence at Rhodes University. Thank you to Universal Media Access Networks for providing equipment and source code which proved invaluable during this study. Without this assistance the research would not have been possible.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1 Introduction

Traditionally audio and video production environments are composed of many different independent audio, video, and command and control transports. These solutions are often proprietary and consist of point-to-point connections, switching systems and processors that are centred on specific media formats. For example, video signals may be distributed using SD-SDI [1] or HD-SDI [2] interfaces using coaxial cables to connect devices together. Audio signals may flow through AES-3 interfaces [3] and coaxial cables. Command and control messages require further interfaces and cables. These messaging protocols vary in format between different types of devices with each of these different signal types processed separately.

There is a growing trend towards transporting diverse content (e.g., web and e-mail traffic, realtime audio and video stream data, and command and control data) via single a networking medium. Having audio and video devices interconnected via a common networking technology provides more flexibility and is more future-proof. A common networking technology provides flexible routing capabilities as all of the signals that are transported by the network are available throughout the network. There is no need to add additional cabling and interfaces to a networked system in order to route a signal to a device, as routing is configured in software. It is also possible to allow additional protocols to run over the network without having to replace the network itself. New protocols can be developed in software to run on existing devices, or new devices can be added to an existing network.

Different types of content have different requirements of the network on which they run. Web and e-mail traffic can be delivered on a best effort basis as their consumption is not time critical. For example, variable delays associated with the delivery of web pages are not critical to them being displayed correctly. If, due to network congestion, a webpage takes a few seconds longer to display than a previous one, the content can still be consumed in a meaningful way. However, the successful transportation of realtime audio and video data is subject to strict *quality of service* (QoS) requirements. Realtime audio and video is consumed immediately after it is produced. For example, a performer at a concert singing into a microphone produces audio to be consumed immediately by an audience (via a set of loud speakers). Ideally, the delivery of realtime audio and video data should be instantaneous, but small bounded delays are acceptable. If the delivery of the audio is subjected to

varying delays, the reproduced audio could contain jitter resulting in perceived poor quality audio. Therefore, a network should provide strict QoS mechanisms for the delivery of realtime audio and video data to avoid data loss, unacceptable latency, and lack of synchronisation.

In order to provide an acceptable environment for the transmission of realtime data, a network has to be designed such that there is no packet loss, and it has to provide a guarantee on the maximum latency for the transmitted realtime data. This implies that there are sufficient resources on the network to support the transmission of realtime data streams. Devices involved in realtime data streaming need to be able to communicate their stream resource requirements to the network so that these can be reserved and guaranteed by the network. The devices on a network also need to share a common sense of time to allow for the synchronisation of multiple streams, and to provide a common time base for sampling a data stream (at a source device) and presenting the data stream (at a receiving device).

Traditional *information technology* (IT) networks do not natively provide the appropriate QoS requirements needed for the transport of realtime data. It has thus been difficult to use these networks for the transmission of this data without the network configuration being tightly constrained. However, there are two open standards-based networking technologies that are capable of natively transporting both non-realtime and realtime data with appropriate QoS for realtime stream data. These are *IEEE 1394* [4] and a recently standardised enhanced form of *Ethernet* [5] and Ethernet *bridging*[1] [6], called *Ethernet audio video bridging (AVB)* [7]. These networking technologies provide diverse mechanisms that allow end stations to communicate their stream resource requirements, thereby allowing these resources to be guaranteed by the network. These networking technologies also provide diverse timing and synchronisation mechanisms. These mechanisms enable the deterministic and timely transmission of realtime stream data.

[1] Ethernet bridging is commonly referred to as Ethernet switching.

1.1 Interoperability and Common Command and Control

There currently exist digital audio devices that are compatible with IEEE 1394 networks, and digital audio devices that are compatible with Ethernet AVB networks are starting to come onto the market. A number of manufactures have produced IEEE 1394 compatible audio equipment:

- Yamaha [8] have manufactured a number of audio devices that make use of IEEE 1394. Their 01X [9] and 01V96 [10] audio mixing desks, for instance, are capable of receiving and transmitting audio signals across IEEE 1394 buses. Via an IEEE 1394 bus, these audio mixing desks can be connected together with their Motif XS8 synthesisers [11] and with *personal computers* (PCs), amongst other devices. This enables PC based recording software to be used to record audio signals produced by the equipment on the IEEE 1394 bus. Yamaha has a patchbay application [12] that allows for signals to be routed between the Yamaha devices on the IEEE 1394 bus. Yamaha also produced the N8 [13] and N12 [14] audio mixing desks that allow for seamless integration with *digital audio workstation* (DAW) software via an IEEE 1394 bus.
- PreSonus [15] manufacture a number of IEEE 1394 capable breakout boxes. These allow the audio capabilities of a PC to be extended via a PC's IEEE 1394 interface. They also manufacture a number of audio mixing desks, such as the StudioLive 16.4.2 16-Channel Digital Mixer with FireWire Interface [16], that allows IEEE 1394 connectivity to a PC to enable the recording and playback of audio.
- M-Audio [17] have developed a number of IEEE 1394 breakout boxes that allow the audio capabilities of a PC to be extended via its IEEE 1394 interface. These devices include the ProFire 610 [18] and the ProFire 2626 [19]. The ProFire 610 provides 6 audio inputs and 10 audio outputs, and the ProFire 2626 provides 26 audio inputs and 26 audio outputs in a mixture of analogue and digital form. These devices also have pre-amps built into them.
- Focusrite [20] have developed a number of IEEE 1394 capable audio interfaces. These interfaces extend the audio interface capabilities of a PC via a PC's IEEE 1394 interface. Devices include their Liquid Saffire 56 [21] interface which provides 28 inputs and 28 outputs of analogue and digital audio, and their Saffire PRO 40 [22] which provides 20 inputs and 20 outputs of analogue and digital audio. These devices also include a number of pre-amps.
- Mackie [23] produce a number of IEEE 1394 capable audio interfaces and audio mixing desks. The Mackie Onyx Blackbird [24] is a 16 input, 16 output audio interface that accepts and

transmits audio in analogue and digital formats. The Onyx 1640i FireWire Recording Mixer [25] allows for the transmission and reception of 16 channels of audio between it and a DAW application via IEEE 1394.

- Alesis [26] have produced a range of breakout boxes and audio mixing desks that are IEEE 1394 capable. Their iO|14 [27] and iO|26 [28] devices provide a number of analogue and digital input and outputs to a PC via a PC's IEEE 1394 interface. Alesis have also produced a range of IEEE 1394 capable audio mixing desks, such as their MultiMix 12 FireWire [29] and their MultiMix 16 FireWire [30].
- Allan and Heath [31] produce the ZED-R16 [32] audio mixing desk. This audio mixing desk has an IEEE 1394 interface allowing the device to be connected to a PC to allow for PC based recording to take place. The IEEE 1394 interface allows for the reception of 18 input signals, and the transmission of 18 output signals.
- TerraTec [33] have produced a number of IEEE 1394 capable breakout boxes. These include the Phase X24 [34] and the Phase 88 Rack FW [35]. These devices provide a number of analogue and digital input and outputs, allowing the audio capabilities of a PC to be extended via its IEEE 1394 interface.
- *Universal Media Access Networks* (UMAN) [36], who are involved in the development of software and hardware for audio and video networks, are actively developing and promoting the use of IEEE 1394 and Ethernet AVB networking technology for the transmission of realtime audio and video data. They have developed a number of IEEE 1394 evaluation breakout boxes and IEEE 1394 bridges (see Chapter 9 "Networked Audio Devices") allowing for flexible large scale audio and video networks to be built.

A number of manufactures are creating Ethernet AVB capable hardware:

- LabX [37] have been actively involved in developing and promoting Ethernet AVB. They have released an Ethernet AVB capable bridge, known as the Titanium 411 Ruggedized AVB Ethernet Bridge [38]. They have also released a *field programmable gate array* (FPGA) based AVB Audio Demo Platform [39] allowing for the evaluation and development of Ethernet AVB endpoints.
- BSS Audio [40] have also released an Ethernet AVB capable bridge, known as the GS724T 24 Port Ethernet AVB Switch [41].

- Broadcom [42] have produced an Ethernet AVB capable integrated gigabit Ethernet controller, known as the BCM57765 [43]. This Ethernet controller is currently available in a number of Apple [44] products. These include the iMac [45], Mac Mini [46], and MacBook Pro [47].
- Marvell [48] have produced Ethernet AVB capable networking equipment. These include a number of gigabit Ethernet bridges and a gigabit Ethernet network interface controller [49].
- Crown Audio [50] have released the PIP-USP4 Module [51] which is an input module for their CTs Series two channel amplifier [52]. This input module enables the amplifier to receive digital audio via an Ethernet AVB network.

Audio signals between professional audio devices on an IEEE 1394 bus can be transported using the layer two protocols defined in IEC 61883-6 [53] (see Section Chapter 6 “Media Transport Protocols”). Connection management between IEEE 1394 audio devices can be achieved with the layer two *Function Control Protocol* (FCP) and the connection management procedures defined in IEC 61883-1 [54]. Professional audio devices on an Ethernet AVB network are able to stream audio to each other via the layer two protocols defined in IEEE 1722 [55] (see Section Chapter 6 “Media Transport Protocols”). Connection management between Ethernet AVB devices may be performed with the layer two protocols and procedures defined in IEEE 1722.1 [56]. A number of other parties have defined and developed protocols and procedures that can be used to perform connection management on professional IEEE 1394 and Ethernet AVB audio devices. These include protocols and procedures making use of the *Simple Network Management Protocol* (SNMP) [57], and protocols and procedures making use of the *Open Sound Control* (OSC) protocol [58].

There is a need to provide compatibility between the audio devices that reside on these disparate networks such that existing IEEE 1394 audio devices will be able to stream audio data to and from Ethernet AVB audio devices. As indicated above, the audio devices that reside on these networks may be controlled by a diverse set of incompatible command and control protocols. It is desirable to have a common representation and network-neutral method of control over the various parameters of these devices. This control should allow for the adjustment of the various parameters that are internal to the devices, and allow for network streaming to take place between these devices. This thesis proposes that such network neutral control is possible, and that mechanisms can be put in place to ensure a congruent user interface to provide control over the parameters of these diverse devices. To this end:

- Proof of concept Ethernet AVB endpoint devices were developed that are able to both transmit and receive audio streams over Ethernet AVB networks (see Chapter 10 “Ethernet AVB Devices”).
- Proof of concept IEEE 1394/Ethernet AVB audio gateway devices were developed that are able to transmit and receive audio streams over IEEE 1394 and Ethernet AVB networks. These devices allow for these streams to be transferred between the two networking technologies (see Chapter 10 “Ethernet AVB Devices”).
- Existing IEEE 1394 audio devices were used in conjunction with the developed Ethernet AVB devices to demonstrate the ability to transfer audio data between audio devices on these diverse networks.
- The XFN [59] [60] *Internet Protocol* (IP) [61] based command and control protocol was used to provide the ability to remotely obtain and set parameter values of the IEEE 1394 and the developed Ethernet AVB audio devices (see Chapter 11 “XFN Control and Representation”).
- A connection management application was developed to allow for user control over the abovementioned audio devices. This application allows for the consistent graphical representation and control over the parameters of these audio devices (see Chapter 11 “XFN Control and Representation”).

1.2 Document Structure

Chapter 2 introduces the IEEE 1394 and Ethernet AVB networking technologies. It provides a high-level overview of each of these networking technologies and highlights the core characteristics of each that make them suitable for the transmission of synchronised low-latency realtime stream data: the ability to communicate network resource requirements, the ability of the network to provide timely and deterministic data transmission, and the ability of network devices to share a common sense of time.

Chapter 3 details the mechanisms that IEEE 1394 and Ethernet AVB provide, allowing network resource requirements (e.g., bandwidth) to be communicated by network devices. This allows resources to be reserved for the devices’ realtime data streams.

Chapter 4 describes the mechanisms that are in place in IEEE 1394 and Ethernet AVB networks that ensure that stream resource requirements are met. This ensures that deterministic low-latency stream transmission is able to take place.

Chapter 5 details how devices on IEEE 1394 and Ethernet AVB networks synchronise their clocks to a common master clock. This enables source devices to instruct receiving devices when to present audio samples such that this presentation happens simultaneously on multiple devices receiving the same streams. It also enables the re-creation of sample clock frequencies.

Chapter 6 discusses the dominant media transport protocols used for the transmission of audio streams across IEEE 1394 and Ethernet AVB networks. It discusses the packet and media formatting rules employed by these protocols as well as their synchronisation mechanisms.

Chapter 7 discusses a few of the command and control protocols that have been used, or proposed, to provide control over networked audio devices. It concludes with a discussion of the XFN protocol. The XFN protocol was used during this study to provide remote control over audio devices.

Chapter 8 discusses the implementation of a simple tunnelling protocol that allows for tunnelling of Ethernet traffic across IEEE 1394 networks to provide a deterministic environment for this traffic. It also discusses how the XFN protocol has been used to represent and control the various parameters of the tunnelling devices to allow for their control from a graphical patchbay application.

Chapter 9 provides an overview of the various IEEE 1394 and Ethernet AVB audio devices that were used or developed during this study. It provides a high-level overview of their functionality and of their core components.

Chapter 10 ties the preceding chapters together and provides a detailed discussion of the architecture and operation of the proof of concept Ethernet AVB devices that were developed during this study. One device is an audio gateway between IEEE 1394 and Ethernet AVB networks, and another device is an Ethernet AVB audio endpoint.

Chapter 11 discusses how the XFN protocol has been used to provide a common method of representing and controlling the disparate audio devices that were used during this study. It discusses

the development of a graphical patchbay application used to provide control over the XFN capable devices. This patchbay application aims to provide a consistent graphical representation to a diverse range of networked audio devices.

Chapter 12 provides concluding remarks on the work discussed in this thesis.

Chapter 2 Network Technologies

IEEE 1394 [4] and *Ethernet audio video bridging (AVB)* [5] are two open standards-based networking technologies that natively provide mechanisms that allow for the transfer of synchronised low-latency realtime audio and video streams between devices. The ability to reserve network resources (such as bandwidth), the ability to provide a deterministic environment to ensure timely stream packet transmission, and the ability to provide timing and synchronisation mechanisms are core requirements for networks transferring these kinds of streams. This chapter provides an overview of these networking technologies and a brief introduction to how the above requirements are fulfilled by each of these networking technologies.

2.1 IEEE 1394

IEEE 1394 is a peer-to-peer serial bus interconnect that allows for asynchronous and isochronous forms of packet transmission between the devices that reside on the bus. There is no dependence on a host system, so there is no host processor or memory bottleneck involved in the transfer of data between devices. A device that resides on an IEEE 1394 bus incorporates an *IEEE 1394 node* which may contain one or more ports (but typically two or three). Figure 1 shows a typical arrangement of IEEE 1394 nodes (Node A – Node F) and their ports. A node receiving data on a particular port will re-transmit it to all other ports.

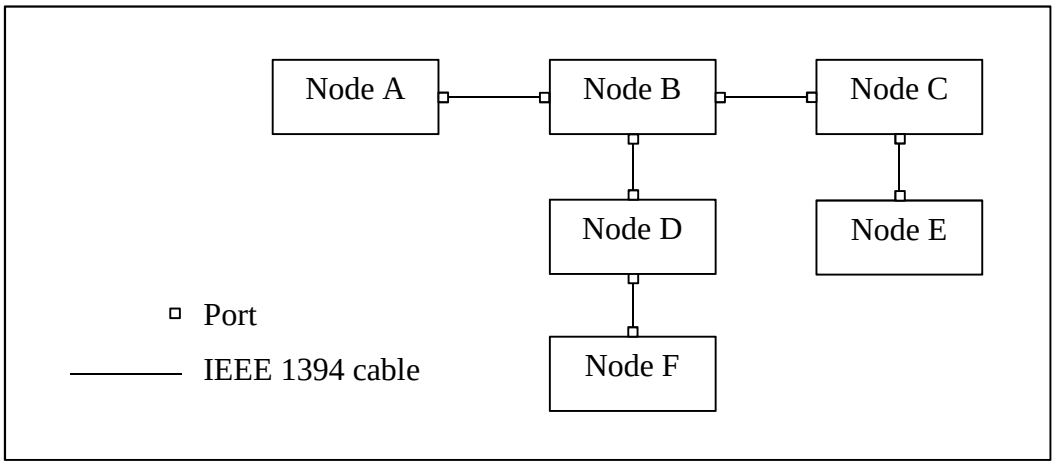

Figure 1: IEEE 1394 topology

An IEEE 1394 bus may contain up to 63 IEEE 1394 nodes and individual IEEE 1394 buses may be bridged together to form larger networks of devices. Up to 1024 buses may exist on a network. Figure 2 shows an example of two IEEE 1394 buses bridged together with an IEEE 1394 bridge [62].

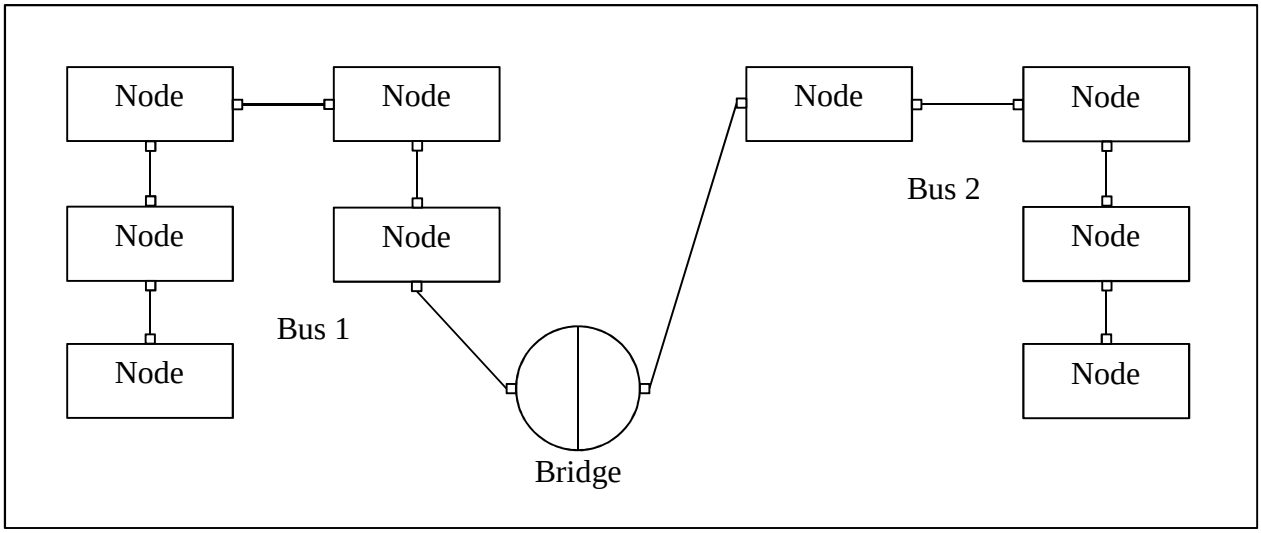

Figure 2: Bridged IEEE 1394 Network

The IEEE 1394 architecture is ideal for applications that require a constant transfer of data, such as those transferring realtime audio and video data as it natively provides deterministic low-latency transmission of data. An IEEE 1394 bus implements an arbitration scheme that guarantees isochronous applications a constant bus bandwidth. Isochronous packets are broadcast onto the bus on one of 64 channels and nodes on the bus can be configured to receive isochronous packets on certain channels. Each node that resides on the bus has its internal clock synchronised with the clock of a master device.

2.1.1 Asynchronous Communication

IEEE 1394 nodes on an IEEE 1394 bus are able to read from, and write to, other IEEE 1394 nodes' registers. These reads and writes are known as *asynchronous transactions*. An asynchronous transaction is initiated by a *requester node* and is received by a *responder node*. Each transaction consists of two *subactions*: a *request subaction*, and a *response subaction*. A request subaction transfers a register address, command, and data (for writes) from the requester to the responder. A response subaction returns the completion status (for writes) back to the requester, or returns data (for reads). Both request and response subactions are communicated using an asynchronous packet.

There are two types of read and write requests: *quadlet* and *block*. A quadlet request reads or writes four octets whereas a block request can read or write a variable number of octets (there is a size limit which is dependent on the transmission speed capability of the transmitting node). An asynchronous packet contains an address that is used to address a particular memory location on a particular IEEE 1394 node.

Connection management requests can be communicated with asynchronous reads and writes to devices. For example, a controlling device may write to another device in order to get it to start transmitting an isochronous stream. The same controlling device may then also write to yet another device to instruct it to start receiving that isochronous stream.

2.1.2 Isochronous Communication

Isochronous packet transmission consists of only one subaction, a *write subaction*. There are no responses to isochronous packet transmission. A device that transmits isochronous packets does so on a particular *isochronous channel*. Each isochronous packet that is transmitted is tagged with a channel number that varies from 0 through to 63. A receiving device can be set to receive packets on one of these channels.

2.1.2.1 Resource Reservation

Before isochronous packets can be transmitted by an IEEE 1394 node, the isochronous channel number and the bus bandwidth required for the stream of isochronous packets has to be obtained and set aside for that stream. An IEEE 1394 node on the bus is elected as the *isochronous resource manager* (IRM) which keeps track of the allocation of isochronous channels and bus bandwidth. An IEEE 1394 node will request the required isochronous stream resources from the IRM, and if these are successfully obtained, they are reserved.

2.1.2.2 Deterministic Transmission

Once a stream's resources have been obtained, the stream's isochronous packets are transmitted at a regular interval, thus providing deterministic transmission of data. On an IEEE 1394 bus, a single

IEEE 1394 node is elected as the *cycle master* node. It regularly transmits a *cycle start* packet (once every 125 µs) and all of the nodes receive the cycle start packet. When a node wanting to transmit an isochronous packet receives a cycle start packet, it will arbitrate for usage of the bus, and if it gains access, it will transmit a queued isochronous packet.

2.1.2.3 Synchronisation

An IEEE 1394 node that wishes to perform isochronous transactions implements a *cycle count* register that is updated by a clock on the node. The cycle master node transmits a cycle start packet every 125 µs. This cycle start packet contains the contents of the cycle master's *cycle count* register and each node updates its *cycle count* register based on the timing value in the cycle start packet. This mechanism allows all of the devices on the bus to be synchronised.

2.2 Audio Video Bridging

Ethernet is the most dominant form of *local area network* (LAN) technology on the market today. The Ethernet protocol is a simple protocol and has been widely adopted [63]. Traditional Ethernet and Ethernet bridging are not able to provide network nodes with guaranteed network resources (such as bandwidth) [64]. While it is possible to transmit synchronised low-latency realtime audio and video data over such a network, the network itself does not guarantee that this is possible. Thus, these types of networks are constrained to avoid interference from unknown devices. With the inability to guarantee network resources, Ethernet frames may be subject to unpredictable frame loss and unacceptable latency variation.

2.2.1 Bridging

This section provides an overview of traditional bridging as it forms the base of Ethernet AVB. A *bridge* is a device that is used to connect IEEE 802 LANs of all types together by relaying and filtering frames between separate *media access control* (MAC) entities. IEEE 802 LANs include IEEE 802.3 (CSMA/CD) [5], IEEE 802.5 (Token Ring) [65] and IEEE 802.11 (Wireless) [66]. In an Ethernet environment, a bridge is often referred to as an Ethernet *switch*. In this document, LAN refers to an individual LAN that is specified by the MAC technology without the inclusion of bridges, and a *bridged LAN* refers to the concatenation of individual IEEE 802 LANs interconnected

by bridges. Figure 3 shows an example of a bridged LAN. In this example, five different individual LANs are bridged together by five bridges. The MAC technology of each of the LANs does not have to be the same.

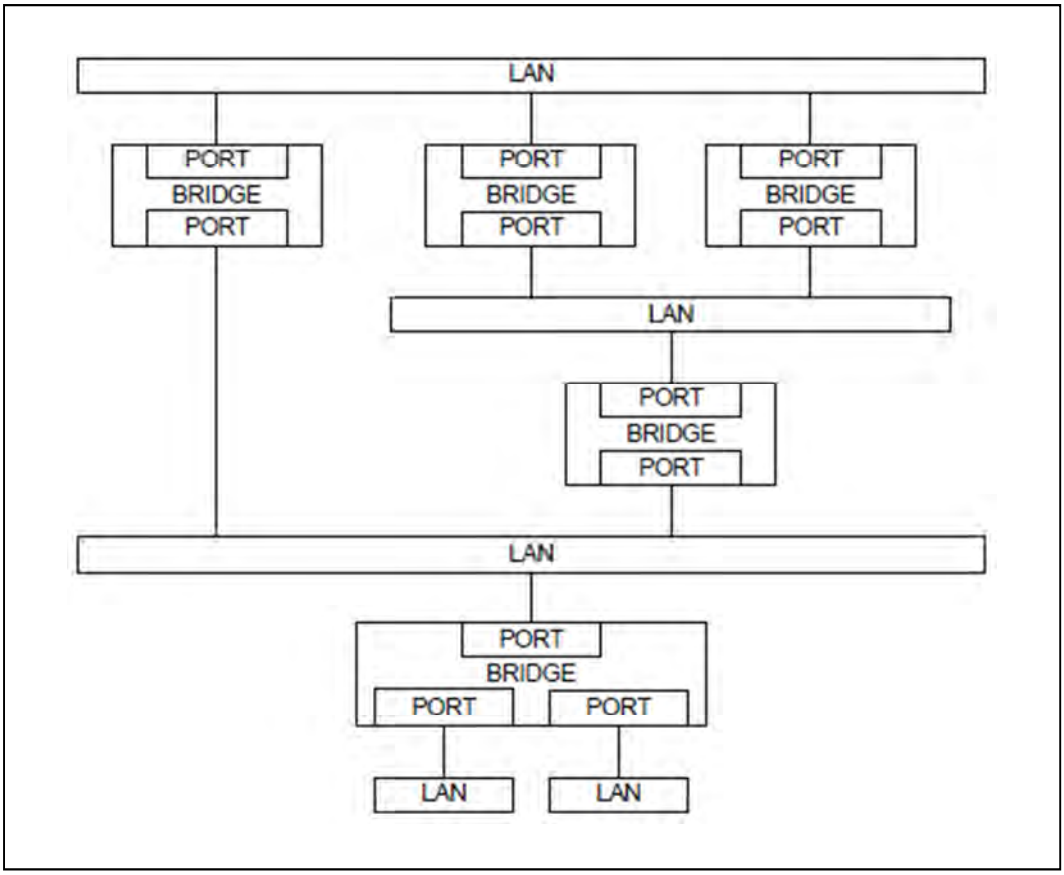

Figure 3: A bridged LAN [6]

2.2.1.1 VLAN Bridging

Virtual LANs (VLANs) allow for the construction of a number of separately manageable virtual bridged LANs to run above bridged LANs. A VLAN has the same properties as a bridged LAN. A VLAN is composed of a set of stations that communicate as if they were attached to a single broadcast domain regardless of their location on a bridged LAN. A bridged LAN may contain a number of VLANs which are configured in software. A VLAN is identified by a *VLAN identifier* (VID). Shown in Figure 4 is a bridged LAN composed of end stations that are each either a *member* of VLAN 1 (indicated as V1) or VLAN 2 (indicated as V2). Figure 5 and Figure 6 highlight the two VLANs that are created to run on top of the bridged LAN. If one of the end stations connected to

VLAN 1 transmits a broadcast frame onto the network, the bridges of the network will only forward that broadcast frame to the LANs containing stations that are members of VLAN 1.

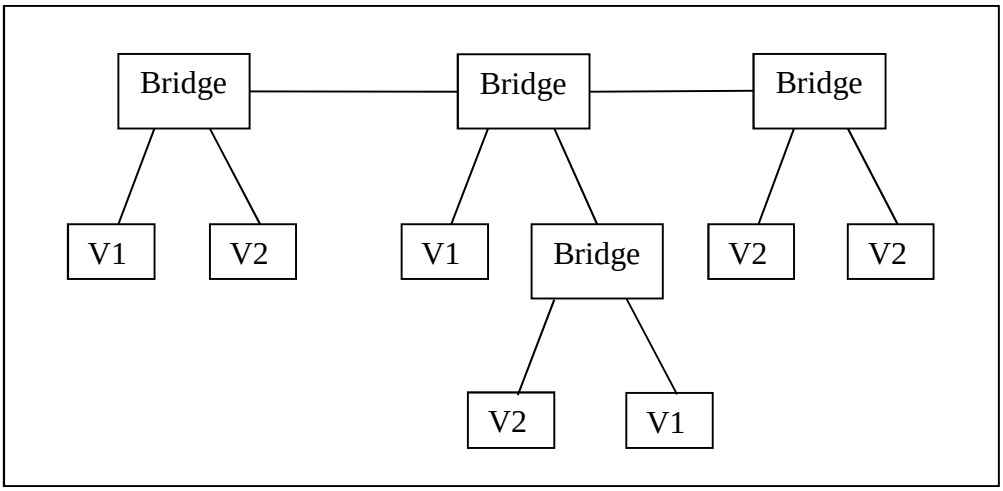

Figure 4: A bridged LAN with VLANs

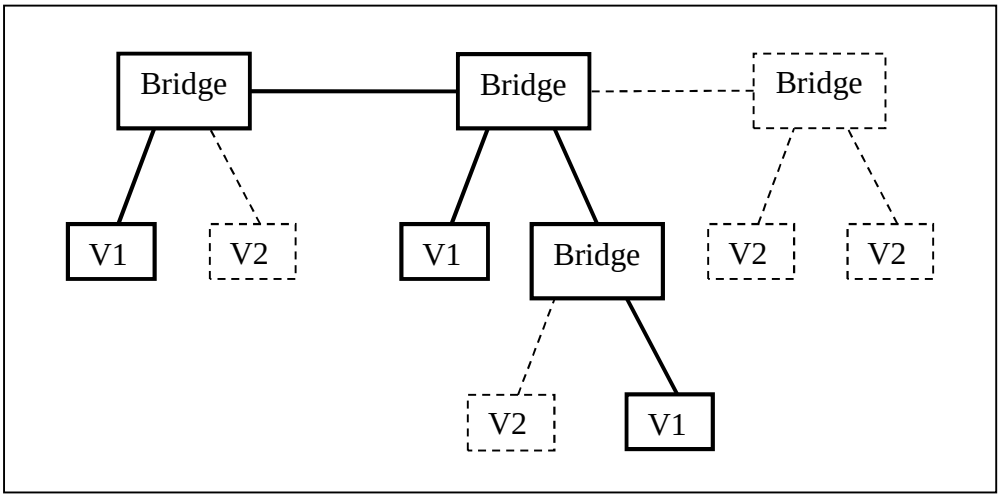

Figure 5: A bridged LAN highlighting VLAN 1

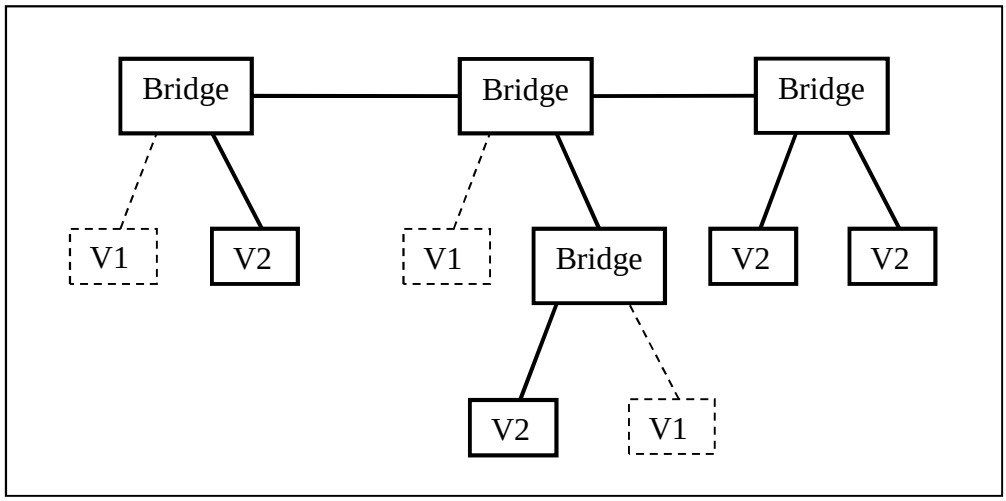

Figure 6: A bridged LAN highlighting VLAN 2

2.2.1.2 Bridge Architecture

A bridge has a *MAC relay entity* that is used to filter and forward individual frames between the ports of a bridge and learn filtering information. Figure 7 shows a two port bridge with a single instance of frame relay between the ports. Each port is able to receive and transmit frames to and from the attached LAN via the *bridge port transmit/receive* process. The *forwarding process* is able to filter frames and forward frames to other ports based on the information contained in the *filtering database*. The filtering database contains appropriate entries that enable a bridge to determine whether frames received (with a given destination MAC address and VID) are to be forwarded through a potential transmission port.

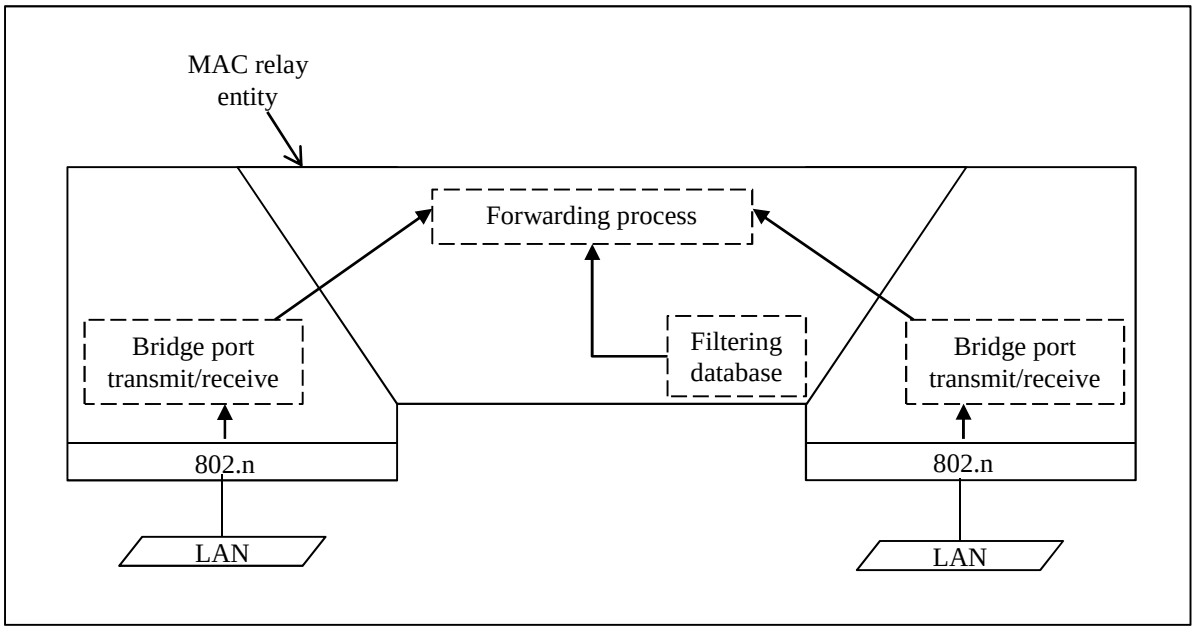

Figure 7: Relaying MAC frames (adapted from [6])

Bridges are not aware of the exact locations of end stations on a bridged LAN, but they determine the ports through which frames should be transmitted in order to ensure that the appropriate end stations receive frames destined to them. Bridges learn about end stations on a network by observing the frames that the end stations transmit. Each frame has a *source MAC address* field that contains the MAC address of the transmitting end station. It is the *learning process* that observes the source address of frames received on each port and updates the filtering database. The learning process is shown in Figure 8.

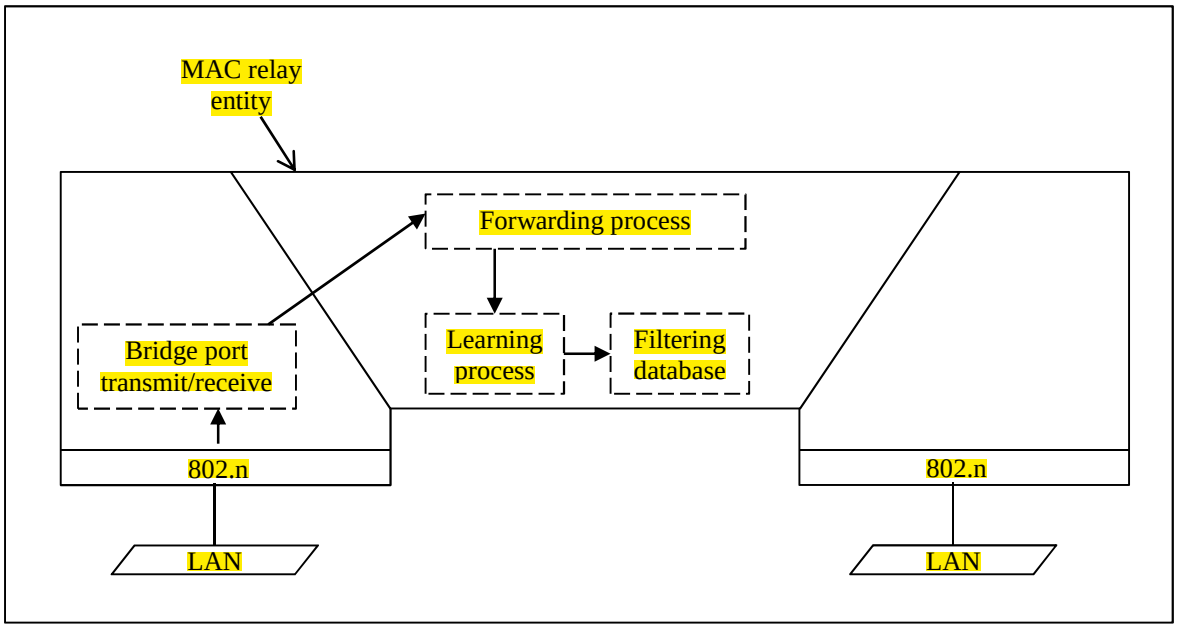

Figure 8: Observing network traffic (adapted from [6])

2.2.2 Audio Video Bridging

The IEEE 802.1 Audio Video Bridging Task Group [7] developed a set of standards commonly known as *audio video bridging* (AVB). This task group is responsible for developing specifications that allow for time-synchronised low-latency streaming services to take place through bridged IEEE 802 networks. Currently their work has focused mainly on the use of Ethernet, but work on 802.11 (Wireless) is also underway. The work described in this thesis focused on the use of Ethernet as the transport medium of the time-sensitive streams. In terms of AVB, a device that transmits a stream of data is known as a *talker*, and a device that receives a stream of data is known as a *listener*.

A key requirement for the transmission of realtime audio and video streams is minimal packet loss. An Ethernet AVB system makes use of Ethernet full-duplex links. The implication of this is that there is no collision of frames on the Ethernet medium. Ethernet AVB frames are standard Ethernet frames that are *VLAN tagged* [6]. VLAN tagging allows for priority information to be carried within the frames, via the VLAN tag's *priority code point* (PCP) field. These priority values are mapped into traffic classes allowing stream requirements to be met. Ethernet AVB frames are transmitted on VLANs (by default, a VLAN with a VID of 2). VLANs are useful for creating separate logical networks above a single infrastructure. This feature could be leveraged to ensure that sensitive audio

streams transmitted across a network (in a VLAN) are only received by devices that are part of the VLAN, and not received by devices that are not part of the VLAN.

There is a set of AVB standards documents that define the various components of an AVB system. These components are briefly introduced below.

2.2.2.1 Resource Reservation

IEEE 802.1 Qat [67] defines the *multiple stream registration protocol* (MSRP). This is a protocol that allows paths (from stream talkers through to stream listeners) to be guaranteed through a bridged LAN for the streams that flow between the devices. It ensures that there are sufficient resources available along the paths to support the streams. A transmitting device is able to communicate its stream resource requirements to a bridged LAN, and a receiving device is able to communicate its intention to receive a stream. This communication allows the network components involved in the transfer of streaming data to guarantee the streams' required resources.

2.2.2.2 Forwarding and Queuing Enhancements for Time-Sensitive Streams

IEEE 802.1 Qav [68] defines forwarding and queuing mechanisms for time-sensitive streams to ensure that the resources allocated to particular streams are guaranteed. End points and bridges that are part of an AVB system implement traffic shaping that guarantees these required resources and smooths the delivery of frames throughout a bridged LAN. Each port's outbound queue that implements the traffic shaping maintains a transmission *credit* which dictates whether frames from the queue may be transmitted or not. The transmission *credit* is decreased and increased at values that are determined by the amount of bandwidth that is reserved for the queue.

2.2.2.3 Timing and Synchronization

IEEE 802.1 AS [69] defines timing and synchronization mechanisms for time-sensitive applications in bridged local area networks. This standard allows for devices on an AVB network to exchange timing information that allows them to synchronise their time base reference clock precisely. Through the operation of the protocol, a single clock is selected as a grandmaster clock from which all other devices derive their time. This clock periodically updates its slave clocks.

2.2.2.4 AVB Systems

IEEE 802.1 BA [70] defines a number of profiles that each define features, options, configurations, defaults, protocols and procedures for bridges, stations and LANs that are to be used to build networks capable of transporting time sensitive streams. Manufacturers can use these defaults and profiles to develop AVB-compatible LAN components. The profiles enable persons to build networks (using these LAN components) that do not require configuration.

2.3 Conclusion

IEEE 1394 and Ethernet AVB are two open standards-based networking technologies that allow for the transport of synchronised, low-latency, realtime audio and video streams between the devices that reside on the respective networks. Each of these networking technologies has mechanisms in place to support the transmission of synchronised low-latency realtime data streams. This chapter gave a brief introduction to these networks. In the chapters that follow, each of the key components that make these networks suitable for the transmissions of synchronised low-latency realtime data will be discussed.

Chapter 3 Resource Reservation

When transmitting realtime data streams across a network there has to be a guarantee that there are sufficient resources to support those streams on the network. A streaming application will be aware of its stream resource requirements, and should communicate these to the network before streaming takes place. If the resources are available on the network, these resources should be reserved for the particular stream(s). If not, the resource reservation should fail. This failure to reserve resources should be communicated to the device requesting the reservation.

In order to develop a common network-neutral command and control protocol for diverse networked audio devices, an understanding of each network's native resource reservation mechanism is required. The command and control protocol should be able to trigger network specific resource reservation procedures with the correct parameters to allow for stream establishment. This chapter explains how resource requirements are natively communicated by devices that reside on IEEE 1394 and AVB networks in order to enable stream connections to be created across these networks.

3.1 Resource Reservation for IEEE 1394

Whenever certain states of any IEEE 1394 node on a bus change (including the addition and removal of IEEE 1394 nodes), a *bus reset* occurs. When this occurs, the bus is reset and it reconfigures itself. During the reconfiguration processes, each node on the IEEE 1394 bus is assigned a unique *physical ID* (or *node ID*) and one of the nodes on the bus is identified as the *isochronous resource manager* (IRM) node. The IRM is responsible for storing isochronous channel and bandwidth allocations. The IRM implements a *bandwidth available* and *channels available* register. The *bandwidth available* register contains the amount of bandwidth currently available for isochronous transfers, and the *channels available* register is a 64-bit bitmap (where each bit represents an isochronous channel) indicating the availability of isochronous channels. Before an IEEE 1394 node is able to transmit isochronous packets, it requests its required network resources (bandwidth and a channel number) from the IRM on the bus. 80% of the available bus bandwidth is available for isochronous transactions.

The *bandwidth available* register is given in terms of *allocation units* where each *allocation unit* is the amount of time required to transmit 4 octets (a quadlet) of data at a data rate of 1.6Gbps. For a transmission rate of 800Mbps, this is the amount of time required to transmit two octets, and at 400Mbps, it is the amount of time required to transmit one octet. 4915 allocation units are available for isochronous transactions. At a data rate of 400Mbps, this translates to a maximum of 4915 octets of data that may be transmitted per isochronous cycle. These limits place a restriction on the maximum data payload size of an isochronous packet, as detailed in Table 1.

Cable speed	Maximum data payload size (octets)
400Mbps	4096
800Mbps	8192
1.6Gbps	16384

Table 1: Maximum data payload size of isochronous packets

Accesses to the *bandwidth available* register are supported only by the *quadlet read* and *lock compare and swap* transactions. A node is able to find out how much bandwidth is available on the bus by reading the IRM's *bandwidth available* register. It is then able to tell whether there is sufficient bandwidth to support its isochronous stream. The *lock compare and swap* transaction passes a data value and an argument value to the target node. The data value is the value being used to update the register (which includes the node's bandwidth requirements), and the argument value is the value previously read. The *lock compare and swap* transaction returns the old value at the target address. If the old value returned is the same as the argument value sent, then there has been no access to the register since it was read and since performing the *lock compare and swap* transaction. Otherwise, the register was updated by another node in the interim and the swap will not be performed.

A node is also able to determine the availability of an isochronous channel, and then request the use of the channel. A node is able to obtain the use of an isochronous channel via the IRM's *channels available* register. The *channels available* register is a 64-bit register where each bit represents a channel. When it is initialised, all of the bits are set to one, indicating that none of the channels have been allocated. A node is able to read the value of the register to determine the next available channel number. It uses a *lock compare and swap* transaction to request the next available channel.

If an IEEE 1394 node's requested resources are available, the node is guaranteed those resources. If, however, any of the requested resources are unavailable, the node is unable to transmit the isochronous packets associated with the resource reservation.

3.2 Resource Reservation for Ethernet AVB

In order to successfully transmit realtime stream data across a bridged LAN (with the required QoS), AVB devices need to ensure that there are sufficient network resources (such as bandwidth and buffer space) on the path from stream talkers to stream listeners before stream transmission can take place.

A stream talker is able to register its intention to transmit a data stream with the network. It transmits a frame containing a *talker advertise* attribute onto the network. This attribute includes details of the characteristics of the stream (e.g., bandwidth and traffic class). Each intermediate bridge that receives the frame is able to ensure that there are sufficient resources to guarantee that the stream frames will be transmitted with the requested parameters.

A stream listener can register its intention to receive a particular data stream. It does so by transmitting a frame containing a *listener ready* attribute towards the talker. This attribute includes details of the stream it would like to receive. The intermediate bridges that receive this frame are able to set up various internal parameters to ensure that there are sufficient resources to transfer the requested stream frames to the listener device. If any resource reservations fail, the participating devices are notified and streaming cannot go ahead. Once a successful path has been established through a bridged LAN, the stream talker is able to transmit a stream.

Stream talker devices are able to tear down streams and de-allocate the resources that are reserved for the streams by withdrawing their stream advertisements. Listeners are able to stop receiving streams by withdrawing their intention to listen to a stream. When a stream is torn down, the resources that were associated with that stream are released.

These stream resource reservation requirements in AVB networks are communicated using the *multiple stream registration protocol* (MSRP) [67]. In addition to the functionality it provides, this

protocol also makes use of the functionality provided by the *multiple MAC registration protocol* (MMRP) [71] and the *multiple VLAN registration protocol* (MVRP) [71]. These protocols are *multiple registration protocol* (MRP) [71] applications. MRP allows for the registration and distribution of information (known as *attributes*) across a bridged LAN. The use of the registered attributes is specific to the application making use of MRP.

3.2.1 Multiple Registration Protocol

MRP is a distributed, many-to-many generic protocol that allows *participants* in an MRP application (for example, MSRP) to *declare attributes*, and to have those attributes *registered* within other participants on a bridged LAN. An attribute is used to convey information and may be composed of one or more fields. For example, MMRP defines a *MAC* attribute that is used to describe a single six octet MAC address, whereas MSRP defines a *talker advertise* attribute (that describes a stream) that is composed of several fields. For example, a *talker advertise* attribute contains an eight octet identifier to identify a stream, and fields that define the maximum frame size and frame rate of the stream.

MRP defines a generic behaviour that all MRP applications conform to. Specific attribute types, their allowed values, and the semantics associated with these values when they are registered, are defined by the specific MRP applications. In essence, MRP is used to establish, maintain, withdraw, and disseminate attribute declarations and registrations among a set of MRP participants. As shown in Figure 9, specific MRP applications inherit the generic behaviour of MRP.

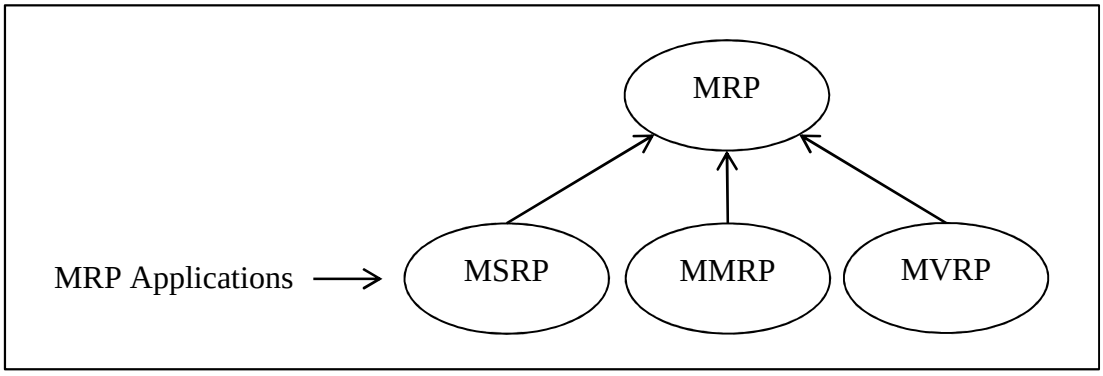

Figure 9: MRP application inheritance

An instance of an MRP application (for example, MSRP) running on a station participates in the MRP protocol with other instances of the same MRP application, and is known as a participant. A participant is able to make or withdraw declarations of attributes. The declaration of an attribute results in the registration of that attribute with the other participants. The withdrawal of a declaration of an attribute results in the removal of registration of the attribute from the other participants. For example, MSRP defines a *talker advertise* attribute and a talker end station is able to declare this attribute. This attribute is registered on all other MSRP participants within a bridged LAN.

Figure 10 shows an example of a bridged LAN. Each end station (denoted with a triangle) has a single port, and each bridge (denoted with a rectangle) has two or more ports. In this example, each port has associated with it a single MRP participant (this is typically the case, but can be defined otherwise per-MRP application). Therefore, each end station has a single MRP participant associated with its port, and a bridge has a single MRP participant associated with each one of its ports. In this figure, a single participant (running on an end station) has made a declaration of an attribute (indicated by the number 1). This attribute is stored by the participant, and it is marked as being declared. The attribute declaration is propagated to all of the other ports attached to the same LAN. The attribute is registered by the participant on the port that receives the declaration (indicated by the number 2). This attribute is stored by the participant, and is marked as being registered. The attribute registration is propagated to, and declared by, all of the other bridge's ports participants (indicated by the number 3). These declarations then repeat the propagation cycle (indicated by the number 4). Once a declaration of an attribute has reached a participant on an end station (for example, the end station indicated by the number 5), it is registered and not propagated any further.

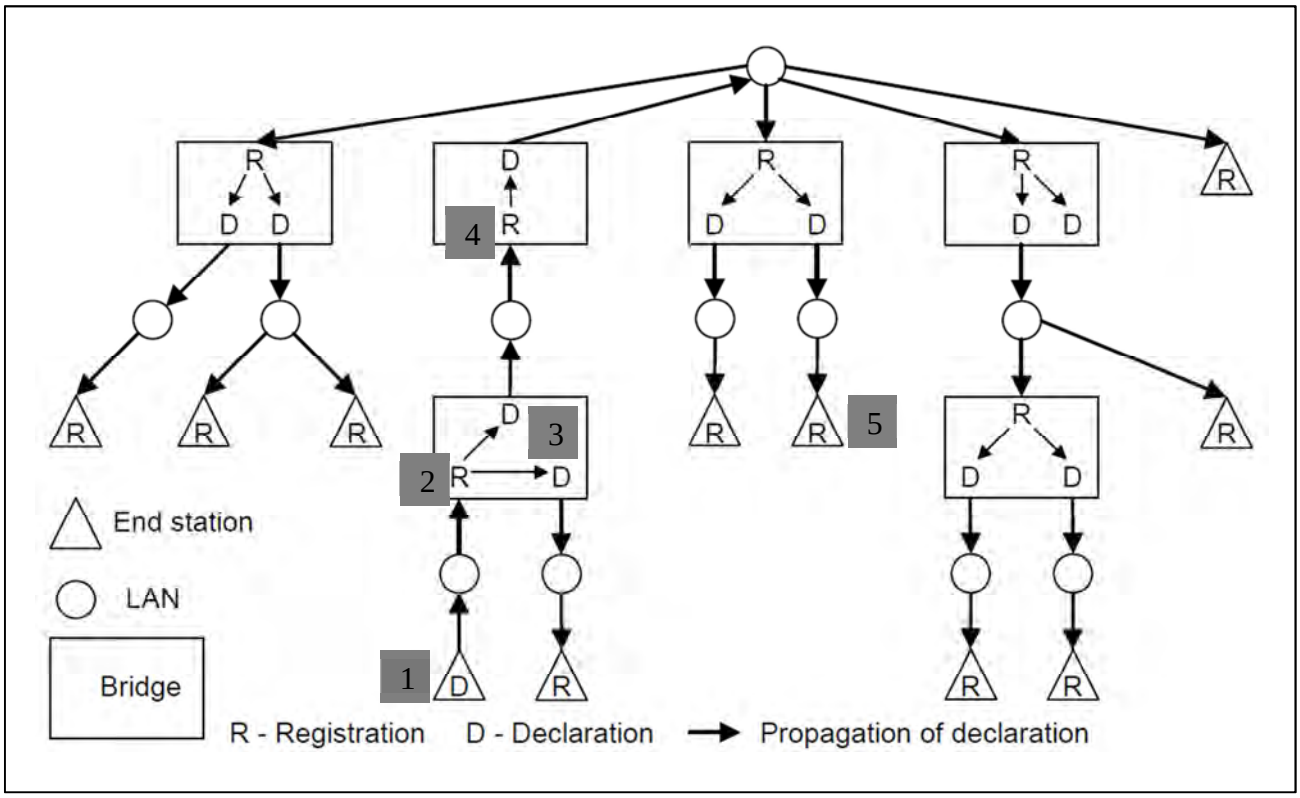

Figure 10: An example of an attribute value propagation from one station (adapted from [71])

Figure 11 shows an example of two different participants (running on different end stations) declaring the same attribute (an attribute with the same type and value) on two different LANs (these end stations are circled in the figure). All of the participants running on end stations register the attribute. Some bridges register the attribute on more than one of its participants.

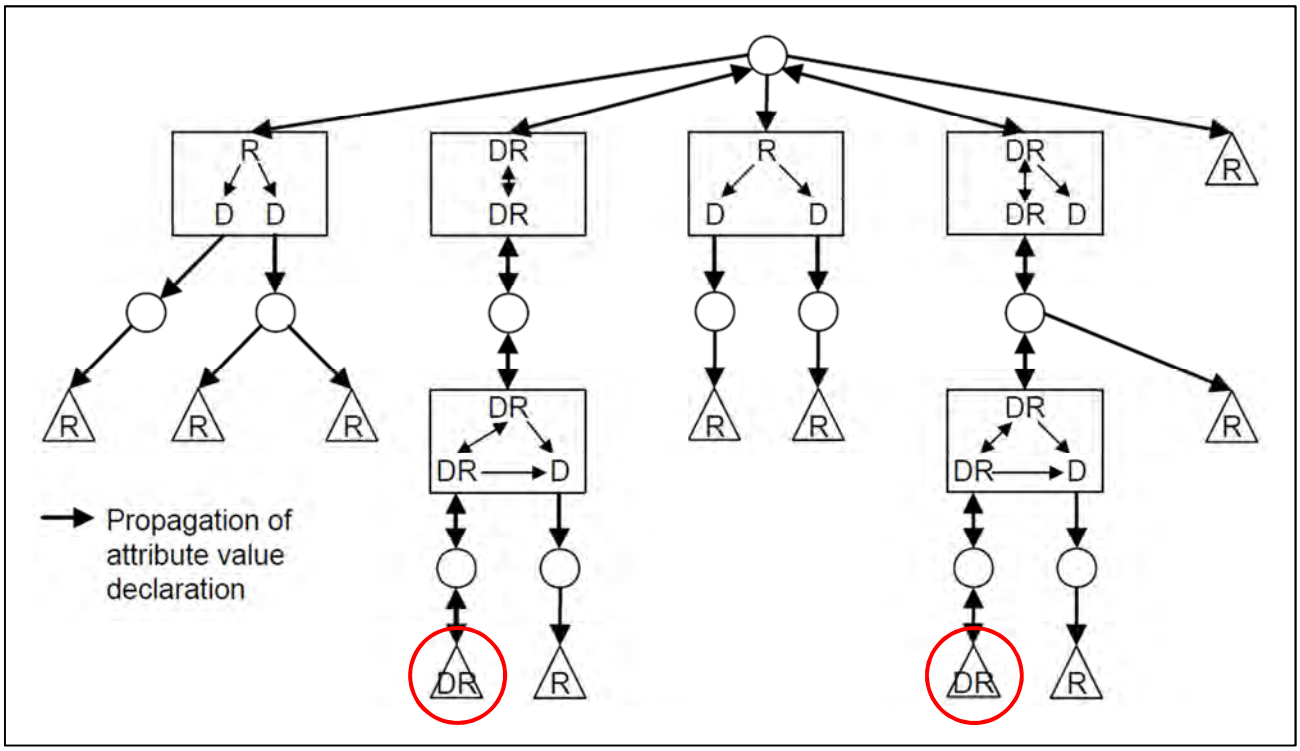

Figure 11: An example of an attribute value propagation from two stations [71]

3.2.1.1 MRP Architecture

An MRP participant consists of the following components:

- ***MRP application* component**: The application component is responsible for the semantics associated with attribute values and attribute registrations. For example, the MSRP application is aware of the semantics associated with the registration of *talker advertise* attributes.
- ***MRP attribute declaration* (MAD) component**: The MAD component is responsible for executing the MRP protocol. It keeps track of attributes, and marks whether each one is declared or registered. It generates MRP messages for transmission, and it processes messages it receives from other participants.

End stations have a single MRP participant per MRP application and bridges have one MRP participant per MRP application per port. For example, in a bridged LAN that supports the MSRP application, an end station will have a single MSRP participant, and a bridge will have an MSRP participant for each port of the bridge. The encoding (and subsequent decoding) of a specific MRP application's attribute values is specific to the MRP application. MRP uses *MRP data units* (MRPDUs) to convey messages generated by the MAD component for a single MRP application.

MRPDUs have a general structure used by all MRP applications. Within bridges, each MRP application has an *MRP attribute propagation* (MAP) component. This component is responsible for propagating information between per-port MRP participants. Figure 12 shows the components of MRP participants (shown in dashed lines) in a two port bridge and an end station.

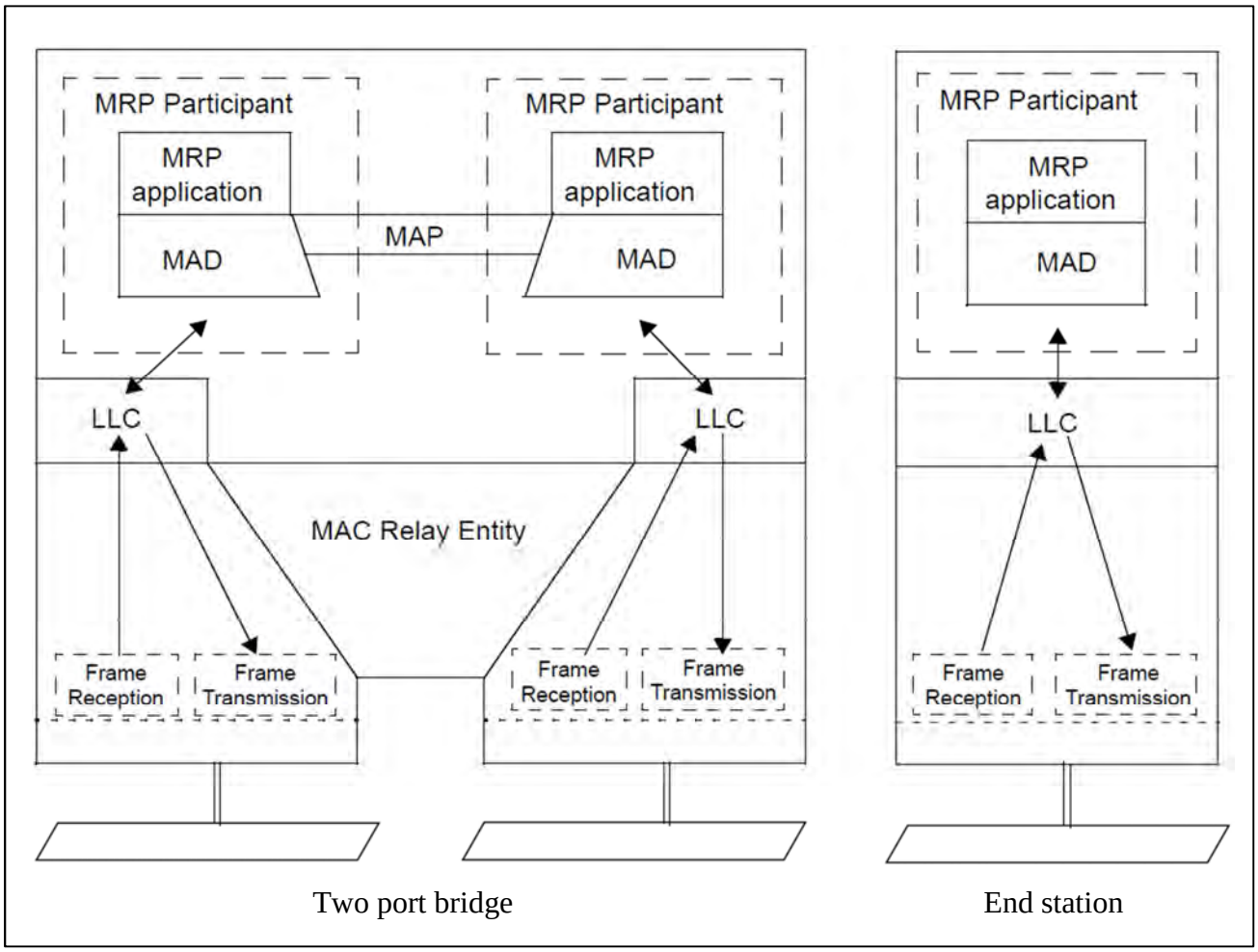

Figure 12: MRP architecture [71]

An MRP application is able to request the MAD component to make and withdraw attribute declarations. This happens via two *service primitives*[2]:

- *MAD join request (attribute type, attribute value, new)*

[2] A service primitive is an interface to a *service provider*, much like a function is an interface to an object in an object-oriented programming language. The service provider carries out the service requested by a *service user*. A service user makes a request to a service provider via a service primitive.

- *MAD leave request (attribute type, attribute value)*

The MAD component is able to notify the MAP component and the MRP application component of changes in attribute registrations. This happens via two service primitives:

- *MAD join indication (attribute type, attribute value, new)*
- *MAD leave indication (attribute type, attribute value)*

These service primitives have the following parameters:

- *Attribute type*: Each MRP application defines a number of attributes and each is identified by a number that is unique to that application. The value of the *attribute type* parameter represents the type of attribute associated with the service primitive invocation.
- *Attribute value*: The value of this parameter is an instance of the attribute referred to by the *attribute type* parameter. If, for example, the *attribute type* parameter refers to a MAC parameter, the attribute value will be a MAC address.
- *New*: The *MAD join request* and the *MAD join indication* service primitives also have a *new* parameter (in addition to the ones mentioned above). This parameter is a Boolean parameter whose value represents an explicitly signalled new declaration. An application making an attribute declaration may explicitly signal that the associated attribute is a new declaration. When the attribute is registered with other participants, the application is notified that the attribute declaration is new. MRP applications choose whether or not they make use of this functionality.

An MRP application defines a set of attribute types and the permitted values for each attribute. It defines the semantics associated with each attribute type and value. The structure and encoding of the attribute types and their values in MRPDUs is application specific. Each MRP application decides on the requirements for MRP state machine support in end stations and bridges.

3.2.1.2 Multiple Attribute Propagation (MAP)

The MAP component (as shown between the two MAD components of the two port bridge in Figure 12) allows for attributes registered on a port to be propagated across a bridged LAN to other MRP participants. Once an attribute has been registered on a port of a bridge, it will be propagated to all other ports of the bridge, which then declare the attribute. Deregistration of an attribute on a bridge port is propagated to all of its other ports via MAP.

3.2.1.3 MRP State Machines

An MRP participant contains a number of state machines. In a full MRP participant, each attribute (whether it is declared, registered, or tracked) has associated with it an *applicant* state machine and a *registrar* state machine. This is shown diagrammatically in Figure 13. When an attribute is created, its associated state machines are created and initialised (see Section 3.2.1.3.2 “Applicant State Machine” and Section 3.2.1.3.3 “Registrar State Machine”). Attributes are not necessarily created when they are declared and/or registered. The time when attributes are created is an implementation decision.

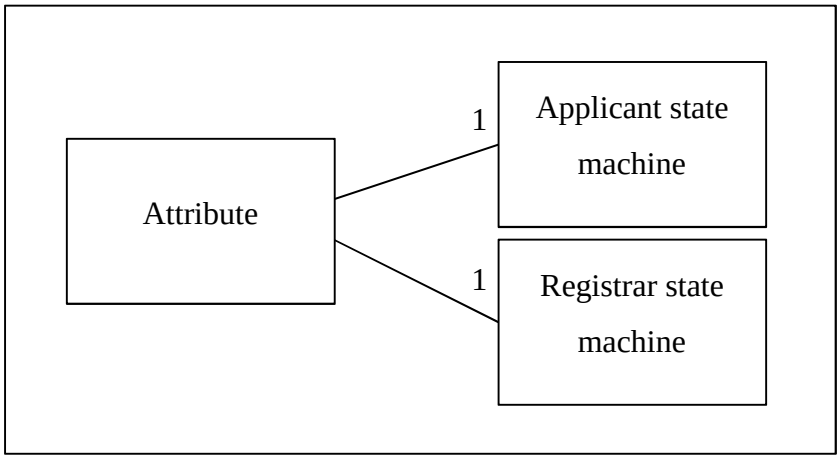

Figure 13: An attribute associated with an applicant and registrar state machine

An MRP participant has associated with it a *leave all* state machine, and a *periodic transmission* state machine. This association is shown diagrammatically in Figure 14.

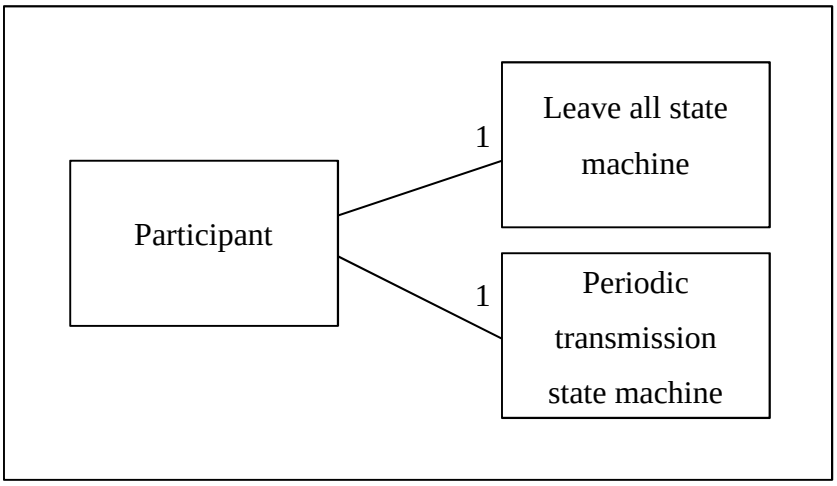

Figure 14: An MRP participant associated with a leave all and periodic transmission state machine

Subsequent sub sections will discuss these state machines, but first the messaging architecture used by the state machines will be described.

3.2.1.3.1 Transmit Opportunity

MRPDUs are used to convey messages between the state machines of participants. A single MRPDU may contain a number of messages. Each message may contain one or more attribute values (each of the same attribute type). For each attribute value there is a corresponding *attribute event*, and an optional value field. The *attribute event* signalled in the message is applied to the state machines of the attribute identified by the *attribute type* and the *attribute value* (examples of *attribute events* will be discussed in the sections that follow). The use of the optional value field is defined by each MRP application. This field is able to carry the range of values 0 through 3. Each MRP application, if it does use the field, associates its own meaning with these values. Figure 15 shows the logical structure of an MRPDU and its contents (the actual encoded structure of the MRPDU may be different to enable efficient transmission).

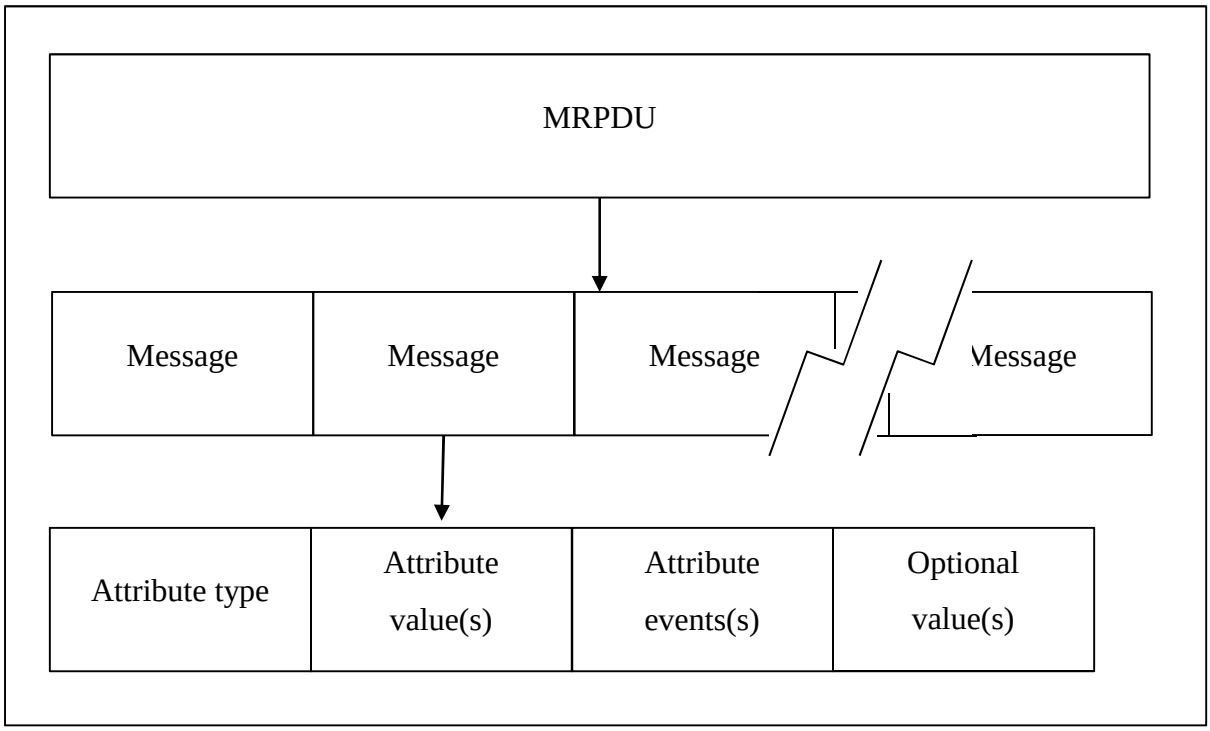

Figure 15: MRPDU structure

If a state machine would like to transmit a message to a LAN, it has to be granted a transmit opportunity. Associated with each participant is a *join period timer*. The join period timer is used to control the time interval between transmit opportunities that are granted to state machines that requested transmit opportunities.

If an MRP participant's state machine wants to transmit a message, it requests a transmit opportunity (if one is not already pending) and is added to a list of state machines (associated with the participant) requesting a transmit opportunity. If this is the first transmit opportunity request after the last transmit opportunity was granted, the join period timer is started. When the join period timer's time expires, the state machines that requested transmit opportunities add their pending messages to an MRPDU. Once all of the pending messages are added to the MRPDU, the MRPDU is transmitted. This process occurs separately for each participant.

The transmit opportunity mechanism facilitates the encoding of multiple attribute messages within single MRPDUs, rather than the transmission of single attribute messages per MRPDU. Each participant is associated with a particular port. MRPDUs are only transmitted through the port that is associated with the participant. Each MRP application defines a unique MAC address and an EtherType. The destination MAC address and EtherType in the MRPDU will be the MAC address

and EtherType associated with the MRP application (for example, MSRP defines its own MAC address and EtherType).

3.2.1.3.2 Applicant State Machine

A declaration of an attribute by an MRP participant is recorded by the attribute's applicant state machine. Certain transitions of the applicant state machine's state may trigger the transmission of MRPDUs. These MRPDUs communicate attribute declaration (or withdrawal) to the other MRP participants. Each applicant state machine maintains a declaration state that records whether it:

- Wishes to make a *new* attribute declaration (as signalled by the *new* parameter of *MAD join request*)
- Wishes to maintain or withdraw an existing attribute declaration
- Has no attribute declaration to make
- It has actively made an attribute declaration
- Has been passive
- Wishes to take advantage of, or simply observe the attribute declarations of others

Figure 16 shows a partial state transition diagram for the initialisation of an applicant state machine. When an attribute is created, its associated state machines are created and initialised. An attribute could be created when it is declared, registered, or at some other time. For example, an MRP application implementation may create all of its required attributes when it is initialised. Even though these attributes exist, they are marked as not being declared, and not being registered. When an applicant state machine is initially created (on attribute creation), it is initialised by signalling it with a *begin* event. The state machine transitions to the VO (*very anxious observer*) state which indicates that the associated attribute is not declared (even though it exists).

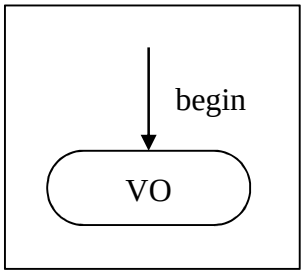

Figure 16: Applicant state machine initialisation

Figure 17 and Figure 18 show partial state transition diagrams for an applicant state machine. These show the state transitions of the state machine when declaring an attribute, and then withdrawing the attribute declaration. They assume that no other events take place other than the ones shown in the figures.

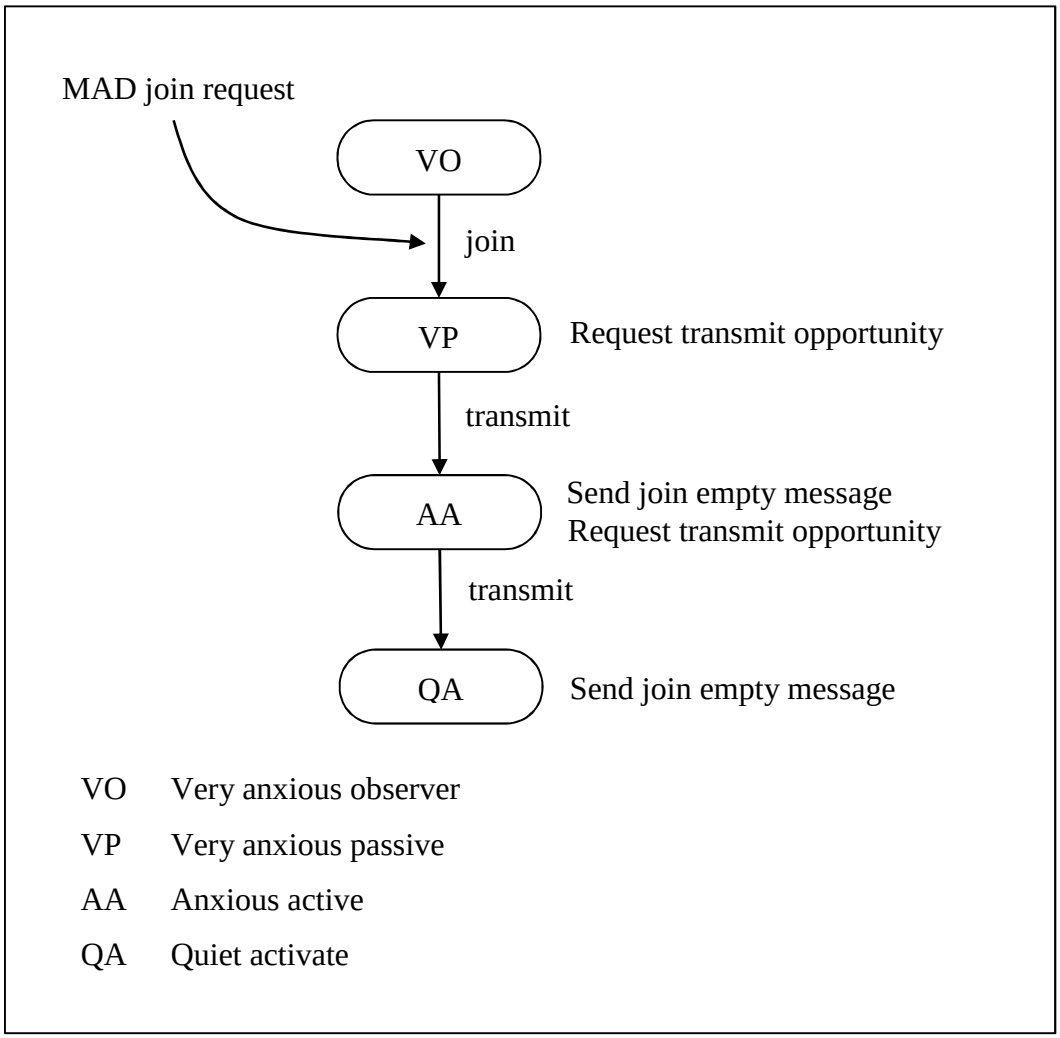

Figure 17: State transitions for the declaration of an attribute

Shown in Figure 17 above is a partial state transition diagram for an applicant state machine. It shows the transition of state when an attribute is declared. When a *MAD join request* service primitive is invoked (by MSRP for a *talker advertise* attribute, for example), the attribute is located within the participant. If the attribute cannot be located, the attribute is created and its applicant state machine is signalled with a *begin* event (as shown in Figure 16). Once the attribute has been located (or created), MAD signals the attribute's applicant state machine with a *join* event. This causes the applicant state machine to transition to the VP (*very anxious passive*) state which indicates that the

associated attribute is being declared but this has not yet been communicated to the bridged LAN. When the state machine transitions into the VP state, it requests an opportunity to transmit (see Section 3.2.1.3.1 “Transmit Opportunity”). Once the requested transmit opportunity arrives (after the join period timer expires), a *transmit* event is signalled to the applicant state machine. This causes the state machine to transition to the AA (*anxious active*) state which indicates that the state machine is declaring the attribute and has communicated this to the bridged LAN. The state machine adds a *join empty* message to an MRPDU which is sent to the bridged LAN. Following this a further transmit opportunity is requested. When the transmit opportunity arrives, another *transmit* event is signalled to the applicant state machine. This causes the state machine to transition to the QA (*quiet activate*) state, which indicates that the applicant is declaring the attribute and has sent at least one message communicating the declaration. The state machine adds a further *join empty* message to an MRPDU which is sent to the bridged LAN.

In the scenarios mentioned above, the *attribute event* communicated in the messages (of the MRPDUs) will indicate *join empty*. The *join empty attribute event* indicates that the attribute is declared by the participant, but is not registered by the participant (i.e., no other participant on the LAN has declared the same attribute). Section 3.2.1.3.3 “Registrar State Machine” details what happens when these messages are received and how the events are interpreted.

Shown in Figure 18 is a further partial state transition diagram for an applicant state machine. It shows the transition of states when an attribute declaration is withdrawn. Once an attribute is declared, and this declaration has been communicated to the bridged LAN, the applicant state machine is in the QA state (as shown in Figure 17). When a *MAD leave request* service primitive is invoked (by MSRP for a *talker advertise* attribute, for example), the applicant state machine for the attribute is signalled with a *leave* event, causing the state machine to transition to the LA (*leaving active*) state. This state indicates that it has communicated the attribute declaration to the bridged LAN but has subsequently received a *MAD leave request* and has yet to communicate this leave request to the bridged LAN. When the state machine transitions to this state, it requests a transmit opportunity. When the transmit opportunity arrives (after the join period timer’s time expires), a *transmit* event is signalled to the applicant state machine. The applicant state machine transitions to the VO state, and it adds an appropriate message to an MRPDU which is sent to the bridged LAN. The attribute’s *attribute event* communicated in the message will indicate *leave*. The *leave attribute event* indicates that the attribute declaration has been withdrawn. Section 3.2.1.3.3 “Registrar State

Machine" details what happens when this message is received by other participants and how the event is interpreted.

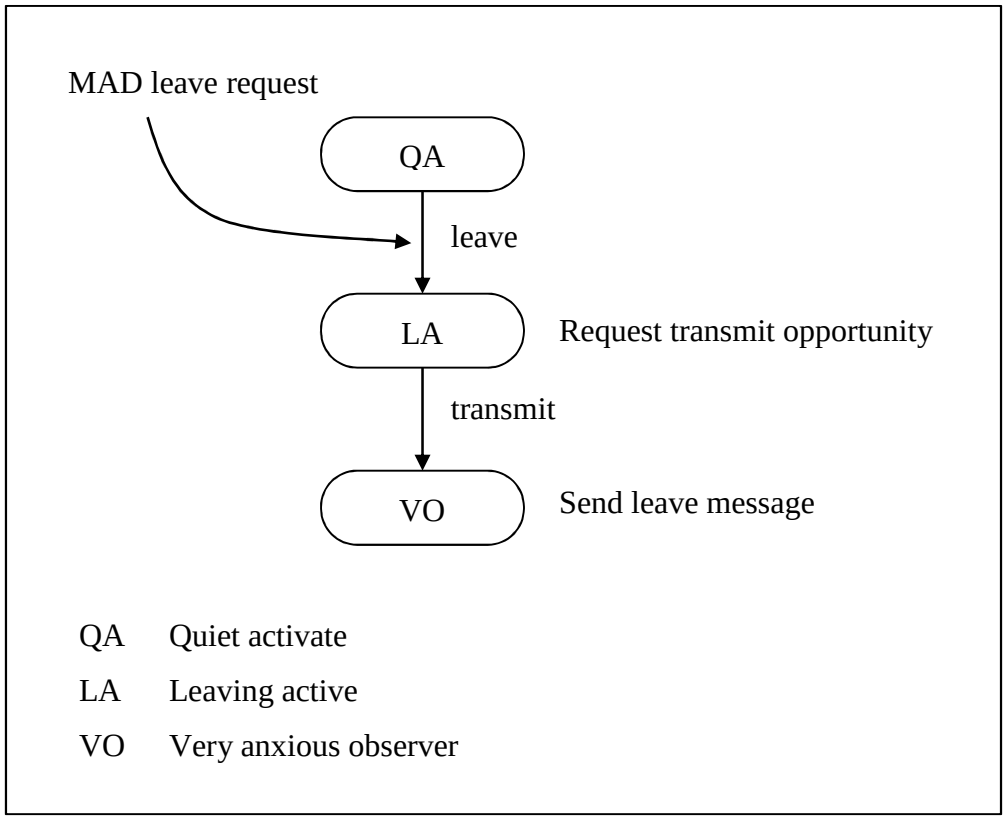

Figure 18: State transitions for the withdrawal of a declaration of an attribute

3.2.1.3.3 Registrar State Machine

The registration of an attribute is recorded by its registrar state machine for each participating end station and bridge port that receives an MRPDU. Its job is to record the declarations of attributes made by other MRP participants. The registrar state machine does not send any MRPDUs. The following states are implemented by each registrar:

- Registered
- Previously registered, but is now being timed out
- Not registered

As shown in Figure 19, when a registrar state machine is initially created (upon attribute creation), it is signalled with a *begin* event. The attribute could be created when it is declared, registered, or at some other time specific to an MRP application implementation. The *begin* event causes it to

transition to the MT (not registered) state which indicates that the attribute associated with the registrar state machine is not registered.

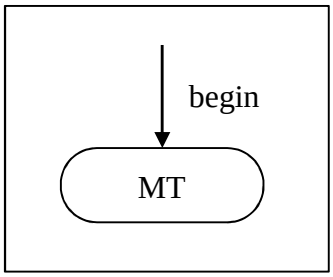

Figure 19: Initial registrar state machine state

Figure 20 shows a partial state transition diagram for a registrar state machine. If a single participant on a bridged LAN has declared an attribute, it will communicate this to the bridged LAN. All of the other participants' registrar state machines (on the same LAN) for the same attribute will be signalled with a *receive join empty* event (as the *attribute event* indicated in the received message is *join empty* (as discussed in Section 3.2.1.3.2 "Applicant State Machine")). This event indicates that the associated attribute that was received is declared, but not registered (by the participant that transmitted the MRPDU). Each registrar state machine transitions to the IN (registered) state. When this happens, a *MAD join indication* is signalled to MAP and the MRP application indicating the attribute type and attribute value.

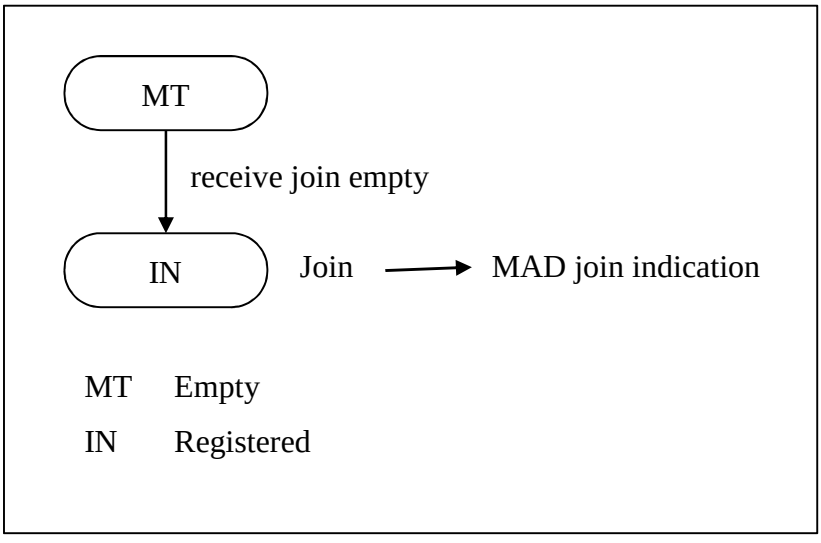

Figure 20: Registrar state after attribute registration

A timer is associated with each registrar state machine and operates when the associated attribute is being timed out. Figure 21 shows a further partial state transition diagram for a registrar state machine. If an MRPDU is received containing an attribute and it is indicated that the attribute declaration has been withdrawn (the *attribute event* is *leave*), the associated registrar state machine will be signalled with a *receive leave* event. This event causes the state machine to transition to the LV (previously registered, but is now being timed out) state and causes the timer associated with the registrar state machine to be started. Once the timer expires, it will signal a *leave timer* event to its registrar state machine, which causes the state machine to transition to the MT state. When the state machine transitions to the MT state, a *MAD leave indication* is signalled to MAP and the MRP application (e.g., MSRP).

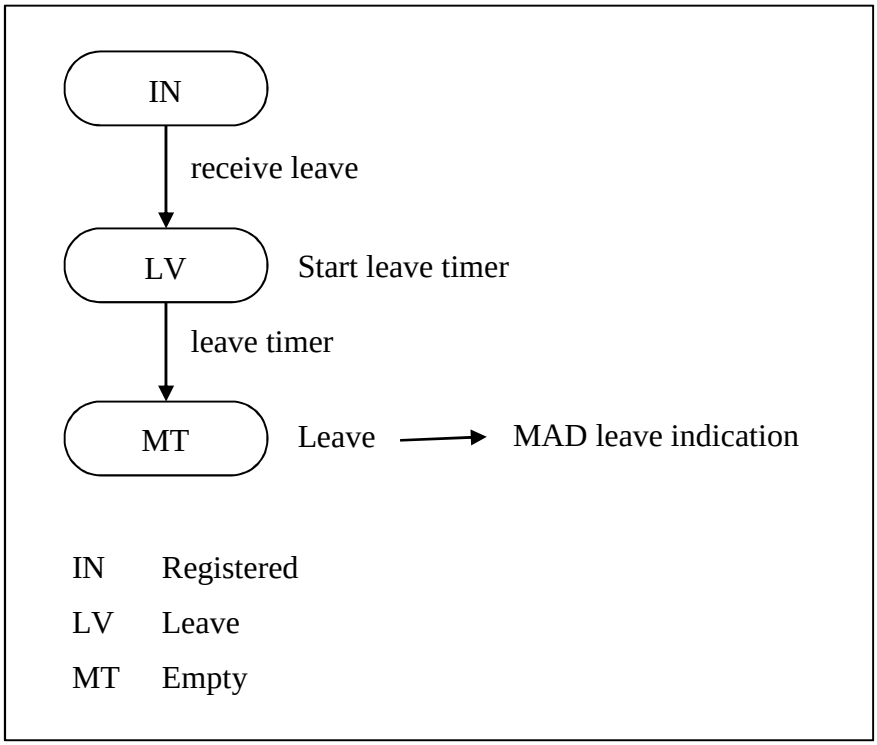

Figure 21: Registrar state after attribute deregistration

3.2.1.3.4 Attribute Declaration and Registration

Figure 22 shows an applicant state machine of a participant (on end station 1), and a registrar state machine of another participant (on end station 2). Also shown is how the state transitions and messaging of one state machine affects the other one when an attribute is declared. An MRP application, for example MSRP, invokes a *MAD join request* service primitive to indicate that it

wants to declare an attribute with a certain value. End station 1 declares an attribute (and thus its applicant state machine is shown), and this attribute is registered on end station 2 (and thus its registrar state machine is shown). Once the attribute's applicant state machine is created, it is in the VO state. The invocation of the *MAD join request* (on end station 1) results in the attribute's applicant state machine being signalled with a *join* event causing it to transition to the VP state. The applicant state machine then requests a transmit opportunity, and when given one, adds an appropriate message to an MRPDU. The *attribute event* associated with the attribute in the message is *join empty*.

When the message is received on the other participant (on end station 2), the attribute is located (or if not found, created and signalled with a *begin* event) and its registrar state machine is signalled with a *receive join empty* event causing the state machine to indicate that the attribute is registered. This results in a *MAD join indication* service primitive being signalled to the MRP application. When the registrar state machine receives the second *join empty*, it stays in the registered state, and does not signal a *MAD join indication* to the MRP application.

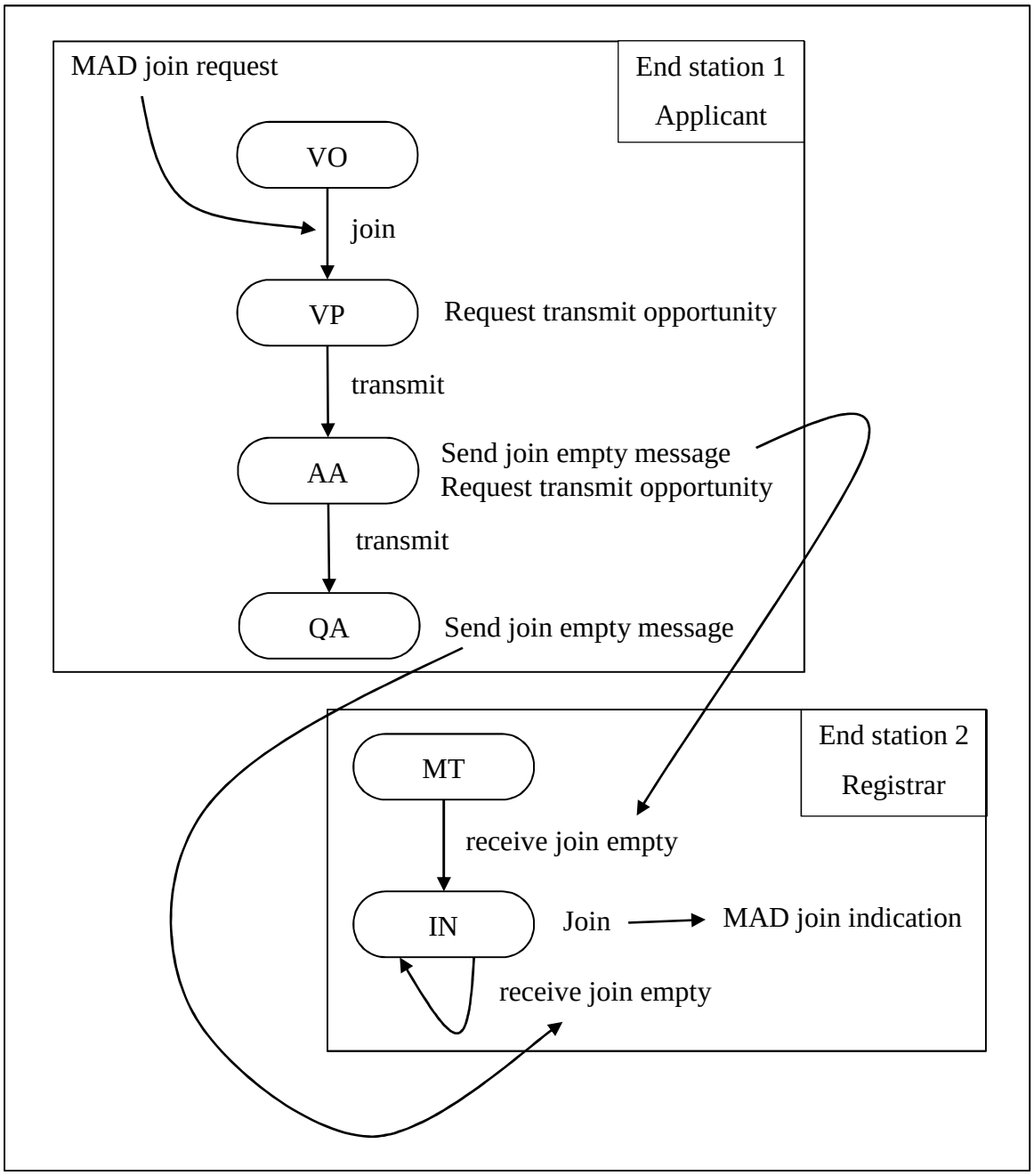

Figure 22: State machine transitions

3.2.1.3.5 Leave All State Machine

Associated with each MRP participant is a leave all state machine. The leave all state machine is responsible for periodically ensuring that all participants on a LAN deregister any stale attribute registrations. An attribute registration could become stale if, for example, the participant that declared the attribute suddenly lost network connectivity.

The leave all state machine can either be in the *active* state or *passive* state and it has associated with it a timer. When the leave all state machine is initially created, it is signalled with a *begin* event. This causes it to transition to the passive state and causes its timer to be started. This is shown in Figure 23.

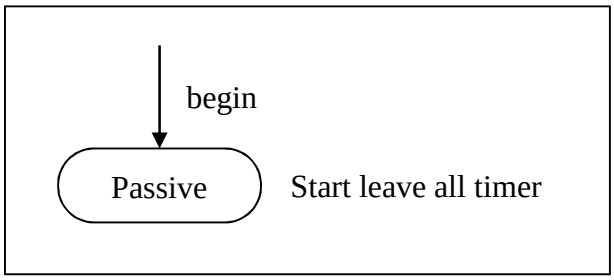

Figure 23: Leave all state machine initialisation

Figure 24 shows a partial state transition diagram for the leave all state machine. When its timer's time expires, a *leave all timer* event is signalled to the state machine. This causes the state machine to transition to the active state and causes a transmit opportunity to be requested, and the timer to be restarted. When the transmit opportunity arrives (when the join period timer's time expires), a transmit event is signalled to the state machine. This causes a *leave all* message to be added to an MRPDU.

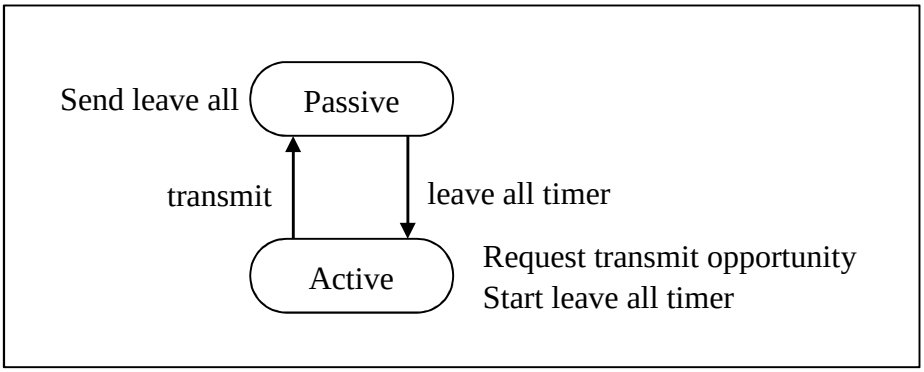

Figure 24: Leave all state machine timer expiring

A *leave all* message is used to communicate that all attribute registrations (of all participants on a LAN) will shortly be deregistered, and that all MRP participants need to reregister their attributes. The *leave all* message applies to all attributes. Any attributes that are registered by the MRP participant that transmitted the *leave all* message, as well as the participants that receive the *leave all* message, are timed out.

Only a single participant's leave all state machine needs to periodically transmit a *leave all* message. Since the MRP protocol is a distributed protocol, each participant is able to transmit *leave all* messages. When a participant receives a *leave all* message (that was generated by another participant's leave all state machine), its leave all state machine's timer is restarted without it generating a *leave all* message.

Figure 25 shows a leave all state machine (on end station 1), a registrar state machine (on end station 2), and an applicant state machine (on end station 3). Assume that the applicant state machine has declared an attribute, and has communicated this to a LAN. The registrar state machine (on end station 2) has registered the attribute. A leave all state machine's timer expires, which results in a *leave all* message being transmitted to the LAN (indicated by the number 1). A *receive leave all* event is signalled to all of the registrar and applicant state machines on the LAN (indicated by the number 2). The registrar state machine transitions to the LV state and starts its timer (as described in Section 3.2.1.3.3 "Registrar State Machine"). This causes the attribute registration to start being timed out.

The applicant state machine transitions to the VP state, requests a transmit opportunity, and eventually transmits a *join empty* message (indicated by the number 3). This *join empty* message is received by the registrar state machine (indicated by the number 4) which causes its timer to stop. The registrar then transitions back into the registered state. If the registrar's timer were to expire, the attribute would have been deregistered. This would have happened, for example, if the station that declared the attribute had lost network connectivity as the registrar would not have received the *join empty* message. This ensures that when a participant declares attributes and it leaves the LAN without withdrawing these attribute declarations, the stale attributes are deregistered from other participants.

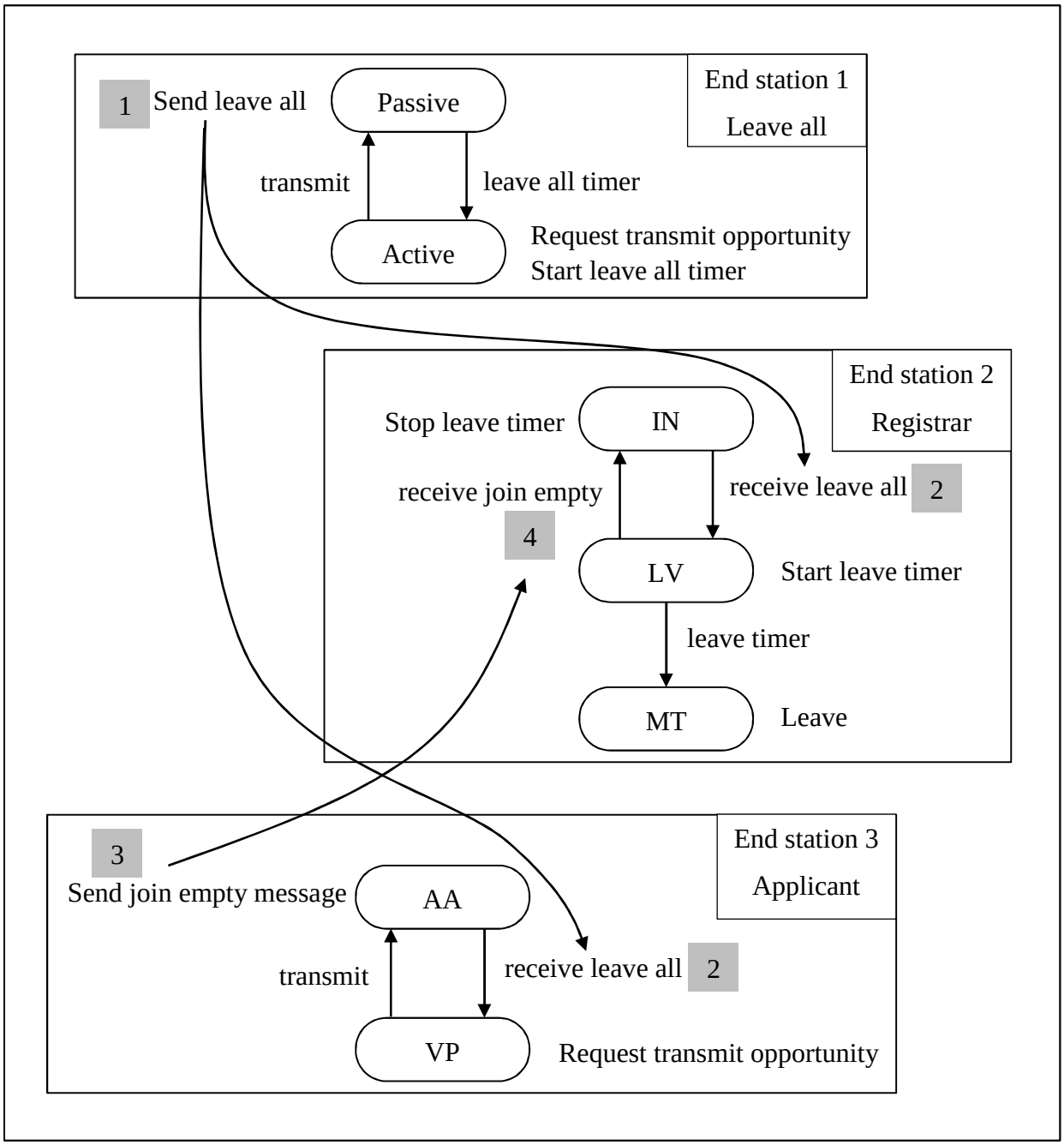

Figure 25: Leave all state machine timer expiring

3.2.1.3.6 Periodic Transmission State Machine

The periodic transmission state machine of each participant periodically causes the states of its declared attributes to be sent to a LAN to ensure that the attributes are registered on other participants. The functionality of the periodic transmission state machine may be disabled and enabled at any time during its existence. MSRP, for example, does not make use of the periodic transmission state machine.

There is a timer associated with the periodic transmission state machine, and it periodically signals a *periodic* event to the participant's applicant state machines. The periodic transmission state machine can either be in the *active* or *passive* state. When the state machine is created, it is signalled with a *begin* event. This causes the state machine to transition to the active state, and causes its timer to be started, as shown in Figure 26.

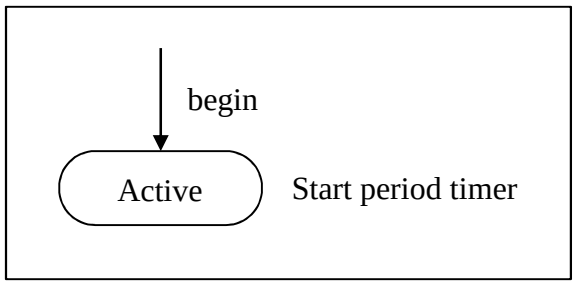

Figure 26: Periodic transmission state machine initialisation

When the periodic transmission state machine's timer expires, the periodic transmission state machine is signalled with a *periodic timer* event. The state machine remains in the active state, but causes the state machine to signal all applicant state machines with a *periodic* event and causes its timer to be restarted, as shown in Figure 27.

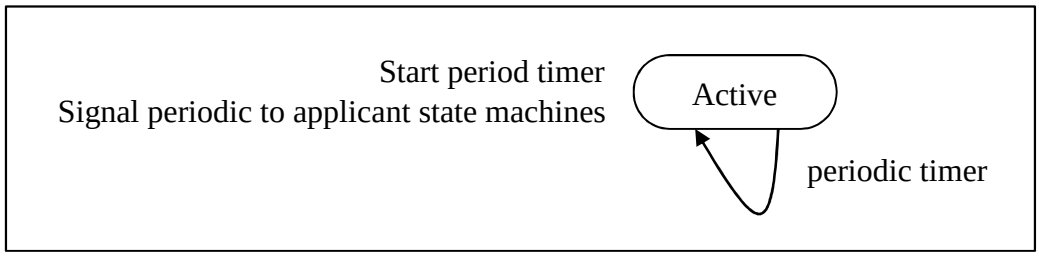

Figure 27: Periodic state machine timer expiring

If, for example, an applicant state machine has recorded the declaration of an attribute, and has transitioned to the QA state (as shown in Figure 17) and is signalled with a *periodic* event, it will cause the applicant state machine to transition to the AA state. This transition causes a transmit opportunity to be requested. When this transmit opportunity arrives, the state machine transitions back to the QA state and communicates the attribute registration state to the bridged LAN. Figure 28 shows an applicant state machine being signalled with a periodic event.

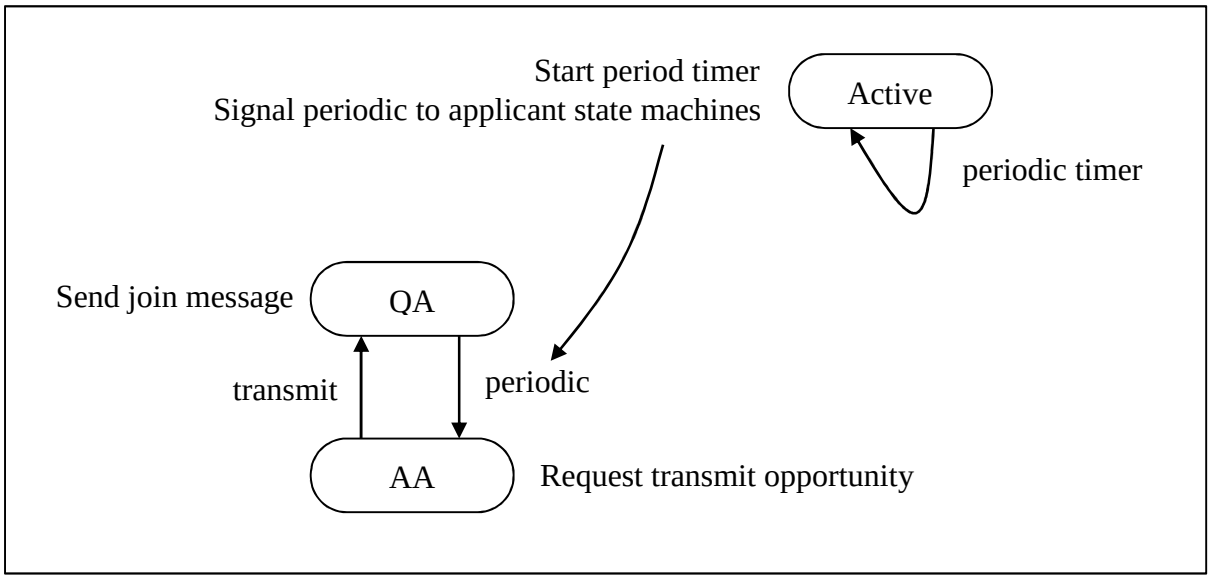

Figure 28: Applicant state machine being signalled with a periodic event

3.2.1.4 Protocol Timers

Various timers are used by MRP to cause actions to take place after a defined period of time:

- The join period timer is used to control the time between transmit opportunities that are granted to applicant and leave all state machines that requested transmit opportunities. There is a join period timer per MRP participant.
- The *leave period timer* is used to control the amount of time that a registrar state machine will wait in the "previously registered, but is now being timed out" state before it transitions to the "not registered" state.
- The *leave all period timer* controls the frequency at which the leave all state machine generates *leave all* messages. One of these timers exists per MRP participant.
- The *periodic transmission timer* controls the frequency at which the periodic transmission state machine indicates to the applicant state machines that its timer has expired. This timer is used to stimulate periodic transmission and ensure that current attribute declarations are registered on all participants on a LAN. One of these timers exists on a per-port basis.

Given in Table 2 are the default timer values.

Timer	Default Value (milliseconds)
Join period timer	200
Leave period timer	600-1000
Leave all period timer	10000
Periodic transmission timer	1000

Table 2: The default MRP timer values

The correct operation of the MRP protocol is not critically dependent on the values of the various timers associated with a participant's state machines. The protocol will, however, operate more efficiently if certain relationships are maintained between timer values used by peer participants:

- The value of the leave period timer of the registrar should be at least twice the maximum join period timer, and also at least six times the timer resolution. This gives applicant state machines sufficient time to communicate their attribute declarations to a LAN to ensure that registrar state machines do not deregister their attributes unnecessarily. If, for example, a registrar state machine (on a port attached to a LAN) and an applicant state machine (on another port on the LAN) for the same attribute receive a *leave all* message, the registrar will start timing the attribute out. The applicant, once it has received the *leave all* message and has been given a transmit opportunity, will communicate the attribute declaration to the LAN, causing the registrar to stop its leave period timer (and it will move back into the registered state). The applicant state machine has to be given sufficient time to communicate the attribute declaration before the attribute is unnecessarily deregistered.
- To minimize the potential volume of *leave all* messages, the value chosen for the leave all period timer of the leave all state machine should be large relative to the leave period timer of registrar state machines.

3.2.1.5 MRP Application Addressing

Each specific MRP application uses a unique EtherType in order to identify the application protocol. MRP applications also use a specific MAC address in the destination address field of their MRPDUs. This MAC address is used to identify the MRP application. The specific set of multicast addresses that have been set aside for MRP applications have the following properties:

- If an MRP application is using a particular MAC address, any frames that are destined to that address are not forwarded by the bridge component. These frames are passed to the relevant MRP applications on the bridge, which process them and decide if the frames should be propagated (using the MAP component).
- For any of the addresses that are not used by any MRP applications, the frames destined for that address and received by the bridge are forwarded to all other ports.

3.2.2 Multiple MAC Registration Protocol

MMRP is an MRP application that allows end stations and bridges to register and deregister MAC address information with the bridges and end stations of a bridged LAN. The services provided by MMRP allow end stations to dynamically control reception of frames destined to unicast and multicast MAC addresses. MMRP allows a bridge or end-station to request that frames with a given destination unicast or multicast MAC address be forwarded to it.

Figure 29 shows an example bridged LAN that is composed of a number of bridges, LANs and end stations. Each port on the bridged LAN has associated with it an MMRP participant. A number of end stations have registered (via MMRP) the fact that they would like to receive frames with a destination multicast address *m* (these end stations are marked with the letter M). The registration and propagation of this registration information is handled by MRP. If *any* end station sends out a frame with a destination multicast MAC address *m* (in this example, the end station marked with the letter T), the bridges of the network will forward the frame out of their ports where the MAC address *m* has been registered. Bridges receiving the frames will not forward the frames out of ports where the MAC address is not registered.

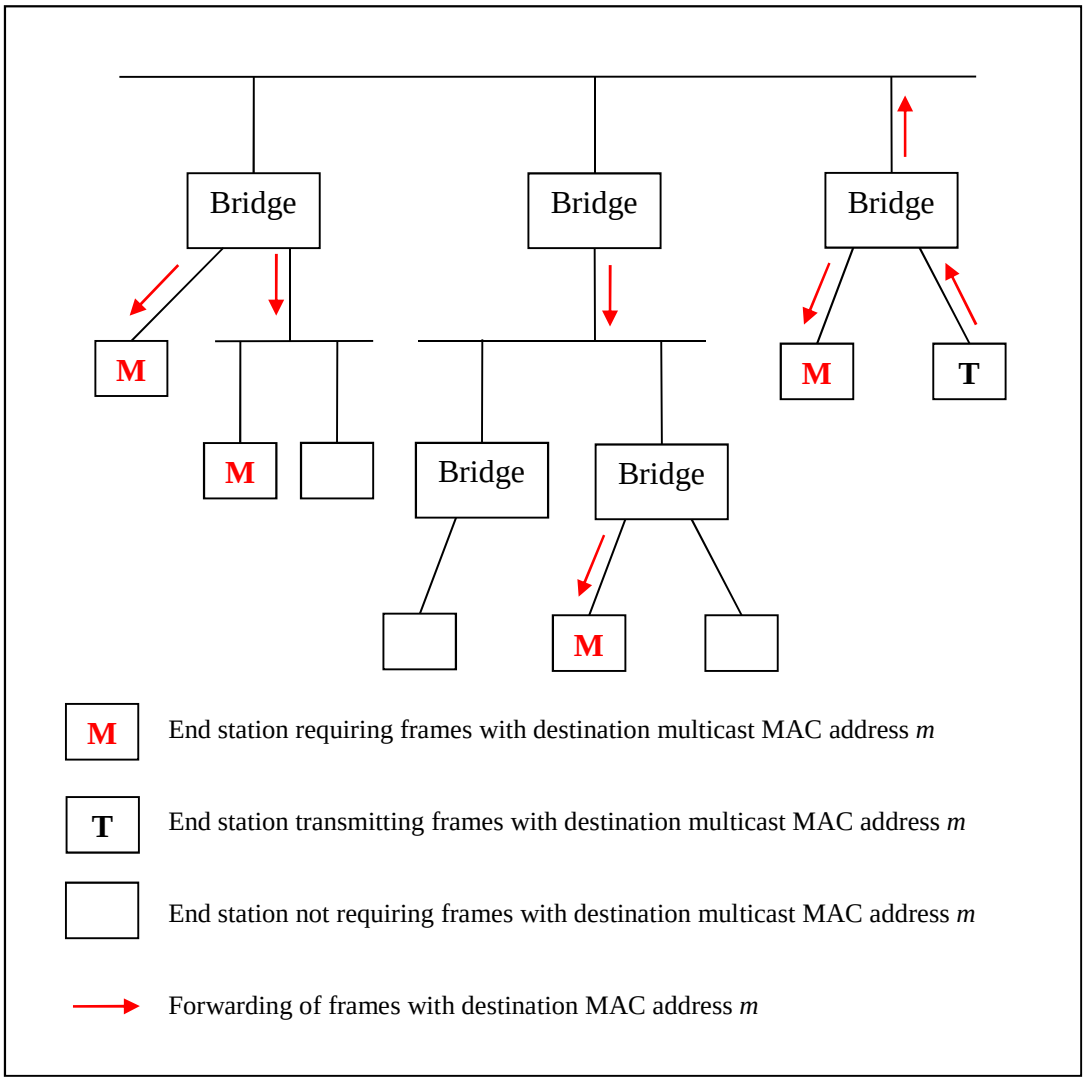

Figure 29: Example MMRP registrations

MMRP allows for multicast frames to be forwarded only to parts of a bridged LAN where they are needed (as opposed to being broadcast throughout a bridged LAN) thus saving bandwidth. End stations are able to utilise information registered by MMRP to provide *source pruning*. This allows end stations that are sources of frames with destination multicast MAC addresses to suppress the transmission of these frames if the information registered by MMRP indicates that there are no recipients of the frames.

Figure 30 shows the architecture of MMRP for a two-port bridge and an end station. Each MMRP participant consists of an MMRP application and a MAD component. The MMRP application component is responsible for the semantics of MMRP attribute registration and deregistration. It interacts with the filtering database and adds or deletes appropriate entries to ensure that the requests

of MMRP are carried out. The MAD component executes the MRP protocol (as discussed in Section 3.2.1 “Multiple Registration Protocol”). For network bridges, MMRP attributes are propagated between the various MMRP participants via a MAP component.

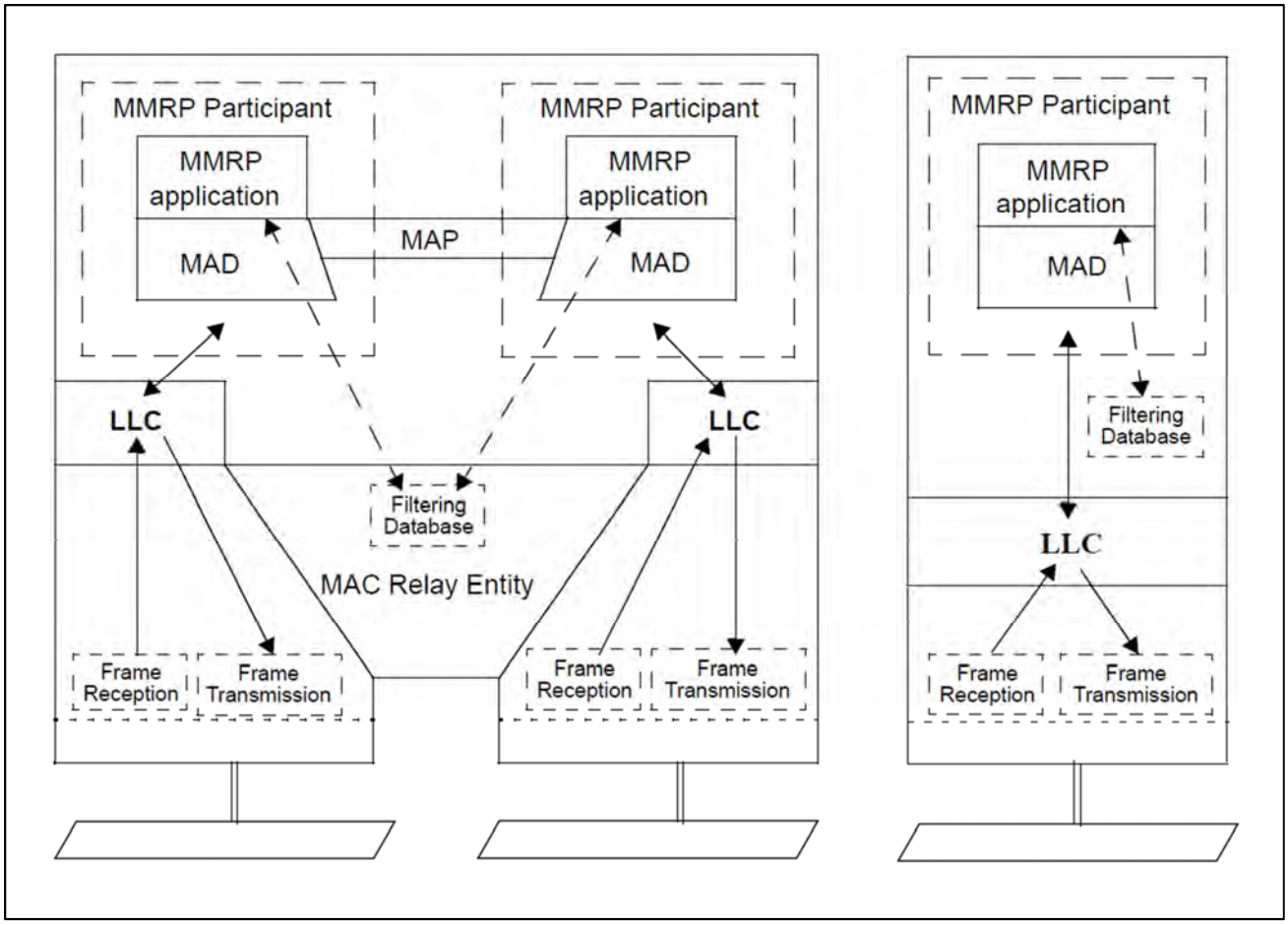

Figure 30: MMRP architecture [71]

3.2.2.1 MMRP Attributes and Service Primitives

MMRP uses the attribute declaration and propagation mechanisms provided by MRP in order to register its attributes on bridges and end stations. MMRP defines a MAC attribute type and the following service primitives:

- *Register MAC address (MAC)*
- *Deregister MAC address (MAC)*

3.2.2.1.1 Registering and Deregistering MAC Addresses

MMRP provides the ability to dynamically register and deregister MAC addresses via a *register MAC address* and a *deregister MAC address* service primitive. The registering and deregistering of MAC addresses allow for control over the reception of frames. Each of these service primitives has a MAC address parameter. This MAC address parameter is six octets long and represents a MAC address.

When a *register MAC address* service primitive is issued to an MMRP participant, it issues a *MAD join request* to its MAD component indicating an attribute type of MAC with an attribute value equal to the requested MAC address. Invoking the *MAD join request* results in a MAC attribute being declared. The MAC attribute is six octets long and is used to represent a MAC address. This declaration is communicated to the LAN and registered by other MMRP participants on the LAN. Bridges propagate the registration to their other ports (as discussed in Section 3.2.1 “Multiple Registration Protocol”).

Figure 31 shows an end station application registering a MAC address. This application could be a video application that is interested in receiving a stream of frames destined to a multicast MAC address, for example. The application interfaces with the MMRP participant and invokes the participant’s *register MAC address* service primitive indicating the multicast MAC address of interest (indicated by the number 1). The MMRP participant handles the registration and requests that MAD declare an attribute (via the *MAD join request* service primitive) of type MAC with a value equal to the indicated MAC address (indicated by the number 2). The MAD component declares the attribute and is responsible for communicating the attribute declaration to the LAN (via an MRPDU) such that it is made known to the bridged LAN (indicated by number 3). Section 3.2.2.1.2 “Responding to Registration Events” discusses what happens when the attribute is registered on other stations on the LAN.

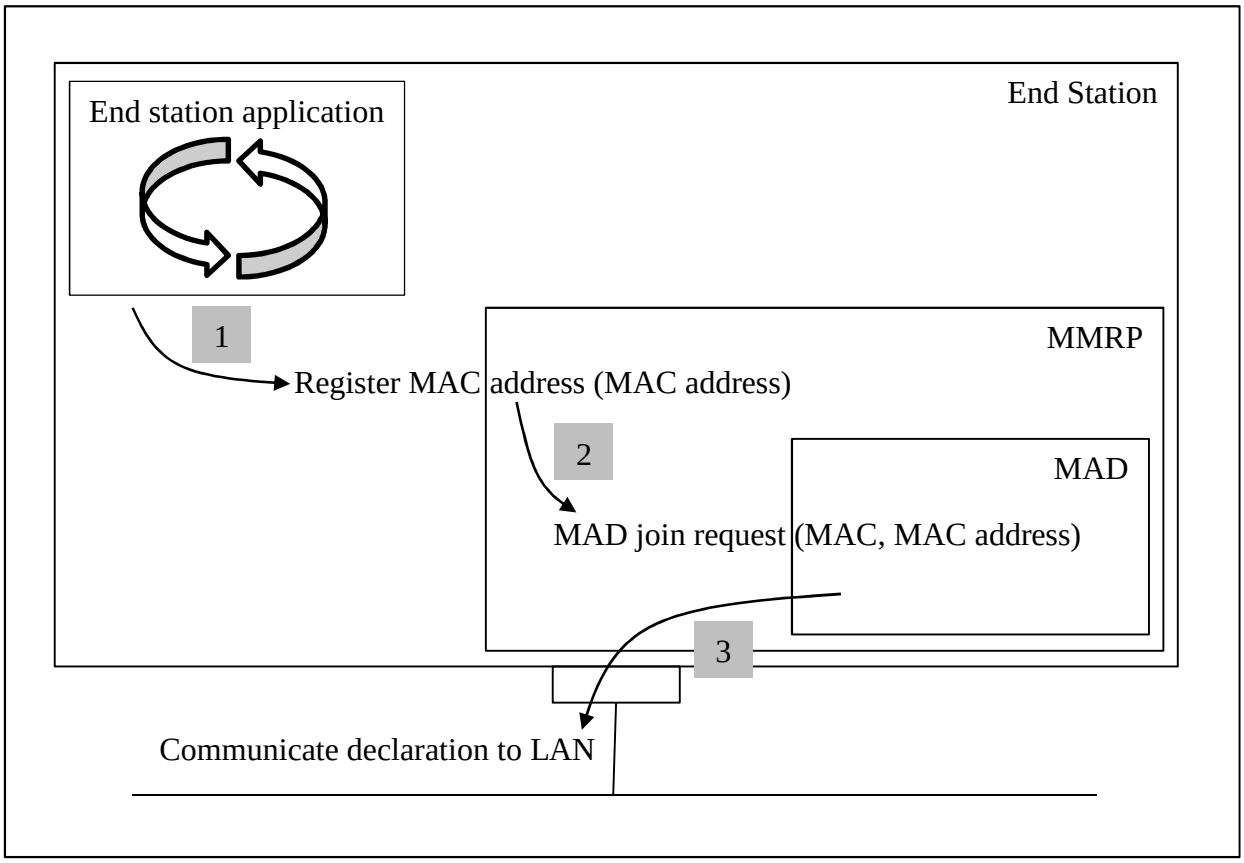

Figure 31: ***Register MAC address***

When a *deregister MAC address* service primitive is issued to an MMRP participant, it issues a *MAD leave request* to its MAD component indicating an attribute type of MAC with an attribute value equal to the requested MAC address. Invoking the *MAD leave request* results in the attribute declaration being withdrawn, and this withdrawal being communicated to the LAN.

Figure 32 shows an example of an end station application deregistering a MAC address. Once again, this could be a video application, but this time it no longer wishes to receive the stream of frames destined to the multicast MAC address. This application interfaces with the MMRP participant and invokes its *deregister MAC address* service primitive indicating the destination MAC address of the frames that it no longer wishes to receive (indicated by the number 1). The MMRP participant issues a *MAD leave request* to its MAD component indicating an attribute of type MAC and an attribute value equal to the requested MAC address (indicated by the number 2). The MAD component then withdraws the MAC attribute declaration and communicates this attribute withdrawal to the LAN (via an MRPDU). Section 3.2.2.1.2 “Responding to Registration Events” discusses what happens when the attribute is deregistered on other stations on the LAN.

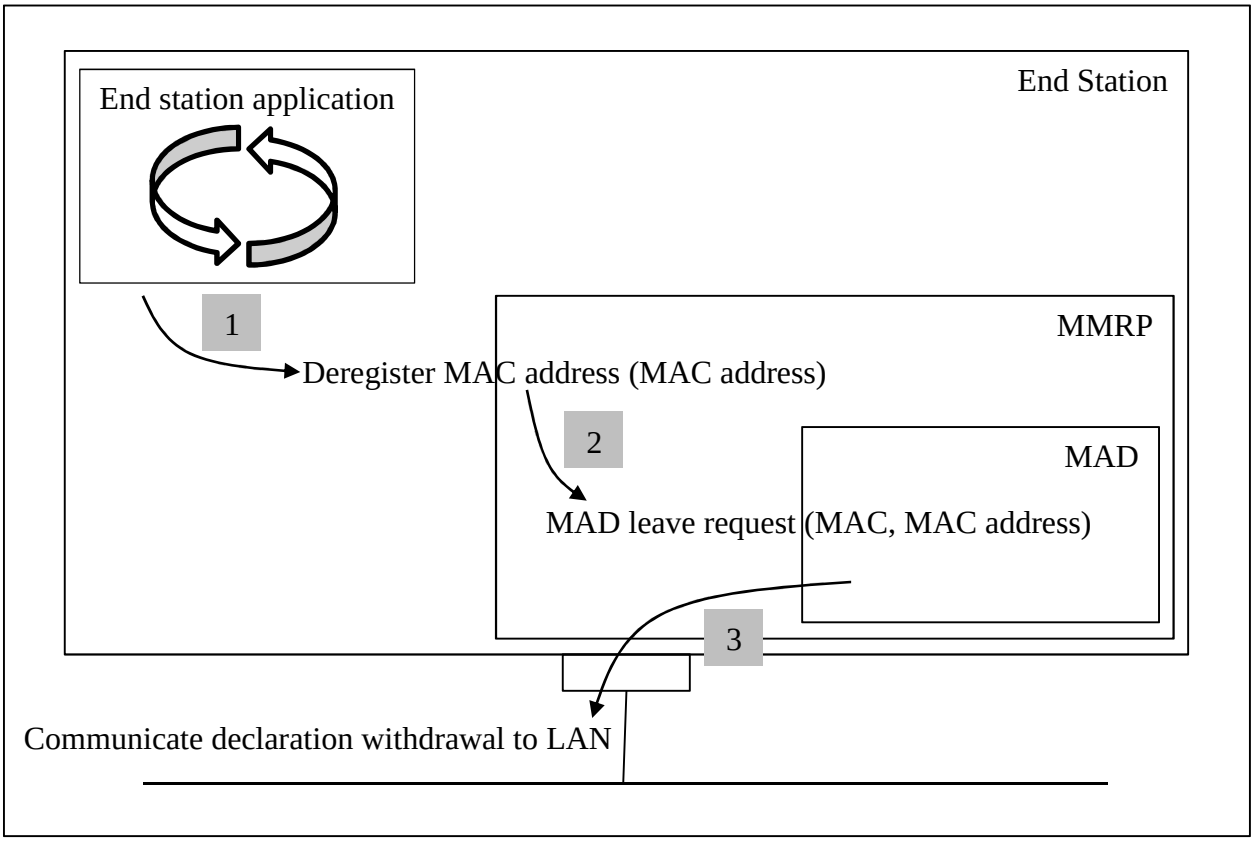

Figure 32: ***Deregister MAC address***

3.2.2.1.2 Responding to Registration Events

Section 3.2.2.1.1 “Registering and Deregistering MAC Addresses” discussed how the registration of MAC addresses are initiated and initially communicated to a LAN. When these attributes are registered by other MMRP participants (residing on bridges and end stations) on a bridged LAN, these stations need to be configured appropriately to honour the requests of the registrations.

Figure 33 shows a bridged LAN. The end station marked with the letter M is interested in receiving frames that are destined to multicast MAC address *m*. It registers the MAC address by declaring a MAC attribute with an attribute value equal to the MAC address *m* (indicated by the number 1). The attribute is communicated to the LAN and is registered on the port of the bridge (indicated by the number 2). The bridge propagates the attribute registration to its other ports and declares the attribute on these ports (indicated by the number 3). These ports communicate the attribute declarations to their LANs which results in the attributes being registered on the ports of the stations of their LANs

(indicated by the number 4). This cycle is continued until the MAC attribute is registered on end stations and cannot be propagated any further.

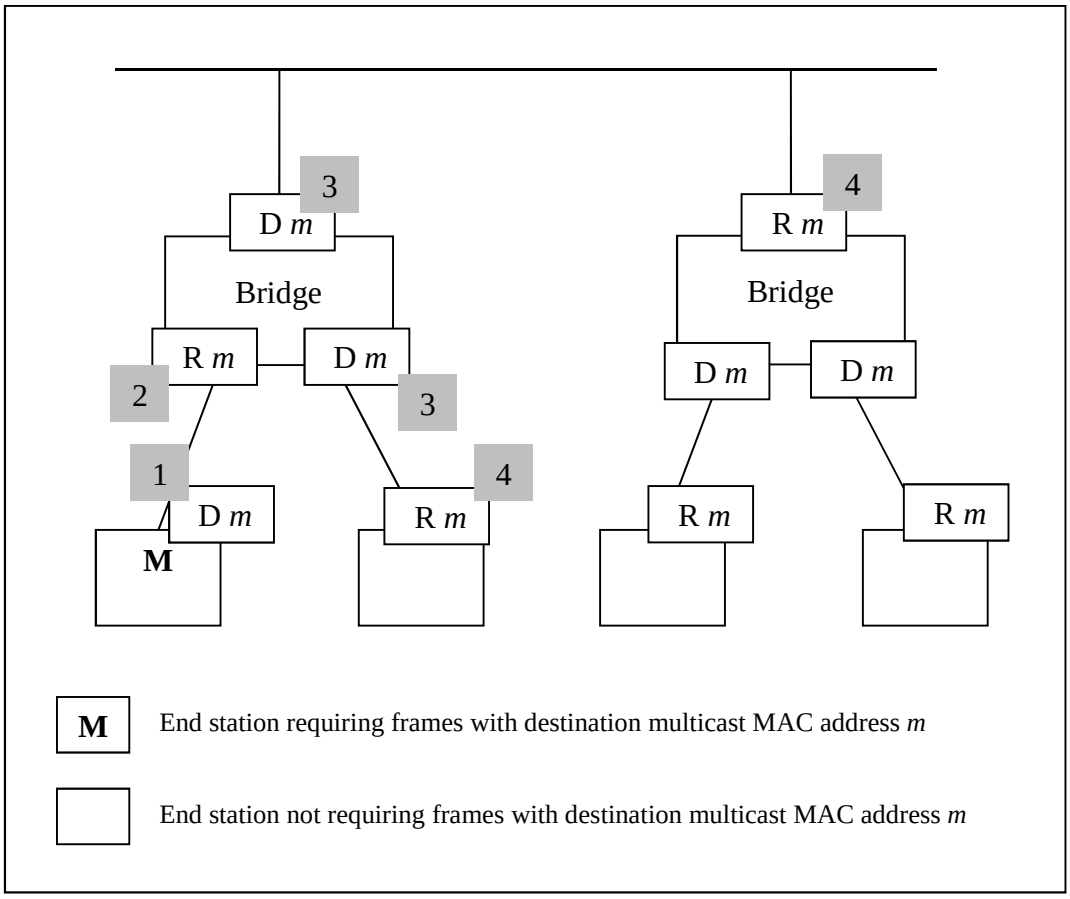

Figure 33: MAC address registration

MMRP responds to the registration and deregistration of MAC address information made by remote bridges and end stations. When a MAC attribute is registered by MRP, MMRP receives notification of this via its MAD component's *MAD join indication* service primitive. When a MAC attribute is deregistered by MRP, MMRP receives notification of this via its MAD component's *MAD leave indication* service primitive.

When MMRP receives a *MAD join indication*, a bridge or end station is configured such that it will forward any received frames (that meet the criteria of the MAC address attribute) out of the port on which the attribute is registered. Shown in Figure 34 is an example of a two port bridge with an MMRP participant receiving an MRPDU communicating an MMRP attribute declaration by a remote station (indicated by the number 1). The MMRP participant's MAD component receives and processes the MRPDU. It registers the attribute and invokes a *MAD join indication* to indicate to

MMRP the registration of the attribute. MMRP adds an appropriate entry to the bridge's filtering database to ensure that frames are forwarded correctly. If, for example, an end station attached to the lower port of the two port bridge has requested that all frames destined to multicast MAC address *m* be forwarded to it, an entry is added to the filtering database that associates the multicast MAC address *m* and the port. The forwarding process is then able to use this information when making forwarding decisions for any received frames destined to multicast MAC address *m*. It will forward any frames with that address out of the associated port.

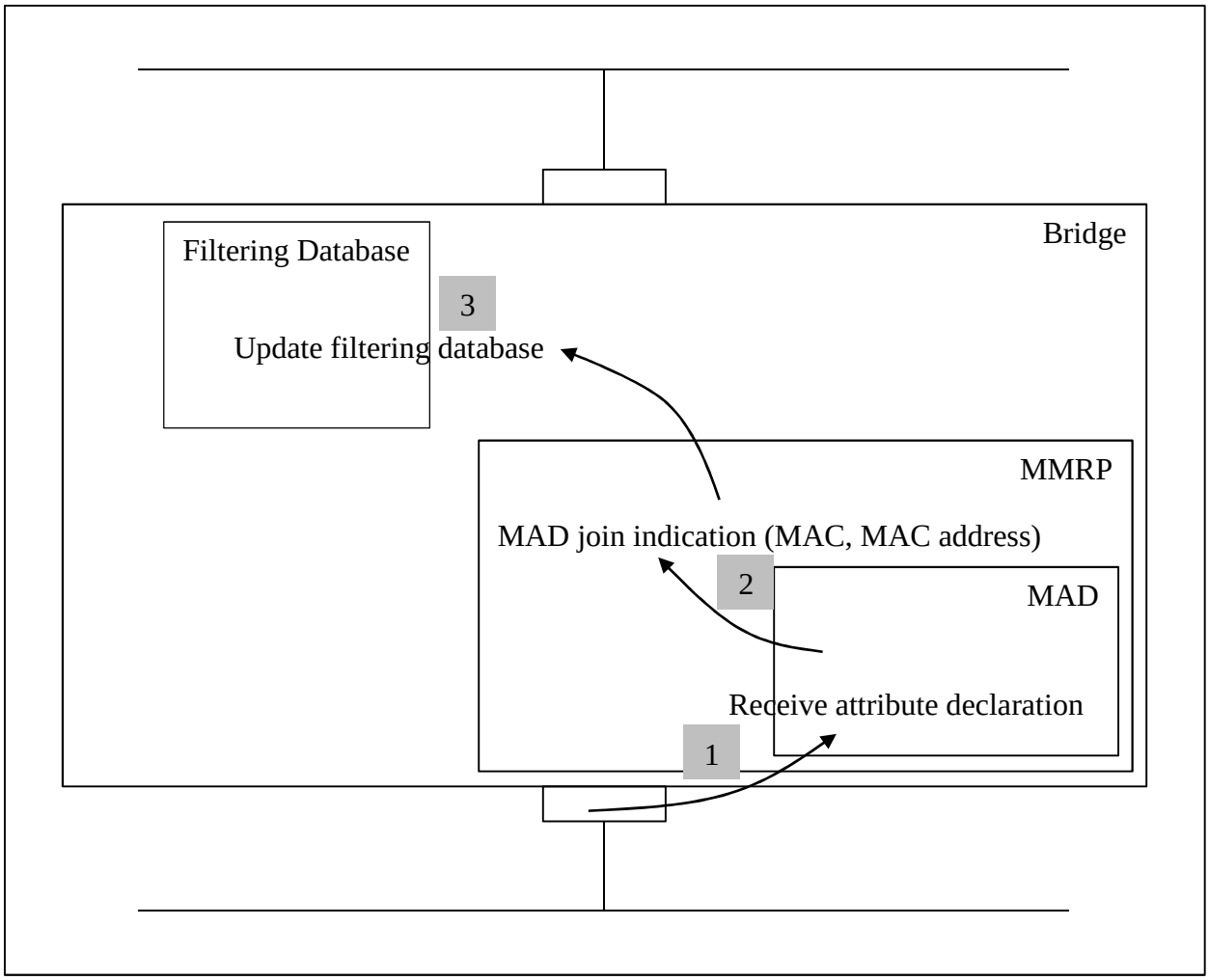

Figure 34: Two port bridge receiving an MMRP attribute declaration

When MMRP receives a *MAD leave indication*, a bridge or end station is configured such that it will filter any frames that meet the criteria of the MAC address parameter. Shown in Figure 35 is an example of a two port bridge (that has MMRP participants associated with its ports) that has received an MRPDU indicating that an MMRP attribute declaration has been withdrawn. The MMRP participant's MAD component receives and processes the MRPDU, deregisters the attribute and

notifies the MMRP component by invoking the *MAD leave indication* service primitive. The MMRP component updates the filtering database such that it no longer forwards the specified frames out of the associated port.

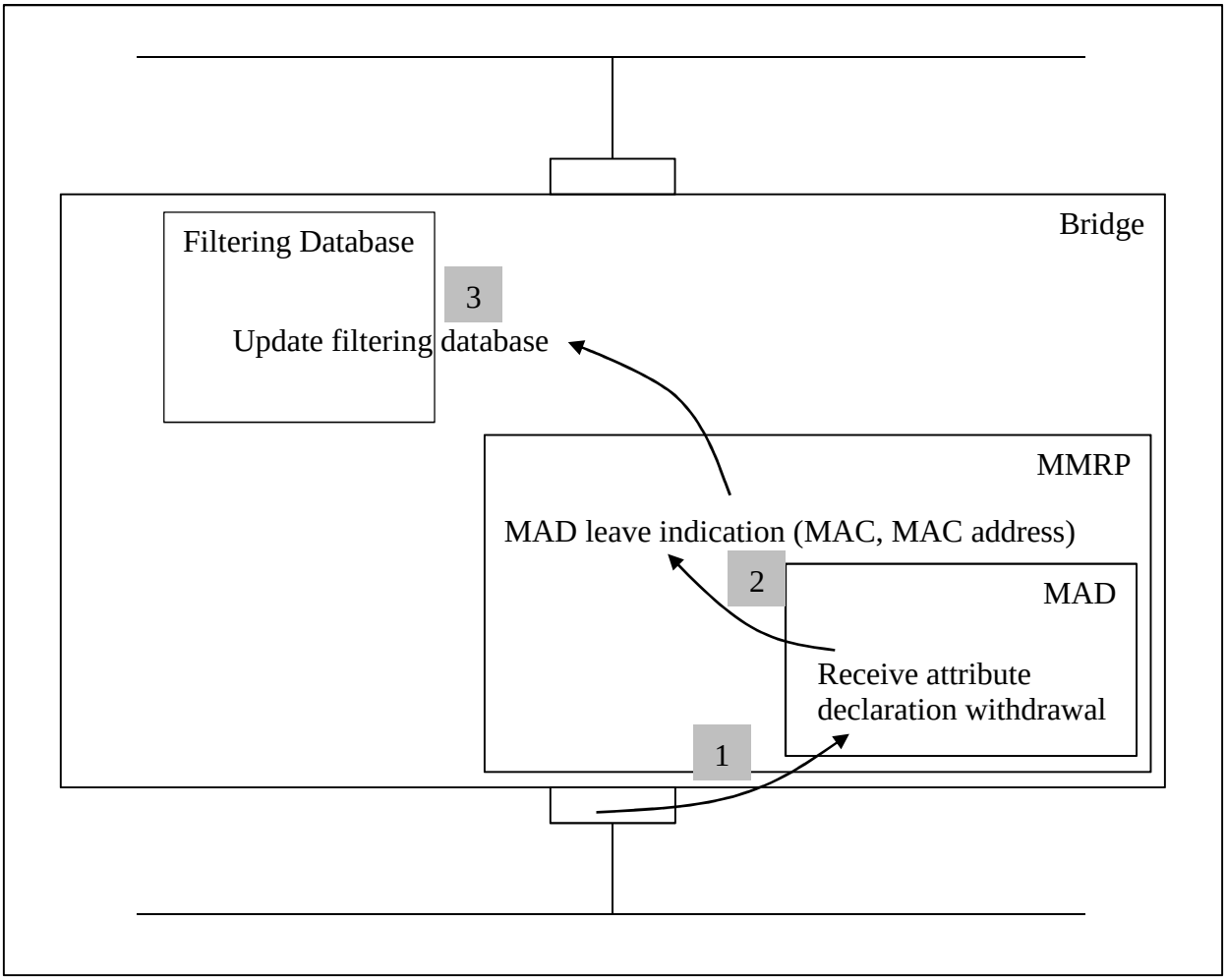

Figure 35: MMRP *MAD leave indication*

3.2.3 Multiple VLAN Registration Protocol

MVRP is an MRP application that allows for dynamic VLAN registrations in bridged LANs. Both end stations and bridges are allowed to create and withdraw attribute declarations that allow them to register themselves as a member of a VLAN, and to deregister that VLAN membership.

End stations are able to register their VLAN membership (via MVRP) with the other MVRP participants on the LAN segment to which they are connected. Each MVRP participant that receives one of these MVRP attribute declarations updates itself to indicate that the VLAN is registered on

the port that received the attribute declaration. VLAN-aware bridges register and propagate VLAN membership on their ports. Incoming registrations and de-registrations are used to update bridge VLAN membership information.

VLAN membership registration on end stations allows for *source pruning*: if an end station has frames to transmit on a particular VLAN, but that VLAN does not consist of any members, the frames will not be transmitted onto the LAN.

Figure 36 shows an example of a bridged LAN. A number of end stations have indicated that they would like to be a member of VLAN *v* (these end stations are marked with a V). These end stations have requested membership to the VLAN *v* via MVRP. This allows for bridges to know where to forward frames that are part of VLAN *v*. If a transmitting end station (in this example, the end station marked with a T) transmits a broadcast frame as part of VLAN *v*, the bridges will only forward the frame on the bridge ports where VLAN *v* has been registered.

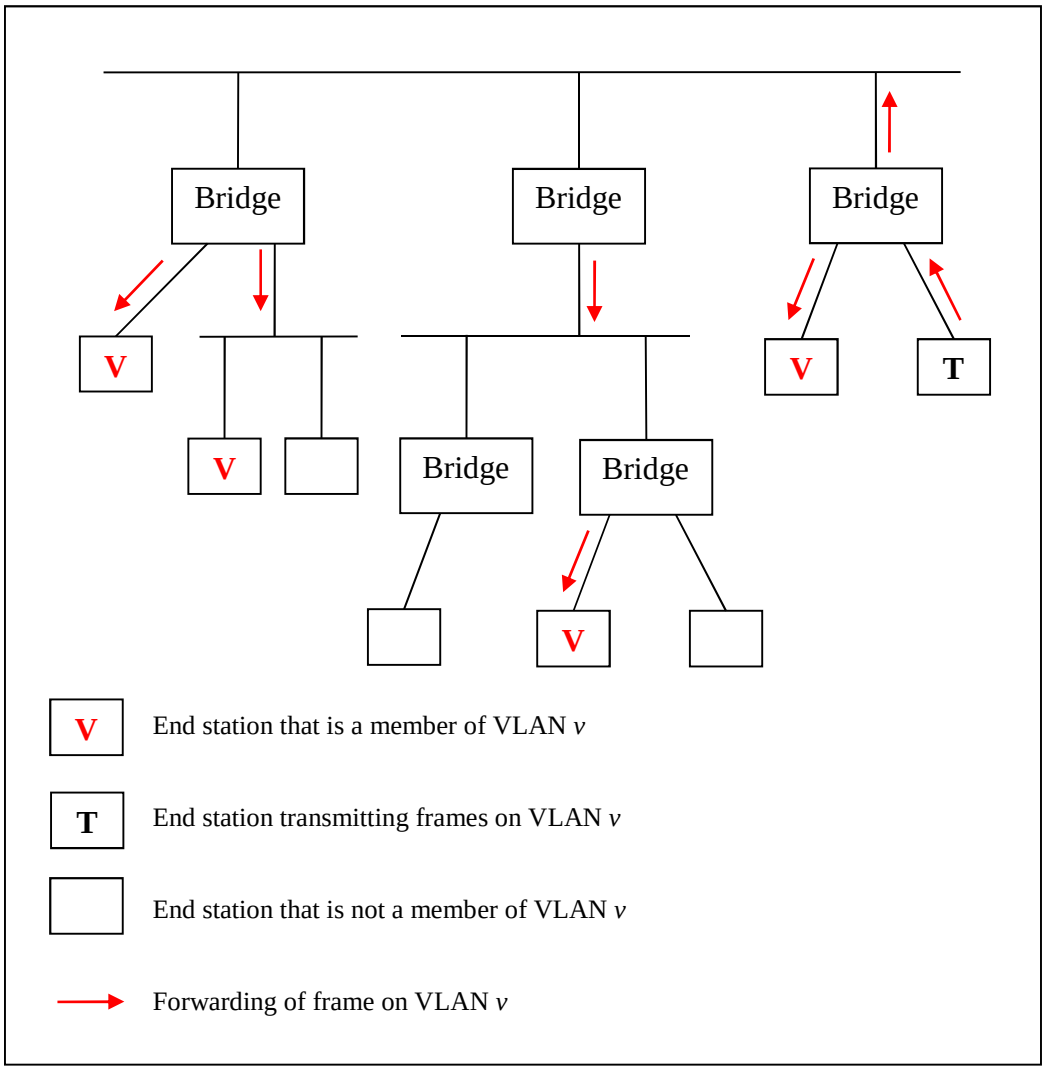

Figure 36: Example MVRP registrations

Shown in Figure 37 is the architecture of MVRP for a two-port bridge and an end station. There exists a single MVRP participant per port. An MVRP participant consists of an MVRP application, a MAD component, and a MAP component.

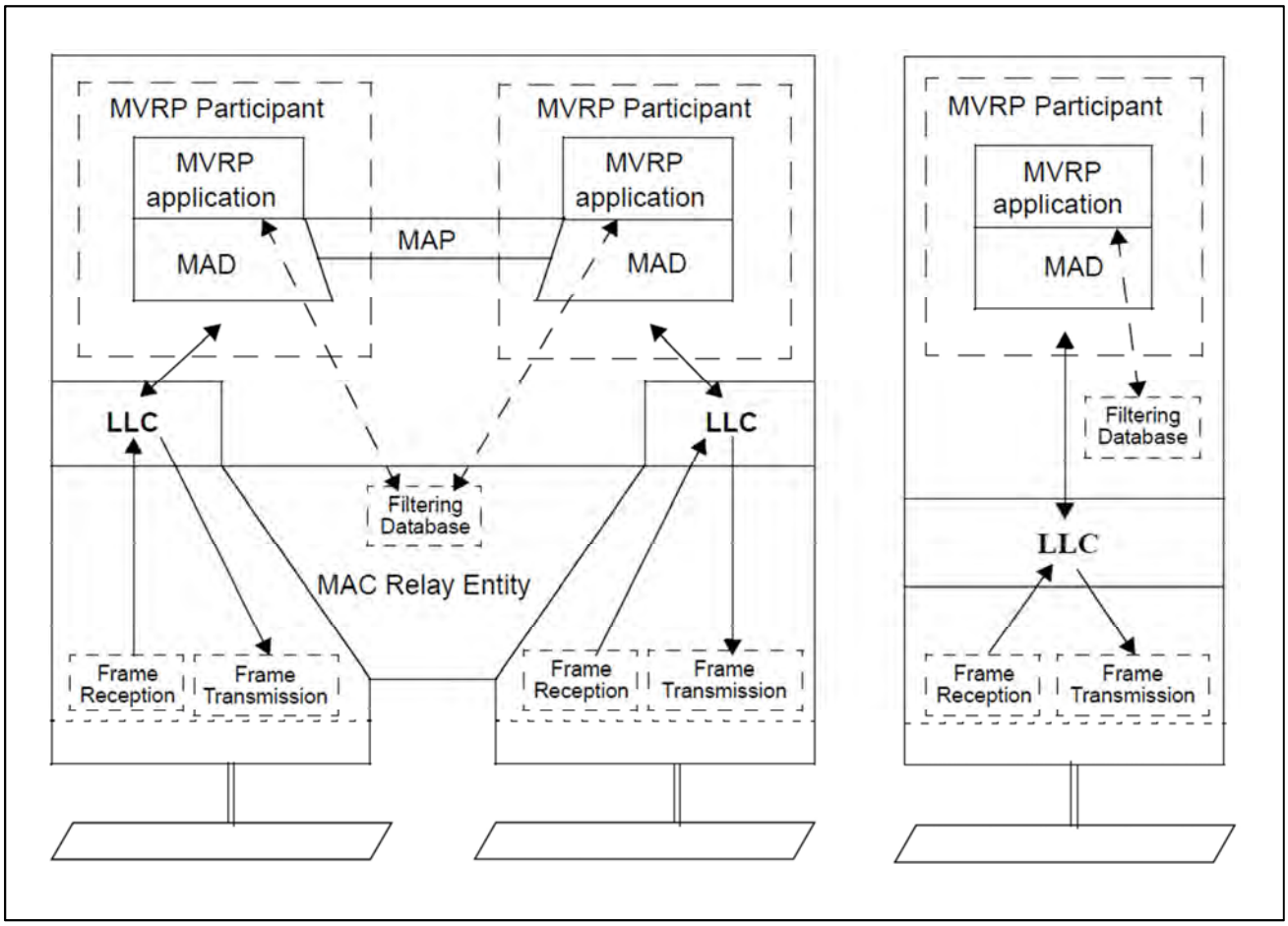

Figure 37: MVRP architecture

3.2.3.1 MVRP Attribute and Service Primitives

MVRP defines a *VID* attribute and two service primitives:

- *Register VLAN member (VID)*
- *Deregister VLAN member (VID)*

The *register VLAN member* service primitive allows for VLAN membership registration, and the *deregister VLAN member* service primitive allows for VLAN membership deregistration. Each of these service primitives has a VID parameter that represents a 12-bit VLAN identifier.

3.2.3.1.1 Registering and Deregistering VLANs

If a station wishes to become a member of a particular VLAN, it will issue a *register VLAN member* service primitive to MVRP with the VLAN ID that identifies the VLAN that it wishes to become a

member of. MVRP then issues a *MAD join request* to its MAD component indicating an attribute type of VID with an attribute value equal to the requested VLAN ID. This causes MAD to declare a VID attribute, the attribute declaration to be communicated to the LAN, and for the attribute to be registered on other ports attached to the LAN. This is shown in Figure 38.

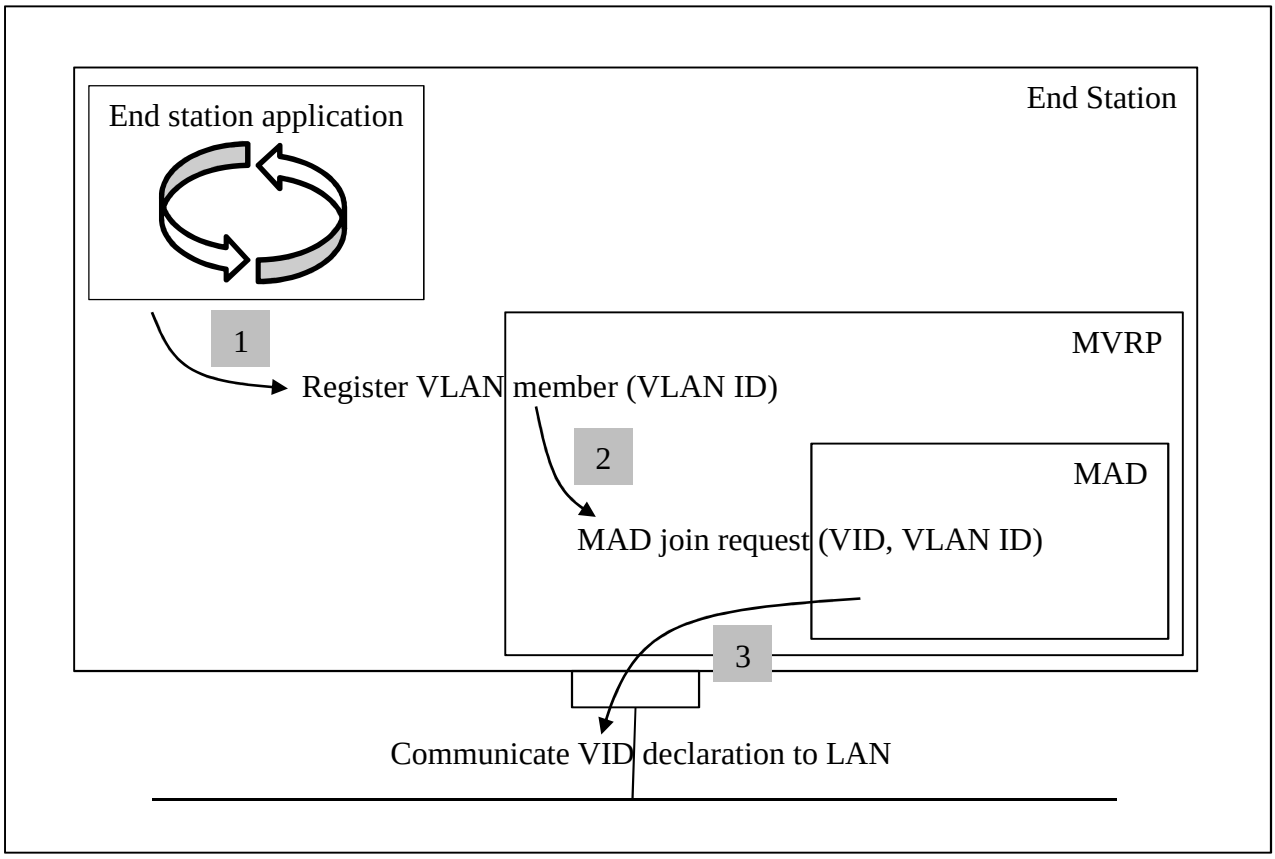

Figure 38: ***Register VLAN member***

A similar process is followed when deregistering a VID. When a *deregister VLAN member* service primitive is issued to MVRP with a VLAN ID, MVRP issues a *MAD leave request* to MAD indicating an attribute type of VID with an attribute value representing the requested VLAN ID. This causes the attribute declaration to be withdrawn, and for this withdrawal to be communicated to the LAN.

MVRP responds to the registration and deregistration of VIDs made by remote bridges and remote end stations. These registrations and deregistrations are indicated to MVRP via MAD's *MAD join indication* and *MAD leave indication* service primitives. An end station or bridge may receive an MRPDU containing a VID attribute declaration. MVRP's MAD component will process this and

register the attribute, and will notify MVRP of the attribute registration via a *MAD join indication*. The MVRP component will configure the bridge or end station such that it will forward frames that are part of the requested VLAN. Shown in Figure 39 is an example of a two-port bridge receiving communication of a VID attribute declaration. A *MAD join indication* is issued for the VID attribute. This results in an appropriate entry being added to the filtering database to associate the port on which the attribute is registered with the VLAN ID. This allows for incoming frames that are tagged as being part of that VLAN to be forwarded out of the port on which the VID attribute is registered.

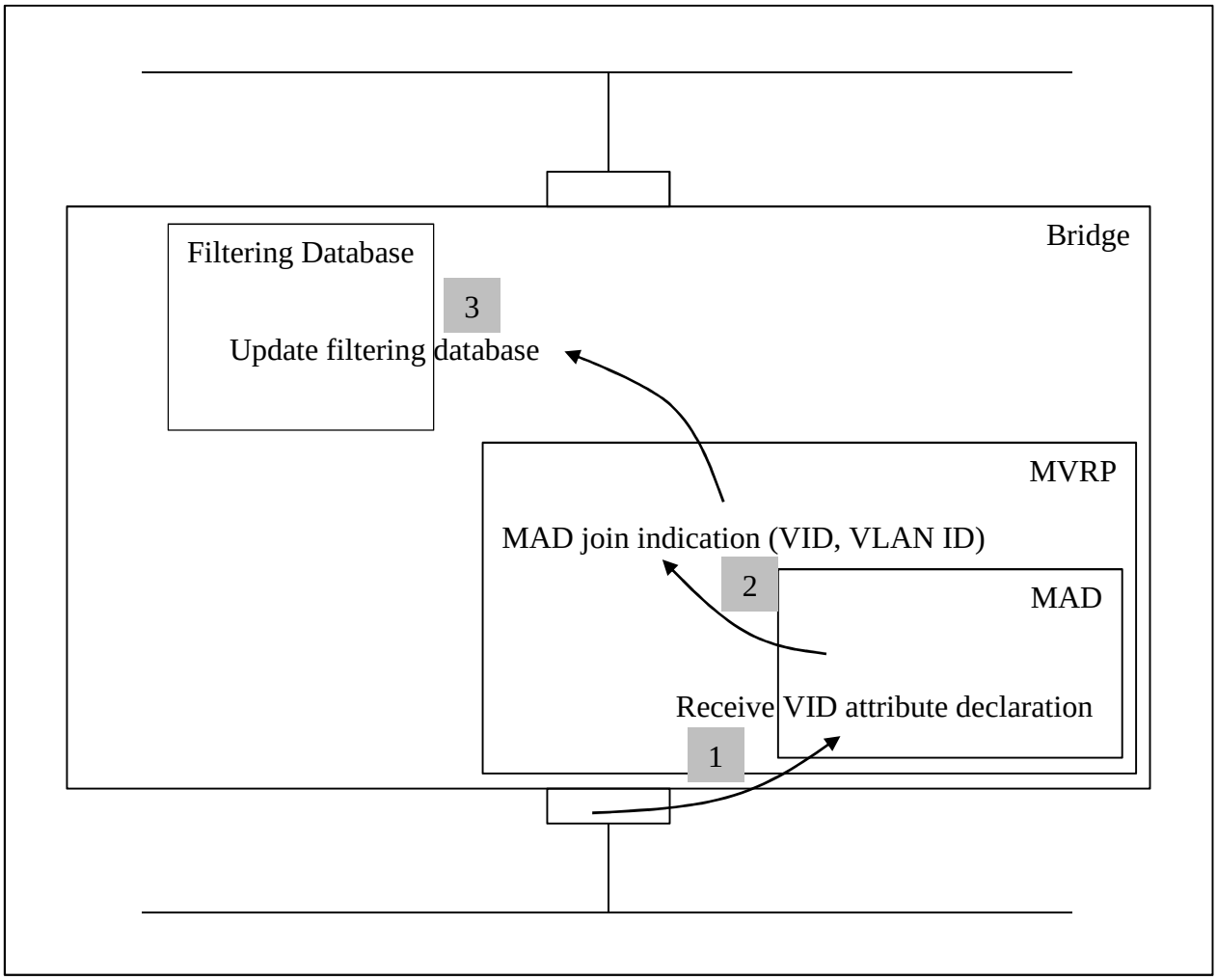

Figure 39: MVRP *MAD join indication*

Similarly, an end station or bridge may receive an MRPDU containing a VID attribute declaration withdrawal. MVRP's MAD component will process this and deregister the VID attribute, and will notify MVRP of the attribute deregistration by signalling it with a *MAD leave indication*. The MVRP component will configure the bridge or end station such that it will filter frames that are part of the requested VLAN for the port. Shown in Figure 40 is an example of a two-port bridge receiving

communication of a VID attribute declaration withdrawal. A *MAD leave indication* is issued for the VID attribute. This results in the appropriate entry being deleted from the filtering database to remove the association between the port on which the attribute is registered, and the VLAN ID. This results in incoming frames that are tagged as being part of that VLAN to be filtered for the port on which the VID attribute was registered.

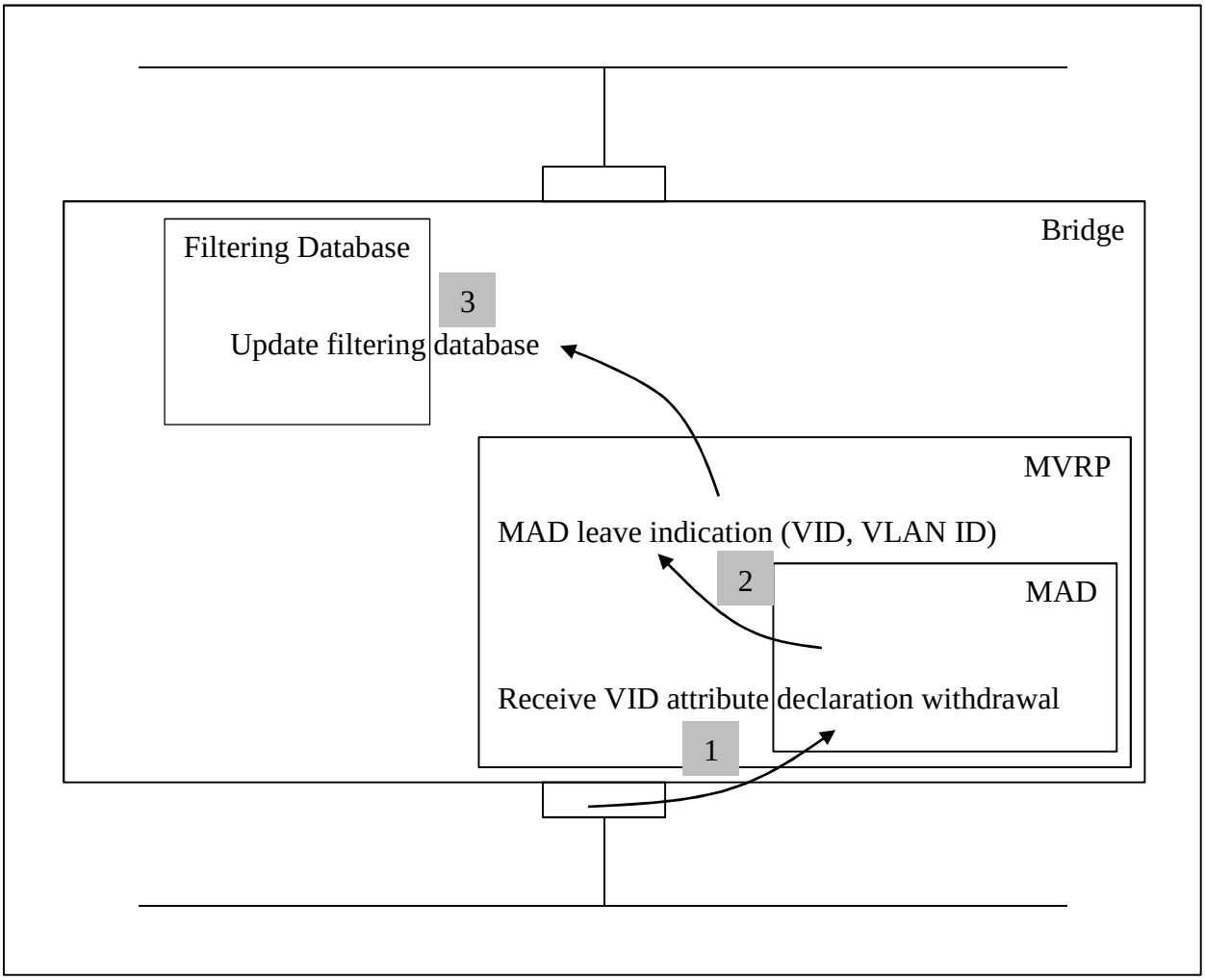

Figure 40: MVRP *MAD leave indication*

3.2.4 Multiple Stream Reservation Protocol

MSRP (which is an MRP application) is a signalling protocol used by end stations to reserve network resources on a bridged LAN. This protocol allows for guaranteed QoS for the transmission of unicast and multicast data streams across a bridged LAN. Transmitted data that is associated with, and

conforms to, a successful stream reservation is not discarded by the network that it is transmitted on. MSRP utilises MVRP, and optionally MMRP.

Talker end stations declare MSRP attributes that are used to define their streams' characteristics. These declarations (and their subsequent propagation through a bridged LAN) allow bridges to allocate the necessary resources to ensure that the streams can traverse the bridged LAN with their requested QoS. Listener end stations declare MSRP attributes that are used to request the reception of the talkers' advertised stream(s). Transmitted data that conforms to a successful stream reservation will not be discarded by any bridge due to congestion on a LAN.

Each bridge on the path from a stream talker to a stream listener will process, possibly combine or alter, and propagate these MSRP attribute declarations. Each stream in an AVB system is uniquely identified by a *stream identifier* (stream ID). Within bridges, talker and listener attributes are associated with each other via a stream ID that is present in each of the attributes. The talker and listener attributes result in changes to the forwarding characteristics of a bridge and the allocation of internal resources when streams are initiated (see Section 4.2.2 "Forwarding and Queuing").

In order for the QoS parameters to be guaranteed, all of the devices in a bridged LAN have to participate in the signalling as well as queuing and forwarding algorithms required of bridges (the queuing and forwarding algorithms are discussed in Section 4.2.2 "Forwarding and Queuing"). MSRP provides a means to interact with higher and lower level networking layers, and provides limited error reporting capabilities.

Shown in Figure 41 is the architecture of MSRP for a two-port bridge and an end station. Each port has associated with it an MSRP participant. Each MSRP participant consists of an MSRP application and a MAD component. For network bridges, MSRP attributes are propagated between the various MSRP participants via a MAP component.

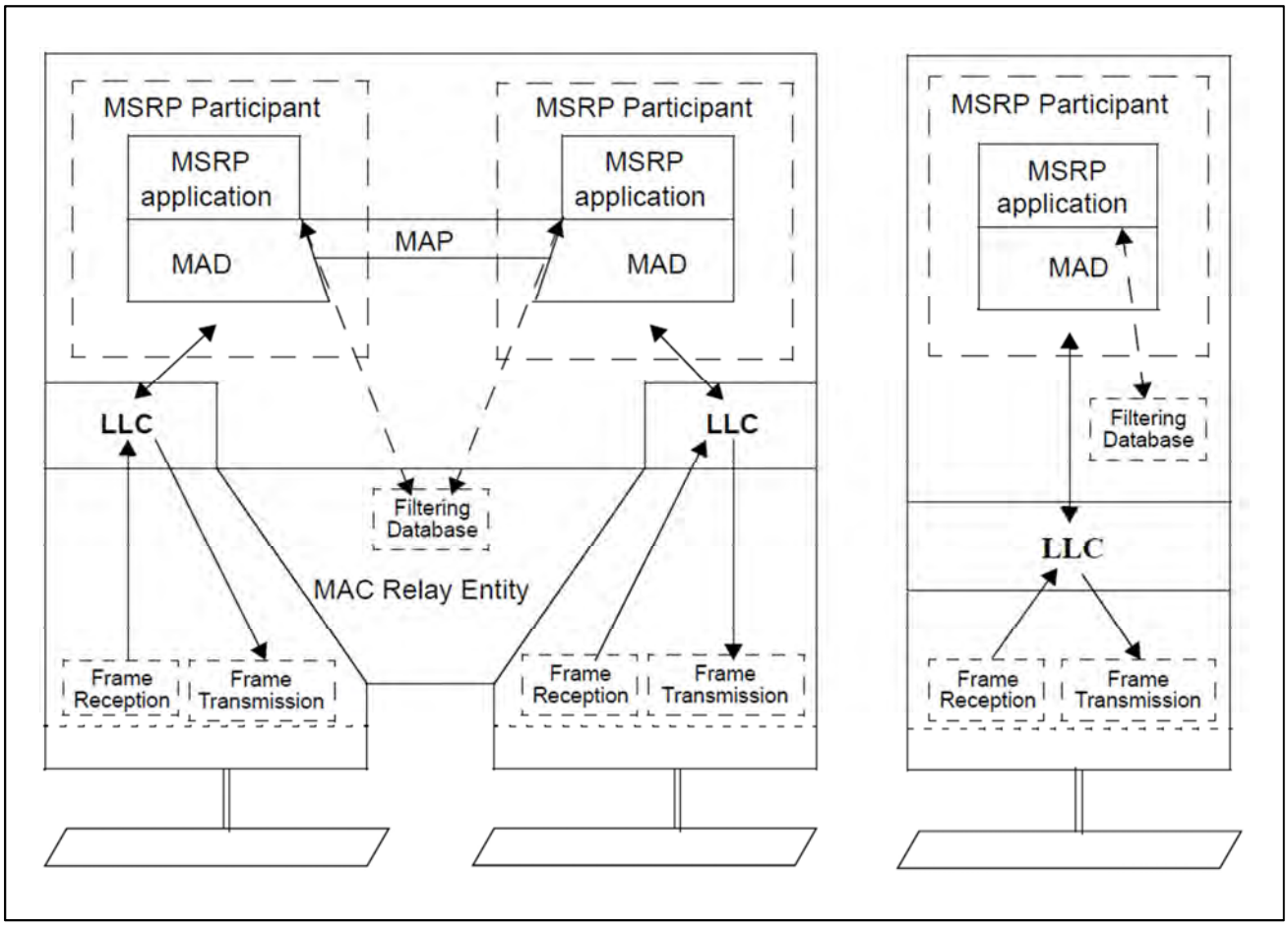

Figure 41: MSRP architecture [67]

3.2.4.1 MSRP Attributes and Service Primitives

Stream talkers and stream listeners use attribute declarations in order to advertise streams and to request attachment to streams respectively. There is no requirement for the order in which stream talkers and stream listeners make their MSRP attribute declarations. The following attribute types are used by MSRP to communicate stream information:

- *Talker advertise*
- *Talker failed*
- *Listener*: the *listener* attribute can in one of three states:
    - *Listener ready*
    - *Listener ready failed*
    - *Listener asking failed*

MSRP provides a set of service primitives which control the declarations of the various attributes defined by MSRP.

The service primitives that are associated with a stream talker are:

- *Register stream request*
- *Deregister stream request*
- *Register attach indication*
- *Deregister attach indication*

The service primitives that are associated with a stream listener are:

- *Register attach request*
- *Deregister attach request*
- *Register stream indication*
- *Deregister stream indication*

These service primitives contain parameters that represent properties of the streams. Section 3.2.4.2 “Talkers Advertising Streams” to Section 3.2.4.3 “Listeners Requesting Attachment to Streams” discuss the high-level way in which the MSRP service primitives and attributes are used in order to configure streams across a bridged LAN. Section 3.2.4.5 “MSRP Attributes and Service Primitives in Detail” discusses these service primitives and attributes in further detail.

3.2.4.2 Talkers Advertising Streams

When a stream talker application has a stream to offer, it will invoke MSRP’s *register stream request* service primitive. MSRP will then request its MAD component to declare a *talker advertise* attribute by invoking its *MAD join request* service primitive. This attribute declaration is sent to the bridged LAN to inform the bridges and listeners about the characteristics of the stream the talker can provide.

Talker attributes gather QoS information along the network paths that they are transmitted on. MSRP examines the availability of network resources (such as bandwidth) for the stream, and as the attribute is propagated through bridges on a bridged LAN, each bridge will update the attribute. When a bridge receives the *talker advertise* attribute declaration, it is registered by the MSRP

participant on the port that receives the attribute declaration. The MSRP MAP function assesses whether there are sufficient resources on each of the outbound ports of the bridge for the stream. If it is found that an outbound port is not able to support a particular stream, MSRP's MAP component will change a *talker advertise* attribute to a *talker failed* attribute before propagating it to the MSRP participant on that port. If a *talker failed* attribute is registered by an MSRP participant on a bridge port, it will be propagated to all other MSRP participants of the same bridge as a *talker failed* attribute.

When a *talker advertise* attribute is registered by an MSRP participant, it indicates that the advertisement did not encounter any resource constraints along the path from the stream talker end station. This indicates to a stream listener that there are sufficient resources on the network to support the stream referenced in the *talker advertise* attribute. If a stream listener requests attachment to this stream, it is guaranteed that it will receive it with the described QoS. When a *talker failed* attribute is registered by an MSRP participant, it indicates that the advertisement encountered resource constraints along the path from the stream talker. This indicates to a stream listener that the stream will not be available to it because of resource constraints somewhere along the network path from the talker end station.

Figure 42 shows an example of a bridged LAN consisting of a talker and three listeners. The talker registers a stream that it has to offer with the bridged LAN. It does so by declaring a *talker advertise* attribute for the stream (D:TA). This *talker advertise* attribute is registered by the MSRP participant associated with port 1 of the bridge (R:TA). The MSRP MAP function is notified of the *talker advertise* attribute registration. The MAP function analyses the availability of resources to determine if each outbound port is able to support the stream. In this example, each port, except port 3, has sufficient resources to support the stream. The MAP function instructs the MSRP participants of port 2 and port 4 to declare a *talker advertise* attribute, and instructs the MSRP participant of port 3 to declare a *talker failed* attribute (D:TF). This results in a *talker advertise* attribute being registered on the listeners connected to port 2 and port 4, and a *talker failed* attribute being registered on the listener connected to port 3 of the bridge (R:TF).

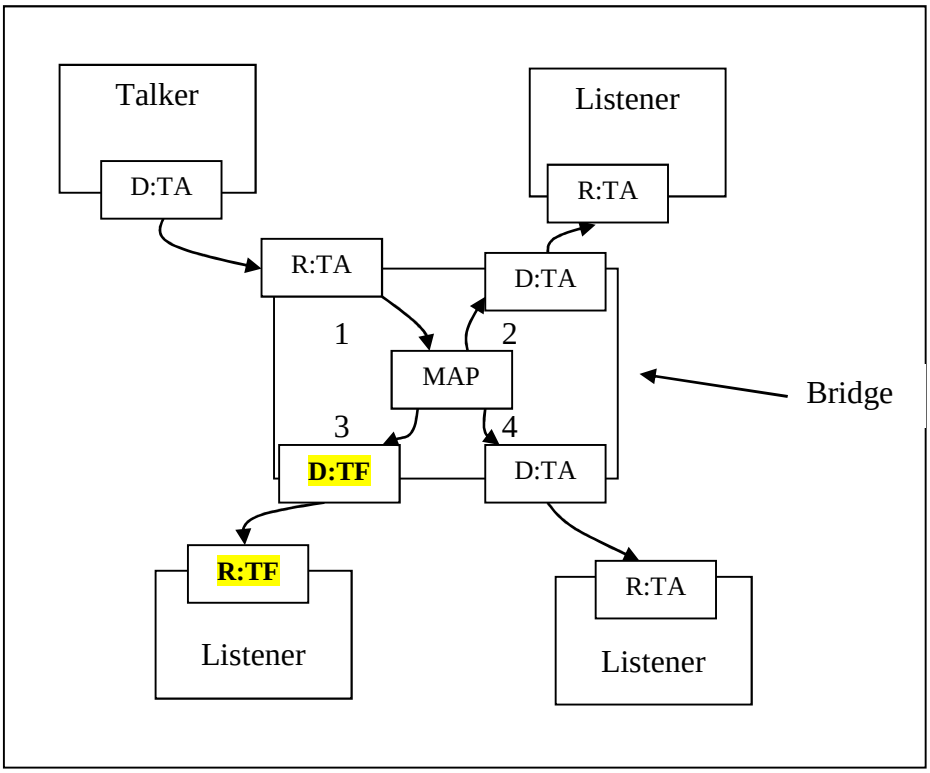

Figure 42: An example of *talker* attribute propagation

By default, *talker* attribute declarations are sent out of all non-blocked ports unless *talker pruning* is enabled. The *talker pruning* option limits the scope of the talker declaration propagation. When *talker pruning* is enabled, before a talker is able to advertise a stream, a listener must first issue a *register MAC address* service primitive to MMRP for the destination MAC address of the talker's stream. When the talker advertises the stream, the *talker advertise* attribute is only forwarded out of the ports that have the stream's destination multicast MAC address associated with it.

When a *talker advertise* or a *talker failed* attribute is registered with a listener's MSRP MAD component, the MSRP application is notified of this via the *MAD join indication* service primitive. The MSRP participant then notifies the listener application of the stream registration via the *register stream indication* service primitive.

When a talker's stream is no longer available, the talker requests MSRP to deregister the stream. The talker does so by invoking MSRP's *deregister stream request* service primitive, which results in MSRP invoking its MAD component's *MAD leave request* to withdraw the *talker advertise* attribute declaration. The *talker* attribute is deregistered from all ports on the bridged LAN. Each MAD component notifies its MSRP application of the attribute withdrawal by invoking the *MAD leave*

request service primitive for the *talker* attribute. The MSRP participant notifies a listener application of the stream deregistration by issuing it with a *deregister stream indication* service primitive.

3.2.4.3 Listeners Requesting Attachment to Streams

When a listener wants to receive a particular stream, it makes a request to receive this stream by invoking MSRP's *register stream attach* service primitive specifying the listener declaration type. MSRP then invokes its MAD component's *MAD join request* service primitive to request it to declare a *listener* attribute. The listener declaration type is carried within the optional value field of a message within an MRPDU. The listener declaration types are *ready*, *ready failed*, and *asking failed*:

- A *listener ready* attribute indicates that one or more listeners are requesting attachment to the referenced stream and there are sufficient resources available along the path(s) back to the stream talker for all listeners to receive the stream.
- A *listener ready failed* attribute indicates that two or more listeners are requesting attachment to the referenced stream and at least one of the listeners have sufficient resources along the path to receive the stream, but one or more other listeners are unable to receive the stream because of network resource allocation problems.
- A *listener asking failed* attribute indicates that one or more listeners are requesting attachment to the referenced stream but none of those listeners are able to receive the stream because of network resource allocation problems.

If a listener has a *talker advertise* attribute registered for a stream that is wishes to receive, it will request attachment to the stream as shown in Figure 43:

1. The listener has a *talker advertise* attribute registered for a stream (R:TA), and the listener is ready to receive the stream.
2. The listener requests membership to the VLAN referred to in the *talker advertise* attribute (see Section 3.2.4.5.1.2 "Data Frame Parameters"). This is performed via the *register VLAN member* service primitive of MVRP (D:VM) (see Section 3.2.3.1.1 "Registering and Deregistering VLANs").
3. The listener declares a *listener ready* attribute for the stream (D:LR).

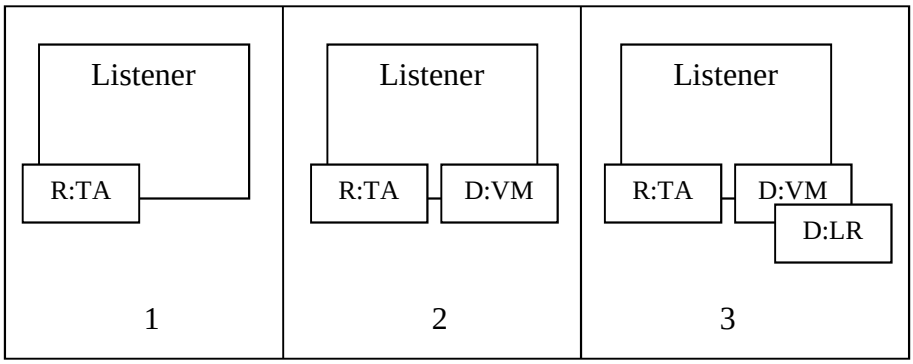

Figure 43: Listener responding to a *talker advertise*

If a listener has a *talker failed* attribute registered for a stream that is wishes to receive, it will respond as shown Figure 44:

1. The listener has a *talker failed* attribute registered for a stream (R:TF), and the listener is ready to receive the stream.
2. The listener declares a *listener asking failed* attribute for the stream (D:LAF).

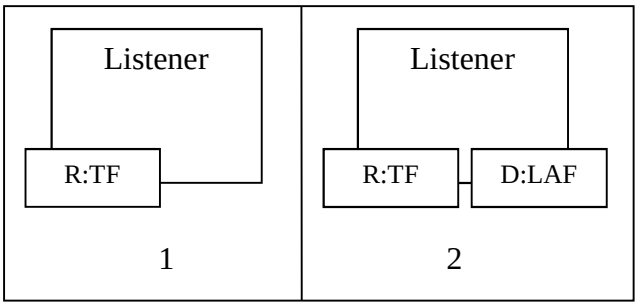

Figure 44: Listener responding to a *talker failed*

A listener may request attachment to a stream before the stream has been advertised by a talker. When this happens, a listener declares a *listener asking failed* attribute for the stream.

A bridge's MSRP MAP function will use the stream ID field of the *talker* and *listener* attributes to associate *listener* attributes with *talker* attributes. This allows *listener* attributes to be forwarded only on the ports on which the *talker* attributes are registered (this is referred to as *listener pruning*). If any bridge along the path from a stream talker to a stream listener does not have sufficient resources available to support a stream, its MSRP MAP function will change a *listener ready* attribute to a *listener asking failed* attribute. If it is found that there is no *talker* attribute declaration to associate with a *listener* attribute declaration, the *listener* attribute will not be propagated. If a bridge receives a *listener ready* attribute declaration and it associates it with a *talker failed* attribute declaration, the

bridge will transform the *listener* attribute declaration into a *listener asking failed* declaration before forwarding it.

If at some point in a bridged LAN a bridge has a *listener ready* on one port, and a *listener asking failed* on another port for the same stream, the MSRP MAP function will merge these two attribute declarations into a single *listener ready failed* attribute declaration before propagating it. This will be forwarded to the stream talker. When a stream talker receives a *listener ready failed*, it knows that one or more listeners want the stream, but not all of them can receive it. The stream is still transmitted by the talker.

Figure 45 shows the same example network as shown in Figure 42. As in Figure 42, the talker has advertised a stream, and this advertisement has been propagated throughout the network. Port 3 of the bridge can however not support the stream due to resource constraints. The listener attached to port 2 and the listener attached to port 4 of the bridge would like to receive the stream being advertised by the talker. Each of the listeners declares a *listener ready* attribute (D:LR). These *listener* attributes are registered on the respective listeners' bridge ports (R:LR). The MSRP MAP function then propagates these attributes as *listener ready* attributes towards the talker. These attributes are merged on port 1 of the bridge as a *listener ready* attribute before being forwarded and registered on the talker.

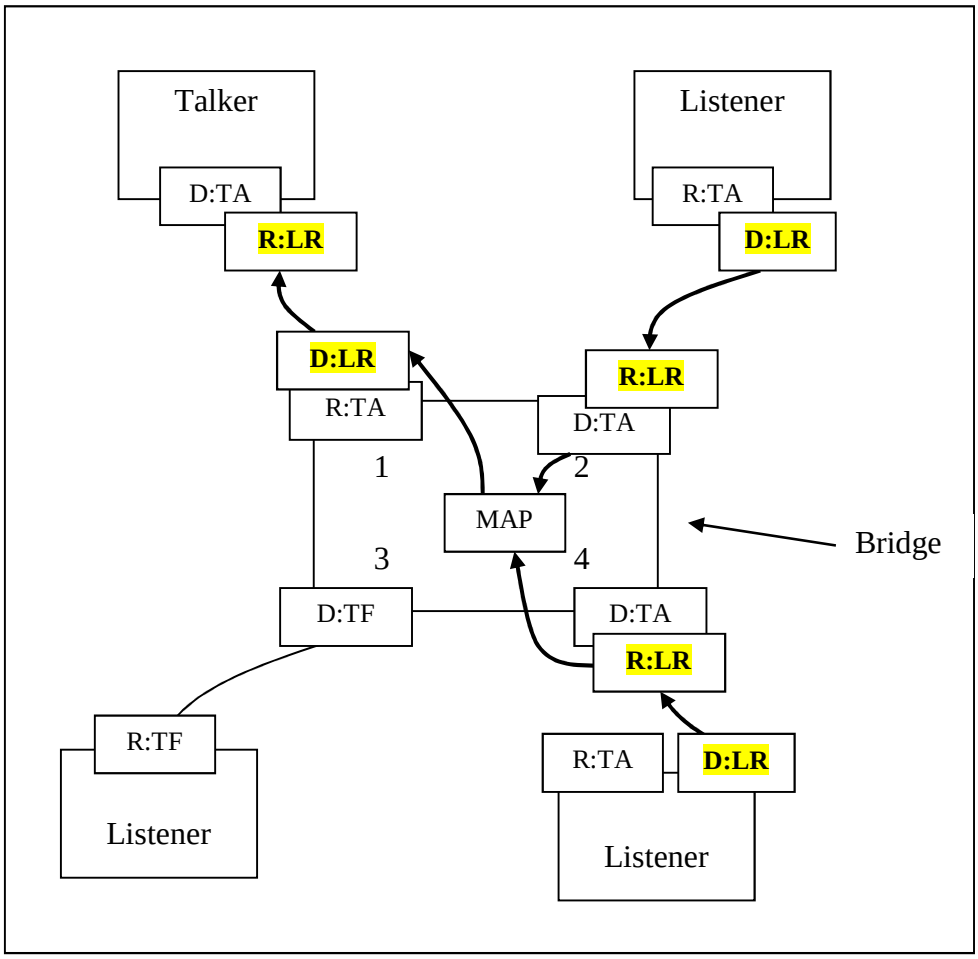

Figure 45: Example *listener* attribute propagation and merging

Figure 46 shows the same example network as shown in Figure 45. However, the listener attached to port 3 of the bridge would also like to receive the stream being advertised by the talker. Due to resource constraints, the bridge is unable to forward the stream out of port 3. As the listener has a *talker failed* attribute registered for the stream, it declares a *listener asking failed* attribute (D:LAF). This *listener asking failed* attribute is registered on port 3 of the bridge (R:LAF), and propagated as a *listener asking failed* attribute. At port 1, the *listener asking failed* is merged with the *listener ready* attribute. The resulting *listener* attribute is a *listener ready failed* attribute (D:LRF). This *listener ready failed* attribute is forwarded to the talker and registered on its port (R:LRF).

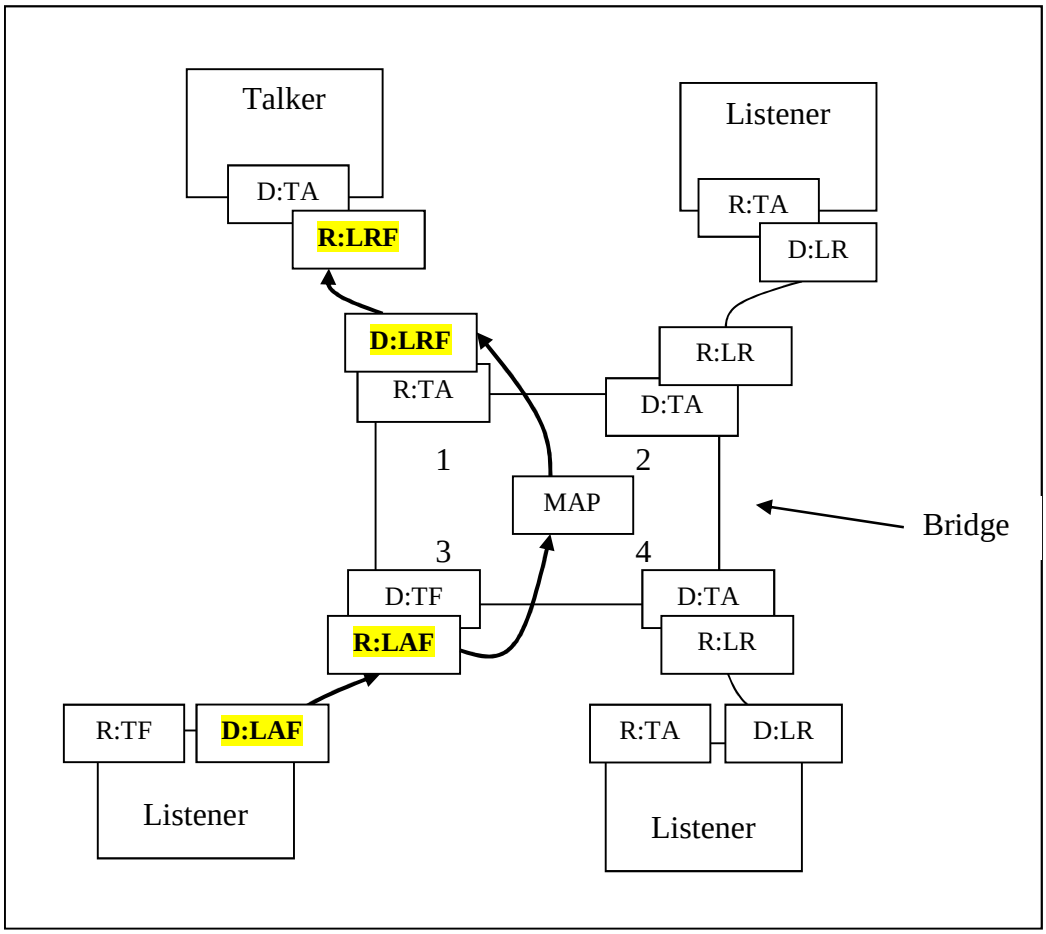

Figure 46: Example *listener* attribute propagation and merging

Table 3 shows how *listener* attributes are propagated from incoming ports:

- If a *listener* attribute for a particular stream is registered on a port of a bridge, and there are no *talker* attributes for that stream on any other ports, the *listener* attribute is not be propagated.
- If a *listener asking failed* attribute for a particular stream is registered on a port of a bridge, it will be propagated as a *listener asking failed* to the other ports of the bridge.
- If a *listener ready* or *listener ready failed* attribute for a particular stream is registered on a port of a bridge, and a *talker failed* attribute for that stream is registered on another port, a *listener asking failed* attribute will be propagated to that port.
- If a *listener ready* or *listener ready failed* attribute for a particular stream is registered on a port of a bridge, and a *talker advertise* attribute for that stream is registered on another port, the *listener* attribute will be propagated as-is to that port.

		Talker		
		(none)	***Advertise***	***Failed***
Listener	**(none)**	(none)	(none)	(none)
	Ready	(none)	*Listener ready*	*Listener asking failed*
	Ready failed	(none)	*Listener ready failed*	*Listener asking failed*
	Asking failed	(none)	*Listener asking failed*	*Listener asking failed*

Table 3: *Listener* attribute propagations

When an MSRP participant registers or declares a *listener* attribute for a particular stream, it merges this attribute with any *listener* attributes that it may already have for that same stream into a single *listener* attribute. The resulting *listener* attribute type is shown in Table 4.

First declaration type	Section declaration type	Resultant declaration type
Ready	None or *ready*	*Ready*
Ready	*Ready failed* or *asking failed*	*Ready failed*
Ready failed	Any	*Ready failed*
Asking failed	*Ready* or *ready failed*	*Ready failed*
Asking failed	None or *asking failed*	*Asking failed*

Table 4: *Listener* attribute merging

When a talker registers a *listener* attribute, it examines the stream ID and declaration type of the merged *listener* attribute:

- If the stream ID matches the stream ID of one of the streams that the talker is able to supply and the declaration type is either *listener ready* or *listener ready failed*, the talker can start the transmission of this stream immediately.
- If the declaration type is *listener asking failed*, the talker stops the transmission of the stream (if it is currently transmitting the stream).

If a talker receives a *MAD leave indication* for a *listener* attribute and if the stream ID matches one of the streams that the talker is transmitting, the talker stops the transmission of the stream (if it is currently transmitting the stream).

Figure 47 shows an example of a talker end station merging *listener* attributes:

1. The talker has registered one *listener* attribute with a declaration type of *ready* and has started the transmission of the requested stream (R:LR).
2. A second *listener* attribute (with the same stream ID as the first one) is registered on the talker with a declaration type of *ready failed* (R:LRF).
3. The second attribute will be merged with the first one and the resulting attribute will have a declaration type of *ready failed* (R:LRF).

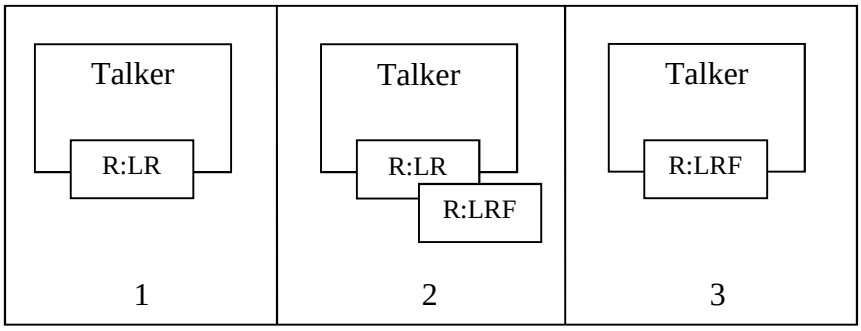

Figure 47: Merging of listener attributes

When a listener no longer wishes to receive a particular stream, it will make a request to stop receiving the stream. It will invoke its MSRP participant's *deregister attach request* service primitive for the stream. This results in MSRP invoking its MAD component's *MAD leave request* for the *listener* attribute to request it to withdraw the attribute.

3.2.4.4 Updating Queuing and Forwarding Information

When incoming *listener* attribute processing has been completed for a port, a bridge's MAP component will update the bridge's filtering database to either filter any frames with the stream's destination MAC address, or forward any frames with the stream's destination MAC address. If a *talker advertise* attribute is matched with a *listener ready* or *listener ready failed* attribute, the filtering database is configured to forward the stream's frames, otherwise it is configured to filter the stream's frames.

When MSRP's MAP component is called upon (when an attribute is registered on a port of a bridge), a change in the streams that are filtered and those that are forwarded can occur. These changes are reflected in the amount of bandwidth that is currently reserved for use by the queues that are

associated with the streams. If a stream is removed (for example, when a talker withdraws its *talker advertise* attribute), MAP decreases the bandwidth reserved for the stream. MAP increases the amount of reserved bandwidth when streams are successfully established.

In order ensure that no stream packets are dropped, if the reserved bandwidth of a port is going to be increased, the bandwidth should be updated before the filtering database is updated as the bridge should not start forwarding a stream's frames before bandwidth has been allocated to it. Conversely, if the reserved bandwidth of a port is going to be decreased, the filtering database should be updated before the reserved bandwidth is updated.

Section 4.2.2 "Forwarding and Queuing" contains further details on how bandwidth is reserved for streams.

3.2.4.5 MSRP Attributes and Service Primitives in Detail

Section 3.2.4.1 "MSRP Attributes and Service Primitives" gave a high-level overview of MSRP's attributes and service primitives. This section describes these service primitives and attributes in more detail.

3.2.4.5.1 *Talker* Attributes

The structure of a *talker advertise* attribute and a *talker failed* attribute is shown in Table 5.

Talker advertise	***Talker failed***
Stream ID: Eight octets	Stream ID: Eight octets
Data frame parameters: Eight octets	Data frame parameters: Eight octets
TSpec: Four octets	TSpec: Four octets
Priority and rank: One octet	Priority and rank: One octet
Accumulated latency: Four octets	Accumulated latency: Four octets
	Failure information: Nine octets

Table 5: Structure of *talker* attributes

The fields shown in Table 5 are defined in Section 3.2.4.5.1.1 “Stream ID” through to Section 3.2.4.5.1.6 “Failure Information”

3.2.4.5.1.1 Stream ID

The 64-bit stream ID field is used to match *talker* attributes with their corresponding *listener* attributes. The stream ID consists of the following sub-components:

- The first 48-bits is the 48-bit MAC address that is associated with the system sourcing the stream to the bridged LAN. The entire range of possible 48-bit addresses is acceptable.
- The last 16-bits is an unsigned integer value that is used to distinguish among multiple streams sourced by the same system.

A stream ID is unique across the entire bridged LAN. These are generated by a system offering streams, or could be generated by a controlling system.

3.2.4.5.1.2 Data Frame Parameters

The data frame parameters component specifies the destination MAC address and VLAN ID that are common to all frames belonging to the data stream. The information contained in it is used by bridges to appropriately configure their *filtering databases*.

Valid MAC addresses are multicast MAC addresses, or local administered addresses. A system that is not VLAN aware uses a VLAN ID of 2 whereas a VLAN aware system may use any valid VLAN ID (1 through 4094).

3.2.4.5.1.3 TSpec

This 32-bit field is the traffic specification that is associated with the stream. It contains two elements:

- *Max frame size*: This is a 16-bit unsigned field that is used to allocate resources and adjust queue selection parameters in order to supply the QoS requirements of the talker. It represents the maximum frame size that the talker will produce for the stream. This value does not include any overhead for the media specific framing. The bridge or talker will determine the media specific

framing overhead on the egress port. This value is added to the max frame size when calculating the amount of bandwidth to reserve.

- *Max interval frames*: This is a 16-bit unsigned field that is used to allocate resources and adjust queue selection parameters in order to supply the QoS requirements of the talker. It represents the maximum number of frames that the talker may transmit in one *class measurement interval*. The class measurement interval is dependent on the traffic class that the stream is part of. Traffic class A has a class measurement interval of 125 µs and traffic class B has a class measurement interval of 250 µs.

Section 4.2.2 “Forwarding and Queuing” discusses how the values carried in these fields are used to reserve bandwidth.

3.2.4.5.1.4 Priority and Rank

The priority and rank field is made up of the following components:

- *Data frame priority*: This is a 3-bit field that is used to convey the priority value that the referenced data streams will be tagged with. The value of this parameter will determine which queue the stream’s frames are placed into on an outgoing bridge port.
- *Rank*: This is a 1-bit field that is used by systems to decide which streams can and cannot be served when the port capacity has been exceeded. If a bridge becomes oversubscribed, the rank value will be used to help determine which stream(s) to drop. A value of zero is considered more important than a value of one. The rank field has a default value of one.
- *Reserved*: This is a 4-bit field that is zeroed on transmission, and is ignored when received.

3.2.4.5.1.5 Accumulated Latency

This is a 32-bit unsigned field and is used to determine the worst-case latency that a stream can encounter in its path from the talker to the given listener. The latency should not increase during the life of the reservation. The accumulated latency is initially set to the maximum per-port per-traffic class latency that a frame may experience through the underlying MAC service plus any amounts specified in the *register stream request* service primitive (see Section 3.2.4.5.2.1 “*Register Stream Request*”). The amount specified by the talker is the amount of latency that a stream’s frames will encounter before being passed to the MAC service.

The value of the accumulated latency field is increased by each bridge as the talker attribute propagates through the network. A listener is able to use this latency information in order to decide whether or not the latency is too large for the acceptable presentation of the stream. The accumulated latency is represented in nanoseconds.

3.2.4.5.1.6 Failure Information

If a bridge transforms a *talker advertise* attribute into a *talker failed* attribute, the bridge will add information that indicates the cause of the failure and the identity of the bridge and port on which the failure occurred. The failure information component is made up of two sub-components:

- *Bridge ID*: The value of this field is the ID of the bridge that transformed the talker attribute from a *talker advertise* attribute to a *talker failed* attribute.
- *Reservation failure code*: The value of this field states the reason why the resource reservation failed. These include values that indicate the unavailability of bandwidth and other bridge resources, as well as the validity of parameters included in the *talker advertise* attribute.

3.2.4.5.2 Talker Service Primitives

Associated with a talker are a number of service primitives. Section 3.2.4.5.2.1 "*Register Stream Request*" to Section 3.2.4.5.2.4 "*Deregister Stream Indication*" discusses these service primitives.

3.2.4.5.2.1 Register Stream Request

A stream talker application entity initiates the advertisement of an available stream via the *register stream request* service primitive of an MSRP participant. Shown below are the parameters of the *register stream request* service primitive:

- Stream ID
- Declaration type
- Data frame parameters
- TSpec
- Data frame priority
- Rank

- Accumulated latency

When an MSRP participant receives a *register stream request*, it issues a *MAD join request* to MAD. The attribute type indicated to MAD has a value indicating the type of *talker* attribute. The declaration type parameter has a value that indicates the type of talker attribute to declare. The value of this parameter would indicate either:

- *Talker advertise*, or
- *Talker failed*

The attribute value indicated to MAD contains the values from the parameters of the *register stream request* service primitive.

3.2.4.5.2.2 Deregister Stream Request

When a talker application entity wishes to notify a network that a stream it previously advertised is no longer available, it invokes MSRP's deregister *stream request* service primitive. This removes the advertisement of the stream from the bridged LAN. The MSRP *deregister stream request* has a stream ID parameter.

When the MSRP participant receives a request to deregister a stream via the *deregister stream request* service primitive, it requests MAD to withdraw the *talker* attribute via the *MAD leave request* service primitive. The attribute type indicated to MAD is set to the declaration type currently associated with the stream ID. The attribute value indicated to MAD will contain the value of the stream ID and the other values that were in the associated *register stream request*.

3.2.4.5.2.3 Register Stream Indication

The *register stream indication* service primitive is used to notify a stream listener application that a stream is being advertised by a stream talker somewhere on the attached network. Shown below are the MSRP *register stream indication* parameters:

- Stream ID
- Declaration type
- Data frame parameters

- TSpec
- Data frame priority
- Rank
- Accumulated latency

When an MSRP participant receives a *MAD join indication* from MAD indicating an attribute type of *talker advertise* or *talker failed*, the MSRP application will issue a *register stream indication* to the stream listener application. The *register stream indication* will carry the values indicated by MAD. The value of the declaration type parameter will indicate the type of attribute that was registered.

3.2.4.5.2.4 Deregister Stream Indication

A *deregister stream indication* service primitive is used to notify a stream listener application that a stream is no longer being advertised by a particular stream talker. The MSRP *deregister stream indication* service primitive has a stream ID parameter.

When an MSRP participant receives a *MAD leave indication* indicating an attribute type of *talker advertise* or *talker failed*, the MSRP application will issue a *deregister stream indication* to the stream listener application.

3.2.4.5.3 *Listener* Attribute

The *listener* attribute consists of a single stream ID field. The 64-bit stream ID is used to match *listener* attribute registrations with their corresponding *talker* attribute registrations.

3.2.4.5.4 Listener Service Primitives

Associated with a listener are a number of service primitives. Section 3.2.4.5.4.1 "*Register Attach Request*" through to Section 3.2.4.5.4.4 "*Deregister Attach Indication*" discusses these service primitives.

3.2.4.5.4.1 Register Attach Request

A stream listener application uses a *register attach request* service primitive to request attachment to a stream. Shown below are the parameters of the MSRP *register attach request* service primitive:

- Stream ID
- Declaration type

When an MSRP participant receives a *register attach request*, it will issue a *MAD join request* to MAD. The attribute type indicated to MAD is a *listener* attribute. The attribute value contains the stream ID, and the declaration type is communicated in the optional value field of the message in the MRPDU.

The declaration type takes on values that indicate the type of *listener* attribute:

- *Asking failed*
- *Ready*
- *Ready failed*

3.2.4.5.4.2 Register Attach Indication

The *register attach indication* notifies a stream talker application that a stream is being requested by one or more stream listeners. Shown below are the parameters of the *register attach indication* service primitive:

- Stream ID
- Declaration type

When an MSRP participant receives a *MAD join indication* from MAD indicating an attribute type of *listener*, the MSRP participant issues a *register attach indication* to the stream talker application. The *register attach indication* contains the values from the *MAD join indication*.

3.2.4.5.4.3 Deregister Attach Request

When a stream listener application entity wishes to request detachment from a stream, it issues a *deregister attach request* to the MSRP participant. It has a stream ID parameter.

When an MSRP participant receives a *deregister attach request*, it will issue a *MAD leave request* to MAD indicating an attribute type of *listener*. The attribute value will contain the stream ID. The declaration type that is currently associated with the stream ID will be communicated via the optional value field of the message in the MRPDU.

3.2.4.5.4.4 Deregister Attach Indication

A *deregister attach indication* service primitive is used to notify the stream talker application that a stream is no longer being requested by any stream listeners. It has a stream ID parameter.

When an MSRP participant receives a *MAD leave indication* indicating an attribute type of *listener*, the MSRP participant will issue a *deregister attach indication* to the stream talker application. The attribute value indicated by the *MAD leave indication* contains the stream ID.

3.2.4.6 Stream Importance

MSRP ranks streams to decide on their importance:

- If two streams have the same rank, the age of the streams is compared.
- If both the rank and the streams' ages are identical, the streams' stream IDs will be compared and the stream with the lower stream ID is considered more important.

Certain streams may be considered more important than others (for example, an emergency phone call may be given a higher rank that background music being played over a public address system). If a bridge is not able to support a stream of higher importance due to insufficient bandwidth being available for the stream, streams of lower importance will be dropped such that the stream of higher importance can be supported.

3.2.4.7 Stream Bandwidth Calculations

The bandwidth requirements of a stream include more than just the amount specified in the *register stream request*. MSRP need to adds the *per frame overhead* associated with the media attached to the port.

The total frame size for a stream on an outbound port is the sum of the maximum frame size and the overhead associated with the media attached to the port. The associated bandwidth (in bits per second) is calculated by multiplying the total frame size and the number of frames transmitted per class measurement interval.

Streams in the *listener ready* or *listener ready failed* state reduce the amount of available bandwidth that is available to other streams. For a stream that has no listeners, or streams that are in the asking failed state, the amount of bandwidth is not decreased.

Section 4.2.2.3.2.2 “Deriving Actual Bandwidth Requirements from the Advertised TSpec” discusses this in further detail.

3.3 Conclusion

Before a device may transmit stream data onto a network with guaranteed QoS, it has to communicate the stream resource requirements to the network in order to ensure that those resources are reserved by the network for the stream. IEEE 1394 and AVB are two networking technologies that provide environments that allow for streams to be transmitted with appropriate QoS. Each of these networking technologies has native mechanisms in place that allow for network resources to be reserved for streaming data.

This chapter described the protocols involved in communicating resource requirements on IEEE 1394 and AVB networks. In developing a common network-neutral method of establishing stream connections across these networks, there is a need to be able to trigger:

- The allocation of bandwidth and an isochronous channel number by a transmitting device on an IEEE 1394 network.
- The advertisement of a stream by a talker on an Ethernet AVB network.
- The reception of a talker’s stream (identified by a particular stream ID) by a listener on an AVB network.

Chapter 4 Determinism

The maximum transmission delay for real time stream data across a network must be low and deterministic. A streaming application should transmit with confidence knowing that its data will be given the appropriate opportunities to be transmitted onto a network and that this data will be delivered timeously. Realtime streams require a constant amount of network resources and these resources need to be guaranteed.

In order to develop a method of transferring audio data from one networking technology to another, there is a need to understand the fundamental nature of the transmission mechanisms of the networking technologies involved. Chapter 3 detailed the mechanisms used for communicating resource requirements within IEEE 1394 and AVB networks. This chapter details the native network mechanisms that are in place to guarantee the network resources requested by the stations attached to these networks.

4.1 Determinism for IEEE 1394

Once an IEEE 1394 node has obtained its required stream resources from the *isochronous resource manager* (IRM), its stream packet transmission has to happen in a structured fashion. When a *bus reset* occurs, one of the IEEE 1394 nodes on the bus is identified as the *root node* of the tree structure formed by the nodes. Each port on a node is identified as either a *parent* port or a *child* port. A parent port is closer to the root node than a child port. Before there can be data transmission, *bus arbitration* has to take place. Bus arbitration is a process that determines which node is granted access to the bus.

4.1.1 Bus Arbitration

Bus arbitration begins when a period of bus idle time (known as a *gap*) is recognised. This idle time signals the end of a transmission and varies between isochronous and asynchronous transactions. The *isochronous gap* is the period of time that must be observed prior to arbitration for the next isochronous transaction. This gap is approximately 0.04µs. The *subaction gap* is the period of time

that must be observed prior to arbitration for the next asynchronous transaction. This value may be tuned so that arbitration can begin as early as possible without interfering with the normal completion of a subaction and its subsequent acknowledgment. By default, the subaction gap is approximately 10µs.

Any node wishing to access the bus has to arbitrate for bus access, and has to be granted ownership of the bus by the root node before it may transmit. When a node wants bus ownership, it signals a request towards the root node. Any node on the path to the root node that detects the request forwards it towards the root node, unless it is already signalling a request for itself or for another node. Once the root node receives a request on one of its ports, it will grant ownership of the bus to the requesting node. If two nodes' requests reach the root at the same time, the node attached to the port with the lowest numeric port number is granted access to the bus.

Figure 48 shows an example bus of connected nodes with two nodes (node A and node E) trying to gain access to the bus. Both of these nodes send a request for bus ownership to their parent node. Node B detects node A's request and forwards it to its parent. Node E's request is sent directly to the root node (its parent). The root node (node D) receives Node E's request, and thus grants it access to the bus. Node D (the root node) sends a *data prefix* signal out of port 1 to notify all nodes downstream from the root that it has granted ownership of the bus to a node, and that a packet can be expected. When node E receives notification of bus ownership, it removes its request signal for bus ownership and begins packet transmission. Node C forwards the *data prefix* to its children. When node B detects this, node B stops signalling for bus ownership and forwards the *data prefix* signal to node A which causes node A to remove its request for bus ownership. The reception of the *data prefix* signal indicates to node A that another node has won ownership of the bus.

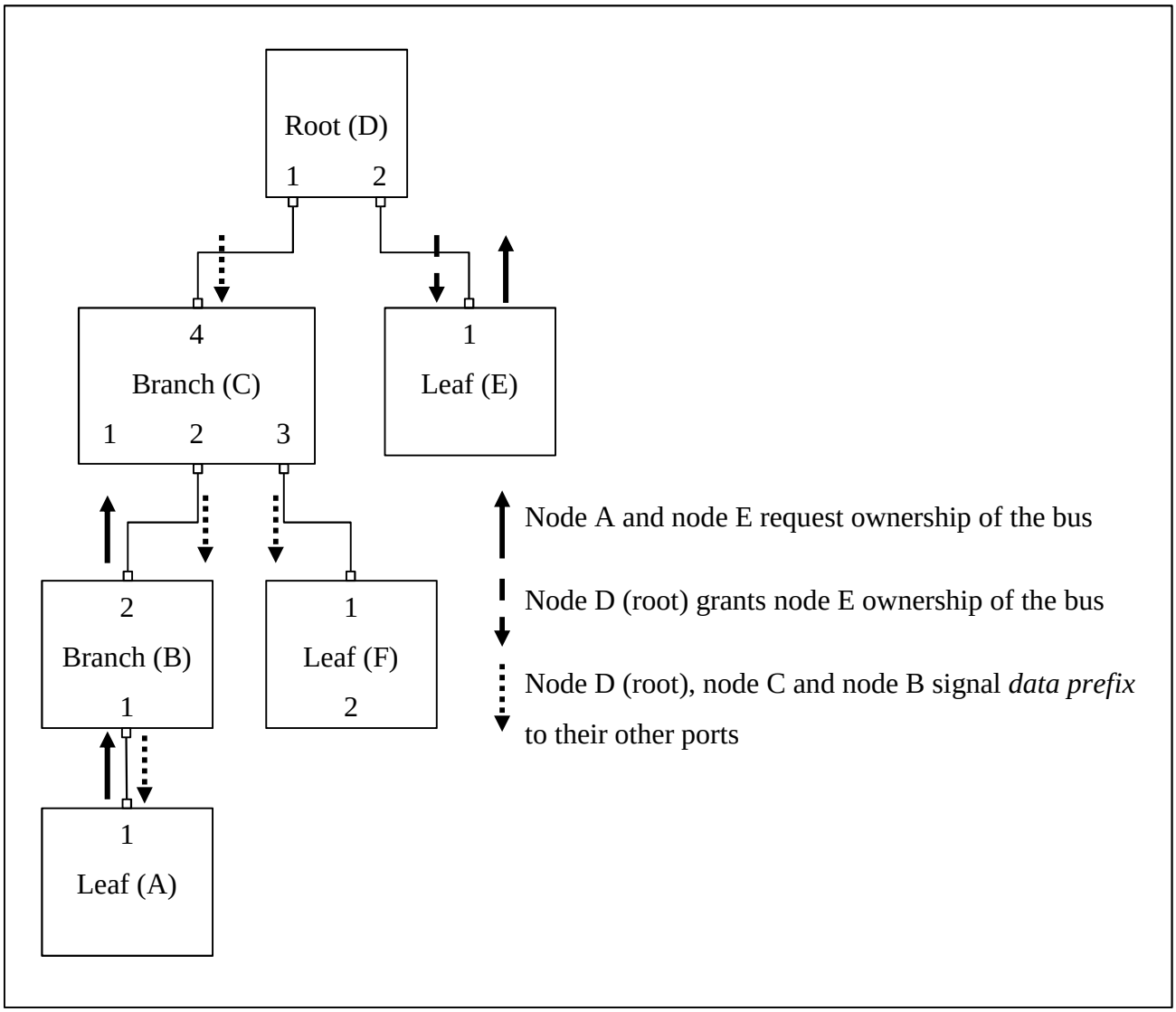

Figure 48: Two IEEE 1394 nodes requesting ownership of an IEEE 1394 bus

4.1.1.1 Isochronous Arbitration

When a node would like to transmit an isochronous packet, it uses the *isochronous arbitration service*. The elected *cycle master* node (which is the same node as the root node) broadcasts a cycle start packet onto the IEEE 1394 bus every 125µs (8000 times per second). Isochronous transactions begin immediately after a cycle start packet has been broadcast onto the bus. Isochronous arbitration will begin when all of the nodes that would like to access the bus observe a bus idle time of approximately 0.04µs. Once arbitration is complete, and a node has been granted access to the bus, the winning node performs its isochronous transaction. Once this transaction is complete, the bus returns to an idle state. Once the bus has been idle for a further 0.04µs, the nodes wishing to perform an isochronous transaction again begin the arbitration process. Any node that would like to perform

an asynchronous transaction is prevented from arbitrating for bus ownership because the longer subaction gap will not occur until all isochronous transactions have completed.

Each isochronous channel requires a certain amount of bandwidth which has been obtained from the IRM. A bandwidth allocation is a portion of each 125µs interval. This ensures that bandwidth is reserved for transmitting devices and not oversubscribed. For each 125µs cycle, once all of the nodes that need to perform an isochronous transaction have done so, the remaining time within the cycle is used for asynchronous packet transmission. If there are no pending asynchronous transactions, the rest of the 125µs cycle goes unused.

4.1.1.2 Cycle Start and Priority Arbitration

The cycle master node broadcasts a cycle start packet onto an IEEE 1394 bus every 125µs. The transmission of a cycle start packet takes priority over other pending asynchronous transactions, thus permitting the start of isochronous transactions. As the cycle master node is the root node, it will always win arbitration at the first subaction gap following the cycle synchronisation event.

4.2 Determinism for Ethernet AVB

The priority information contained in VLAN tagged frames allows for bridges to apply specific frame-forwarding to the frames that form part of streams, and for traffic-shaping to happen at bridges and AVB end-points. End-point devices transmit frames onto a network at a regular time interval. Transmission characteristics are defined by the traffic class that the stream is a part of, and the various QoS parameters that were used when the stream was approved by the network. For example, when a stream is initialised, it may be agreed that the stream talker will transmit frames at a rate of 4.7Mb/s with a frame rate of 8000 packets per second. These 8000 packets must then be transmitted evenly within each second (i.e., one packet transmission every 125µs).

Within a bridge, audio and video frames are forwarded as per usual. They will, however, also be subject to specific traffic manipulation rules. These rules are based on the specific traffic class that the frames belong to and the allocated bandwidth for the traffic class for the egress port of a bridge. The frames that belong to a specific traffic class cannot go beyond the allocated resources.

An Ethernet AVB network also allows for the transmission of non-AVB data (such as web, e-mail and command and control data). This kind of traffic does not have any reserved QoS and frames belonging to this category of traffic may experience frame loss if network resources are constrained.

4.2.1 VLAN Tagged Ethernet Frames

AVB frames are VLAN tagged. In Ethernet frames, a VLAN tag is inserted between the frame header's *source address* field and *type/length* field. Figure 49 shows a VLAN tagged Ethernet frame header which consists of the following fields:

- *Destination address*: this field either contains the unicast MAC address of the station to which this frame is destined, a multicast MAC address if the frame is destined to multiple stations, or a broadcast MAC address if the frame is destined to all stations.
- *Source address*: this field contains the MAC address of the station that sent the frame onto the network.
- *Type/length*: if this field's value is less than or equal to 1500, this field's value represents the length of the data that is carried in the Ethernet packet. If this field's value is greater than or equal to 1536, this field's value identifies the high-level protocol that is being carried in the frame's data field.
- *Tag protocol ID*: This two octet field carries the value 0x8100 which is used to identify the fact that the Ethernet frame is VLAN tagged. This field is in the same position as the *type/length* field of an untagged Ethernet header.
- *Priority code point (PCP)*: This 3-bit field is used to carry priority information. The value of this field can be used to give different handling to frames of different priorities.
- *Canonical format indicator (CFI)*: This 1-bit field is used to specify whether the MAC address is in canonical format or not. This field is always set to zero (canonical format) in bridged Ethernet networks.
- *VLAN ID*: The value of this 12-bit field is used to carry the value identifying the VLAN to which the frame belongs (see Section 3.2.3 "Multiple VLAN Registration Protocol").

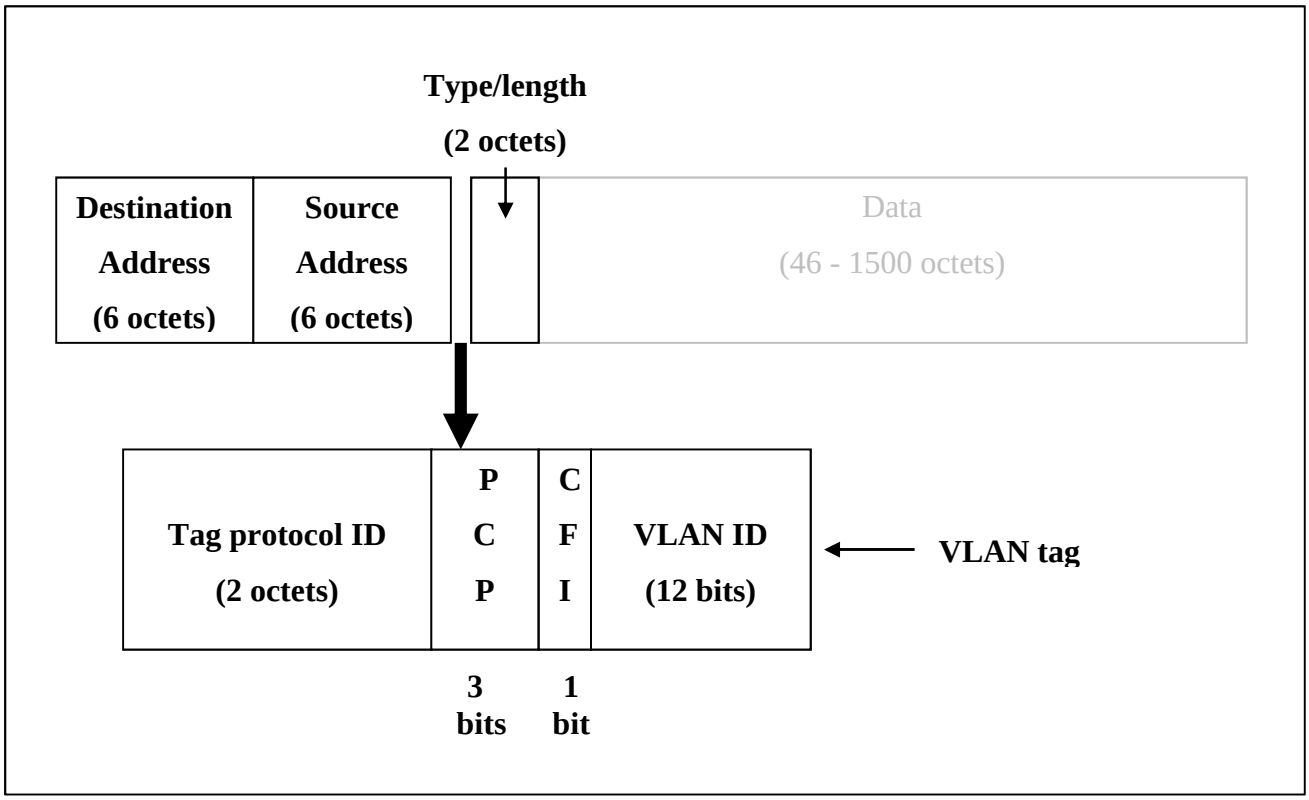

Figure 49: VLAN tag

4.2.2 Forwarding and Queuing

The forwarding and queuing procedures that are defined in IEEE 802.1Qav [68] allow end stations and bridges to provide performance guarantees for the transmission of time and loss sensitive stream data across a bridged LAN. Stream transmission latency has to be bounded and buffering within receiving stations should be minimized for better performance.

VLAN tagged frames, via the value of the PCP field, are used to place stream frames into queues that support stream data whilst non-stream data is placed into other queues. This allows for both stream data and non-stream data to be bridged across a bridged LAN. A number of mechanisms have been put in place to address these performance requirements:

- There is a mechanism to detect the boundary between a set of bridges that support the stream reservation protocol (see section 3.2.1 “Multiple Stream Reservation Protocol”) and the surrounding bridges that do not.

- A set of parameters that together represent the availability of bandwidth for each port. More specifically, these parameters are able to represent the maximum bandwidth available to a given outbound queue and the actual bandwidth reserved for a given outbound queue.
- *Credit-based shaper transmission selection algorithm*: This algorithm is used to shape the transmission of stream-based traffic in accordance with the bandwidth that has been reserved on a given outbound queue.
- A mechanism for mapping the priorities of received frames to the *traffic classes* available on the transmission ports of a bridge.

4.2.2.1 Traffic Classes

A bridge's outbound ports may have associated with them a number of queues for frames waiting to be transmitted. A bridge may implement a number of traffic classes. A traffic class has a one-to-one association with a specific queue on an outbound port of a bridge. Traffic classes are numbered from zero through to N-1 where N is the number of outbound queues that are associated with a given bridge's port. The value of N is between one and eight inclusive. Traffic class zero is used for non-expedited traffic, and any non-zero traffic class is used for expedited traffic.

Figure 50 shows a transmission port of a bridge and the queues that are associated with it. Also shown are the traffic classes that are associated with each queue. In this instance, the transmission port has four queues associated with it, and hence traffic classes numbered from zero through to three are associated with those queues.

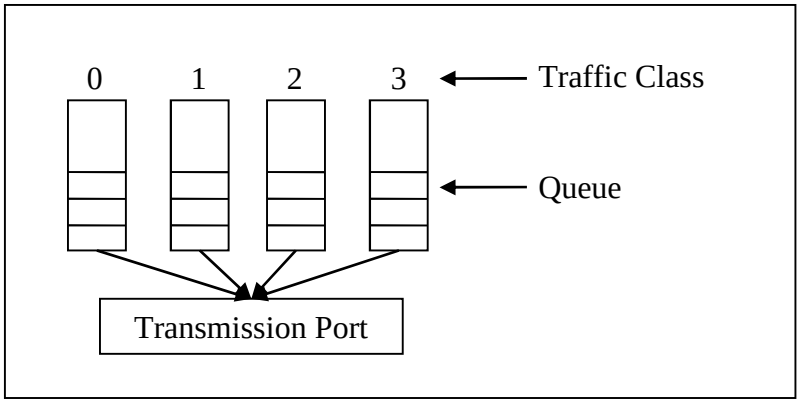

Figure 50: Association of traffic classes to outbound port queues

There is a fixed mapping between the priority associated with each frame and the traffic class that that frame belongs to. This mapping is dependent on the number of traffic classes that a port implements. Table 6 shows the recommended frame priority to traffic class mappings for a bridge that does not support the credit-based shaper algorithm (see Section 4.2.2.3.2 “Credit-Based Shaper Transmission Selection Algorithm” on page 93).

		Number of available traffic classes							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Priority	**0**	0	0	0	0	0	1	1	1
	1	0	0	0	0	0	0	0	0
	2	0	0	0	1	1	2	2	2
	3	0	0	0	1	1	2	3	3
	4	0	1	1	2	2	3	4	4
	5	0	1	1	2	2	3	4	5
	6	0	1	2	3	3	4	5	6
	7	0	1	2	3	4	5	6	7

Table 6: Recommended priority to traffic class mappings [68]

Figure 51 shows an example transmission port of a bridge. A VLAN tagged frame arrives at the bridge (via another port) and is forwarded to the transmission port of the bridge. At the transmission port, the priority value contained within the frame is read, and the corresponding traffic class is obtained from the mapping table. As this transmission port implements four traffic classes, priority four is mapped to traffic class two, and is placed into the queue associated with traffic class two.

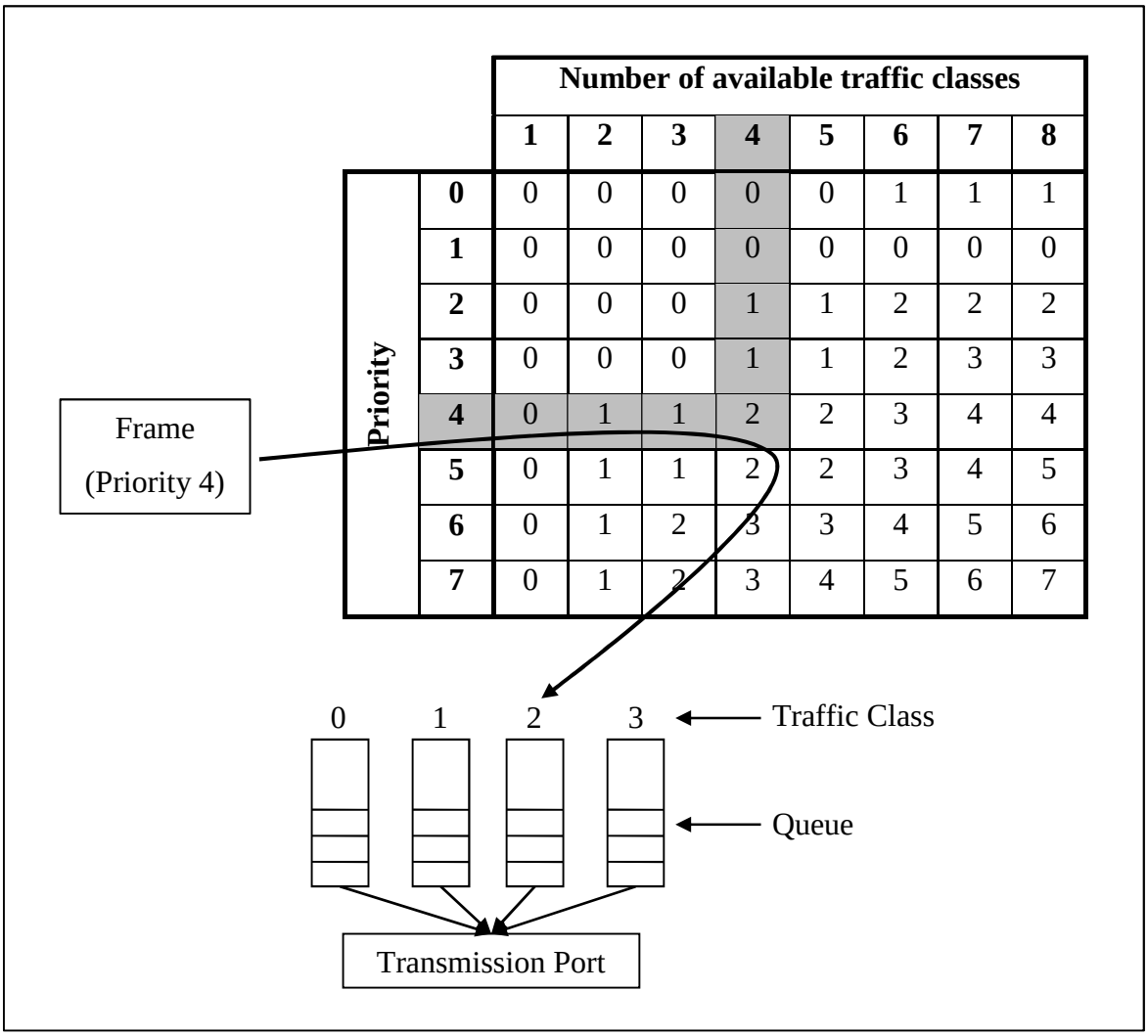

Figure 51: Frame priority to traffic class mapping example

4.2.2.2 Stream Reservation Traffic Classes

A *stream reservation class* (SR class) is a traffic class that allows bandwidth to be reserved for its stream data. SR classes are denoted with successive letters of the alphabet starting with the letter A and continuing up until the letter G (there are seven possible SR classes). Each SR class has associated with it a priority value that is used to map frames to the appropriate traffic class. By default, SR classes are numbered using the highest traffic class numbers that are associated with a bridge port's queues. If, for example, a bridge port implements four traffic classes and two of those traffic classes are SR classes, the SR classes will use traffic class three and traffic class two.

In a bridge that supports the forwarding and queuing of time-sensitive stream data, the default mapping of priority values to traffic class values meets the following constraints:

- The priority values that are used to support SR classes are mapped to traffic class values that support the credit-based shaper algorithm as the transmission selection algorithm.
- Any traffic classes that support the credit-based shaper algorithm have a higher priority than those traffic classes that support the strict priority (or any other) transmission selection algorithm (see Section 4.2.2.3.1 “Strict Priority Transmission Selection Algorithm” on page 93).
- At least one traffic class should support the credit-based shaper algorithm, and at least one traffic class should support the strict priority transmission selection algorithm. The class that supports the strict priority transmission selection algorithm allows for data that is not part of a reservation to be transmitted (e.g., best-effort traffic).

Table 7 shows the recommended default priority to traffic class mappings for SR class A and SR class B. By default, frames that form part of SR class A streams use priority value three, and frames that form part of SR class B streams use priority two. The traffic classes that are shaded are the traffic classes that use the credit-based shaper algorithm (SR classes), and the un-shaded traffic classes are those that use the strict priority transmission selection algorithm.

		Number of available traffic classes						
		2	**3**	**4**	**5**	**6**	**7**	**8**
Priority	**0**	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0
	2	1	1	2	3	4	5	6
	3	1	2	3	4	5	6	7
	4	0	0	1	1	1	1	2
	5	0	0	1	1	1	2	3
	6	0	0	1	2	2	3	4
	7	0	0	1	2	3	4	5

Table 7: Recommended priority to traffic class mappings for SR class A and SR class B

Table 8 shows the recommended default priority to traffic class mappings for a system using SR class B (using priority 2) only.

		Number of available traffic classes						
		2	**3**	**4**	**5**	**6**	**7**	**8**
Priority	**0**	0	0	0	0	0	1	1
	1	0	0	0	0	0	0	0
	2	1	2	3	4	5	6	7
	3	0	0	0	1	1	2	2
	4	0	1	1	2	2	3	3
	5	0	1	1	2	2	3	4
	6	0	1	2	3	3	4	5
	7	0	1	2	3	4	5	6

Table 8: Recommended priority to traffic class mappings for SR class B only

4.2.2.3 Transmission Selection Algorithms

For each port, frames are selected for transmission on the basis of the traffic classes that the port supports and the operation of the *transmission selection algorithm* that is associated with each queue on that port. For a given port and queue, a frame in that queue is selected for transmission from that queue if, and only if:

- The transmission selection algorithm supported by the queue determines that there is a frame available for transmission.
- For each queue that has a numerically higher traffic class value, the transmission selection algorithm associated with that queue determines that there is no frame available for transmission.

This process is illustrated in Figure 52. It shows two bridge ports that each have four queues (hence the traffic classes associated with each of these queues are numbered from zero through to three). Each queue has a transmission selection algorithm associated with it used to determine if a frame is ready for transmission out of its associated queue. The “Transmission Selection” section selects a frame for transmission out of the port.

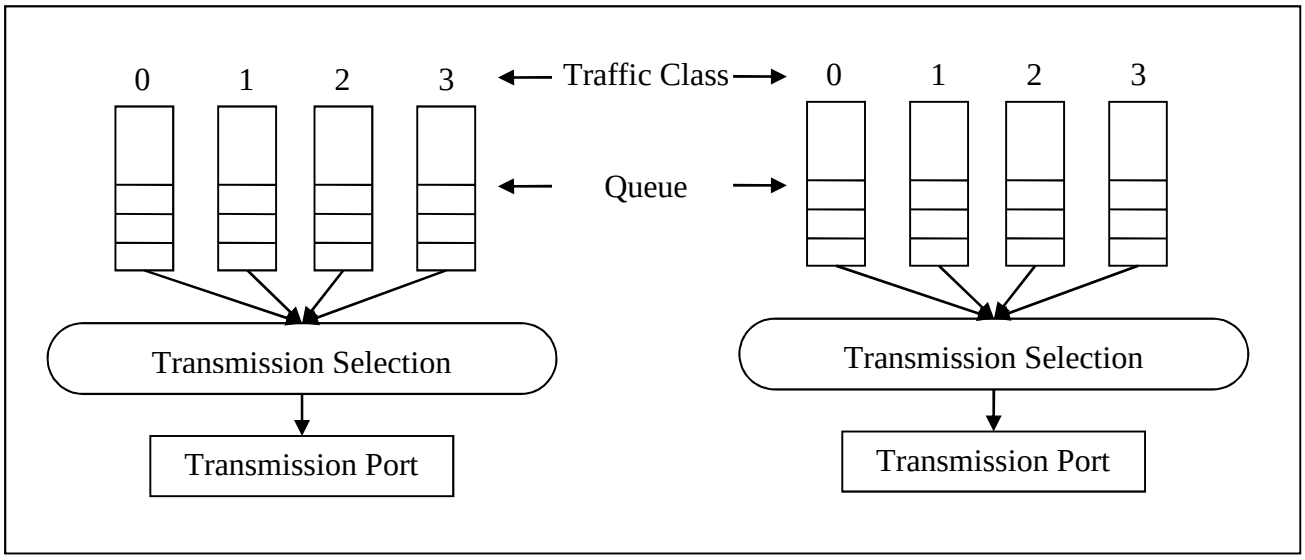

Figure 52: Bridge port transmission selection

IEEE 802.1Qav defines two transmission selection algorithms: the *strict priority transmission selection algorithm*, and the *credit-based shaper algorithm*.

4.2.2.3.1 Strict Priority Transmission Selection Algorithm

The strict priority transmission selection algorithm determines that there is a frame available for transmission from its associated queue if there is at least one frame in the queue.

4.2.2.3.2 Credit-Based Shaper Transmission Selection Algorithm

The credit-based shaper algorithm is used to shape the transmission of stream-based traffic in accordance with the bandwidth that has been reserved on an associated outbound queue. Each queue using this algorithm has a *credit* parameter that represents the transmission *credit* that is available to the queue. The value of this parameter is represented in bits per second. The algorithm determines that a frame is ready for transmission from its associated queue if the following conditions are true:

- The associated queue contains at least one frame.
- The amount of *credit* associated with the queue is zero or positive.

The following are some of the other parameters that are associated with each queue that supports the credit-based shaper algorithm:

- *Port transmit rate*: The value of this parameter represents the transmission rate of the port. The value of this parameter is represented in bits per second.
- *Idle slope*: The value of the *idle slope* parameter represents the actual amount of bandwidth that is reserved for use by the queue (in bits per second), i.e., the maximum portion of the *port transmit rate* that is available to the associated queue. The value of the *idle slope* parameter represents the rate of change of the *credit* parameter when a frame is *not* being transmitted out of the associated queue. The amount of reserved bandwidth is determined via MSRP. If MSRP is not available, the amount of reserved bandwidth can be set by management means.
- *Send slope*: The value of this parameter represents the rate of change of the *credit* parameter when a frame is being transmitted from the associated queue. The value of this parameter is represented in bits per second. The value of the *send slope* parameter is equal to the value of the *idle slope* parameter less the value of the *port transmit rate* parameter.
- *Delta bandwidth*: The value of this parameter represents the bandwidth that can be reserved for use by the queue associated with the traffic class in addition to the *delta bandwidth* values associated with higher priority queues. The value of this parameter is represented as a percentage of the *port transmit rate*. For a given traffic class, the total bandwidth that can be reserved is the sum of the *delta bandwidth* values for the traffic class and all higher traffic classes, minus any bandwidth reserved by higher traffic classes that support the credit-based shaper algorithm. By default, the recommended value of *delta bandwidth* for the highest numbered traffic class is 75% of the port's available bandwidth. For any lower numbered traffic classes, the recommended additional amount of reservable bandwidth (i.e., *delta bandwidth*) is 0%. If the reserved bandwidth for a given queue is less than the maximum value, then any lower-priority queue that supports the credit-based shaper algorithm may make use of the reservable bandwidth that has not been used by the higher priority queue.

  Assume that there are four queues on an outbound port of a bridge and two of these queues support the credit-based shaper algorithm. These two queues are queues 3 and 2 and they support SR class A and B respectively. The other two queues support the strict transmission selection algorithm. This is illustrated in Figure 53.

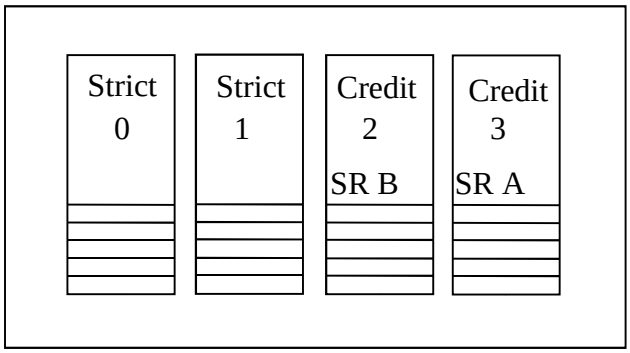

Figure 53: Example outbound queues

Suppose the following:

- *Delta bandwidth* for queue three (SR class A) is 20%
- *Delta bandwidth* for queue two (SR class B) is 30%
- The amount of reserved bandwidth for queue three (SR class A) is 10%

then the maximum amount of bandwidth that can be reserved for SR class B is 40% (30% + 20% - 10%)

Suppose the following:

- *Delta bandwidth* for queue three (SR class A) is 20%
- *Delta bandwidth* for queue two (SR class B) is 30%
- The amount of reserved bandwidth for queue three (SR class A) is 20%

then the maximum amount of bandwidth that can be reserved for SR class B is 30% (30% + 20% - 20%)

If at any time there are no frames in a queue, no frame is being transmitted out of the queue, and the value of the *credit* parameter is positive, the value of the *credit* parameter is set to a value of zero. If a queue uses less bandwidth that what is reserved for it, the unused bandwidth can be used by other traffic classes.

In order for the operation of the credit-based shaper algorithm to operate as designed, each traffic class that supports the algorithm has to be numerically higher than any traffic class that does not support the algorithm.

4.2.2.3.2.1 Credit-shaper Algorithm by Example

Assume that a bridge port has a *port transmit rate* of 100 Mb/s. If, for example, MSRP requests to reserve 25 Mb/s worth of bandwidth for SR class A streams, then the queue's *idle slope* parameter will have a value of 25 Mb/s and the *send slope* parameter will have a value of -75 Mb/s (this is calculated by subtracting the *port transmit rate* from the *idle slope* (25 Mb/s – 100 Mb/s)).

When a frame is being transmitted out of a queue supporting the credit-shaper algorithm, the *credit* value associated with that queue is decreased at a rate equal to the queue's *send slope* parameter. When the frame transmission ends, the value of the queue's *credit* parameter increases at a value equal to the value of the *idle slope* parameter.

Error! Reference source not found. shows a graph showing the value of a *credit* parameter over time.

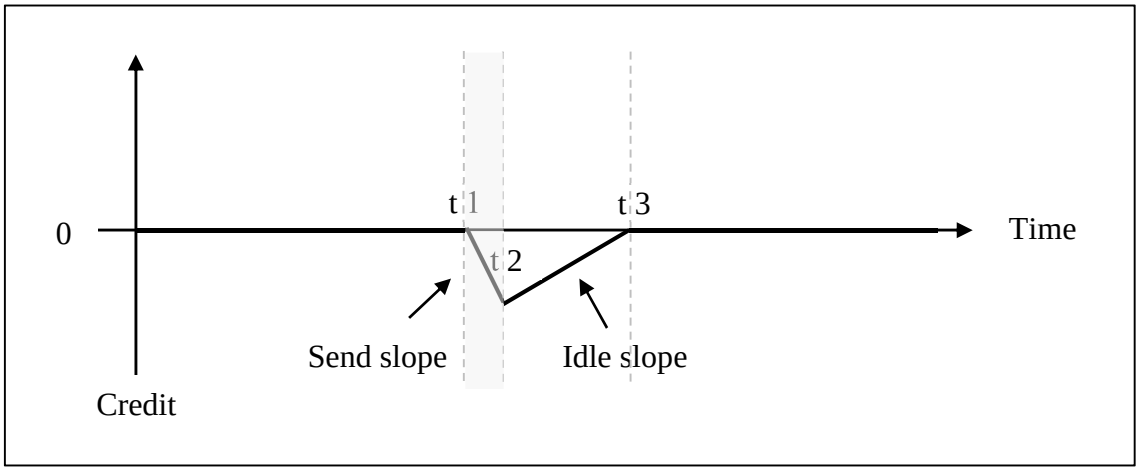

Figure 54: Credit-shaper algorithm frame transmission (no conflicting traffic)

This graph shows what happens when the *credit* parameter has a value of zero and there is no conflicting traffic (there is no higher priority traffic awaiting transmission, and there is no frame being transmitted out of the port). At time t1, a frame is queued for transmission. The frame can be transmitted from the queue immediately (as there is zero or more *credit* and there is no conflicting traffic). When the frame is being transmitted, the value of the *credit* parameter is decreased at a value equal to the *send slope* parameter. In the example, that would be at a rate of -75 Mb/s. At time t2, the frame transmission is complete, and the value of the *credit* parameter increases at 25 Mb/s (which is

the value of the *idle slope* parameter in the example) until it reaches a value of zero (time t3). During the time that the *credit* parameter has a value less than zero (between time t1 and time t3), no other frame transmission is allowed to be initiated from the associated queue. This ensures that the queue does not use more than its allocated bandwidth.

Figure 55 shows a graph that displays the value of a *credit* parameter over time.

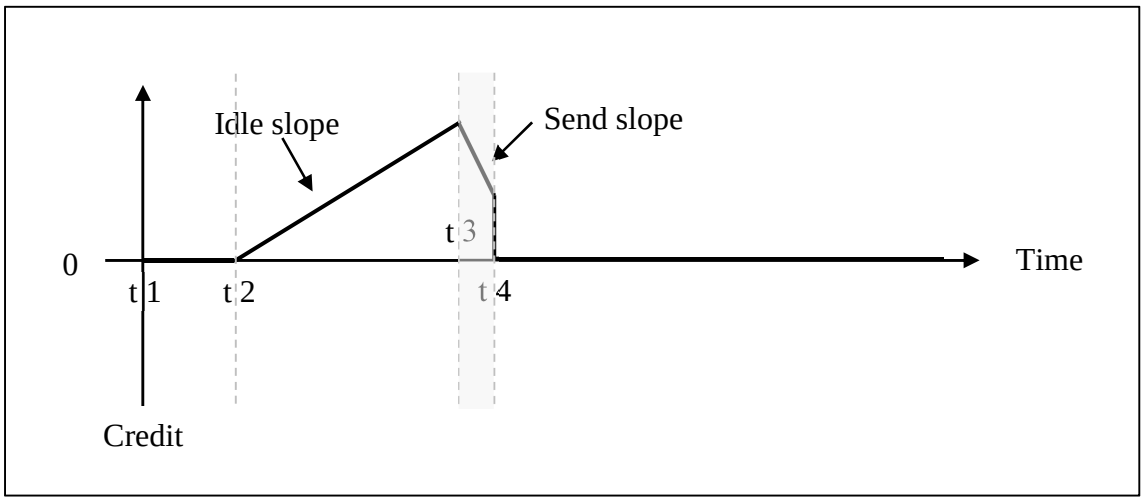

Figure 55: Credit-shaper algorithm frame transmission (conflicting traffic)

At time t1, there are no frames waiting in the queue for transmission, but conflicting frames are being transmitted out of the port associated with the queue. At time t2, a frame is queued for transmission. Since the value of the *credit* parameter is zero, the algorithm will allow the transmission of the frame, but as a conflicting frame is being transmitted out of the port, the frame cannot be transmitted. The transmission of the conflicting frame ends at time t3.Whilst the conflicting frame is being transmitted, the value of the *credit* parameter increases at a value that is equal to the value of the *idle slope* parameter. From the time t2 to the time t3, the frame waits in the queue. At time t3, the transmission of the queued frame begins, and at time t4 the transmission of the frame is complete. During the transmission of the frame, the value of the *credit* parameter decreases at a value that is equal to the value of the *send slope* parameter. The transmission of the frame does not consume all of the *credit* that is available. At time t4, there are no other frames in the queue and thus the value of the *credit* parameter is set to zero.

Figure 56**Error! Reference source not found.** shows a further graph that displays the value of a *credit* parameter over time.

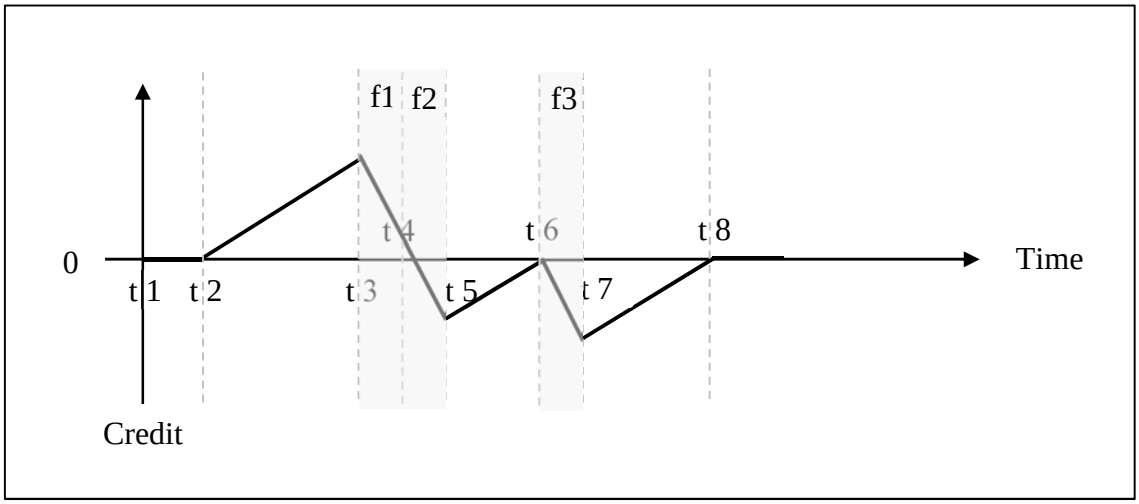

Figure 56: Credit-shaper algorithm frame transmission (burst traffic)

At time t1, there are no frames waiting in the queue for transmission, but conflicting frames are being transmitted out of the port associated with the queue. At time t2, three frames are queued one after the other. As soon as the first frame is queued, the value of the *credit* parameter increases at a rate equal to the value of the *idle slope* parameter as there is conflicting traffic being transmitted by the port. At time t3, the conflicting traffic has been transmitted and the first frame in the queue (f1) is transmitted. Whilst the frame is being transmitted the value of the *credit* parameter decreases at a rate equal to the value of the *send slope* parameter. At time t4 the transmission of the first frame is complete. As the value of the *credit* parameter is positive at this point, the second frame (f2) is transmitted and the value of the *credit* parameter continues to decrease at a rate equal to the value of the *send slope* parameter. At time t5 the transmission of the second frame is complete. As the value of the *credit* parameter is now negative, the third frame cannot be transmitted. The value of the *credit* parameter increases at a rate equal to the value of the *idle slope* parameter. At time t6 the value of the *credit* parameter reaches a value of zero. There is no other conflicting traffic so the third frame (f3) is transmitted. Whilst this frame is being transmitted, the value of the *credit* parameter decreases at a rate equal to the value of the *send slope* parameter. At time t7 the transmission of the third frame is complete and the *credit* parameter increases at a rate equal to the value of the *idle slope* parameter. At time t8 the value of the *credit* parameter reaches zero. As there are no queued frames and no conflicting traffic, the value of the *credit* parameter remains at zero.

4.2.2.3.2.2 Deriving Actual Bandwidth Requirements from the Advertised TSpec

The forwarding and queuing rules use bandwidth parameters that are defined in terms of the actual bandwidth used when frames are transmitted on the medium that supports the underlying MAC. The bandwidth is not defined as the size of the payload that is carried within frames. MSRP specifies the size of the payload that forms part of a stream. This value does not take into consideration the per-frame overhead that is associated with transmitting the frame over a given medium. When calculating the amount of bandwidth to reserve for a given queue, it is necessary to take into consideration the per-frame overhead associated with the transmission of frames.

When calculating the bandwidth consumed by a stream, it is assumed that the stream data is essentially of constant size and transmission rate. The *maximum frame rate* is calculated as follows:

- *Maximum frame rate = maximum interval frames* × (1 / *class measurement interval*)

The *maximum frame rate* is measured in frames per second. The *maximum interval frames* is measured in units of the number of frames transmitted every *class measurement interval*. The *class measurement interval* is measured in seconds.

Stream class A has a *class measurement interval* of 125µs. If, for example, an end station supports stream class A and is transmitting an audio stream at one frame per *class measurement interval* (i.e., it has *maximum interval frames* of one), the maximum frame rate would be calculated as follows:

= *maximum interval frames* × (1 / *class measurement interval*)

= $1 \times (1 / 125 \times 10^{-6})$

= 1×8000

= 8000 frames per second

It is possible to determine the *per-frame overhead* that is added to each payload. The actual *per-frame overhead* is dependent on the protocol stack and the underlying MAC technology. The *actual bandwidth* that is required to support a given stream is defined as follows:

- *Actual bandwidth* = (*per frame overhead* + *payload size*) × *max frame rate*

The *actual bandwidth* is measured in octets per second, and the *per-frame overhead* and *payload size* are measured in octets.

For VLAN tagged Ethernet frames, the overhead associated with each frame is:

- *Inter frame gap* (IFG): 12 octets
- *Preamble*: 8 octets
- Ethernet header (with VLAN tag): 18 octets
- *Trailer*: 4 octets

The total *per-frame overhead* is thus 42 octets.

If, for example, an end-station advertised that the maximum payload associated with the stream is 80 octets (via the *talker advertise* TSpec field), the actual reservation of bandwidth performed by MSRP's MAP component (as represented by the *idle slope* parameter of the queue associated with the stream) will also have to take into consideration the 42 octets of *per-frame overhead*. The *actual bandwidth* will be calculated as follows:

= (*per frame overhead* + *payload size*) × *max frame rate*

= (42 + 80) × 8000

= 112 × 8000

= 976000 octets/s

= 7.808 Mb/s

4.2.2.4 Stream Reservation Protocol Domain

It is possible to connect non-AVB-aware stations to AVB-aware stations and it is also possible for AVB-aware stations to associate different priority values with the SR classes that they support. For example, a bridge may use priority three to identify class A streams, and a second bridge that it is connected to may be configured to use priority four to identify class A streams. Given this, *stream reservation protocol domains* (*SRP domains*) are created to ensure that stream requirements are met and not interfered with from devices outside of the domain. For example, a non-AVB aware end station could be connected to an AVB-aware bridge. This end station is able to transmit VLAN tagged frames that are tagged with a priority value that is being associated with one of the supported SR classes. If this were to happen, the reserved bandwidth for the particular SR class would be oversubscribed resulting in packet loss. Thus, when these frames enter the SRP domain, their priority

values need to be adjusted such that they do not conflict with priority values being used by SR classes (see Section 4.2.2.4.2 “Priority Regeneration”).

Each supported SR class of each port of each bridge of a set of connected bridges needs to be identified as either being at the edge of the set of bridges or not. An SR class of a port is considered to be at the edge of the set of connected bridges if the station at the other end of the link does not support the credit-shaper algorithm for stream transmission, does not support MSRP, or the SR class uses a different priority value to tag the SR class’s stream frames with. The set of bridges form SRP domains for the supported SR classes.

Figure 57 shows an example network composed of three four-port bridges. Bridge B3 is not AVB-aware, and bridge B1 and bridge B2 are AVB aware. Bridge B1 and bridge B2 support SR class A using priority three and SR class B using priority two. End stations End 1 to End 5 are also compliant with the AVB standards and support SR class A using priority three and SR class B using priority two. The right hand port of bridge B2 has been marked as an edge port (this port is known as a *domain boundary port*) for SR class A and SR class B as it is connected to a port of a bridge that is not AVB-aware. All of the other ports of the AVB-aware stations are marked as not being at the edge (these ports are known as *domain core ports*) for both SR class A and SR class B.

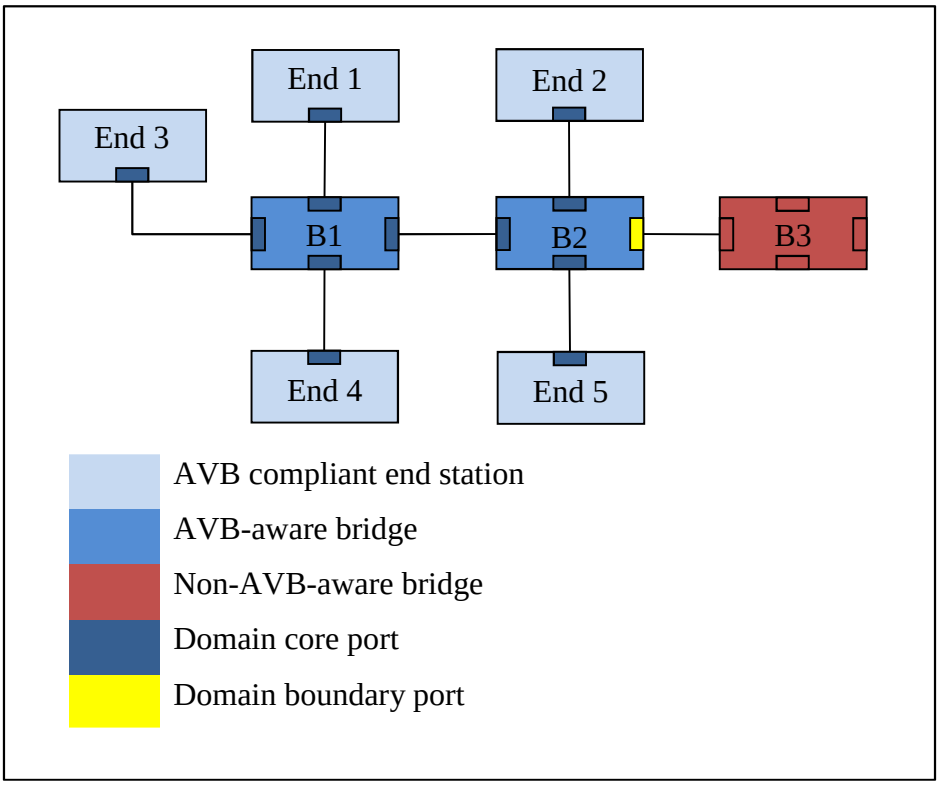

Figure 57: SRP domain with non-AVB bridge

Figure 58 shows another example network composed of three four-port bridges. All of the bridges are AVB aware. Bridge B1 and bridge B2, and End 1 through to End 5 support SR class A using priority three and SR class B streams using priority two. Bridge B3 and End 6 through to End 8 only support class B streams using priority two. As a result of this, the right hand port of bridge B2 is marked as a *domain boundary port* for SR class A. Any frames entering the port that have been tagged with a priority value of three will have their priority value adjusted.

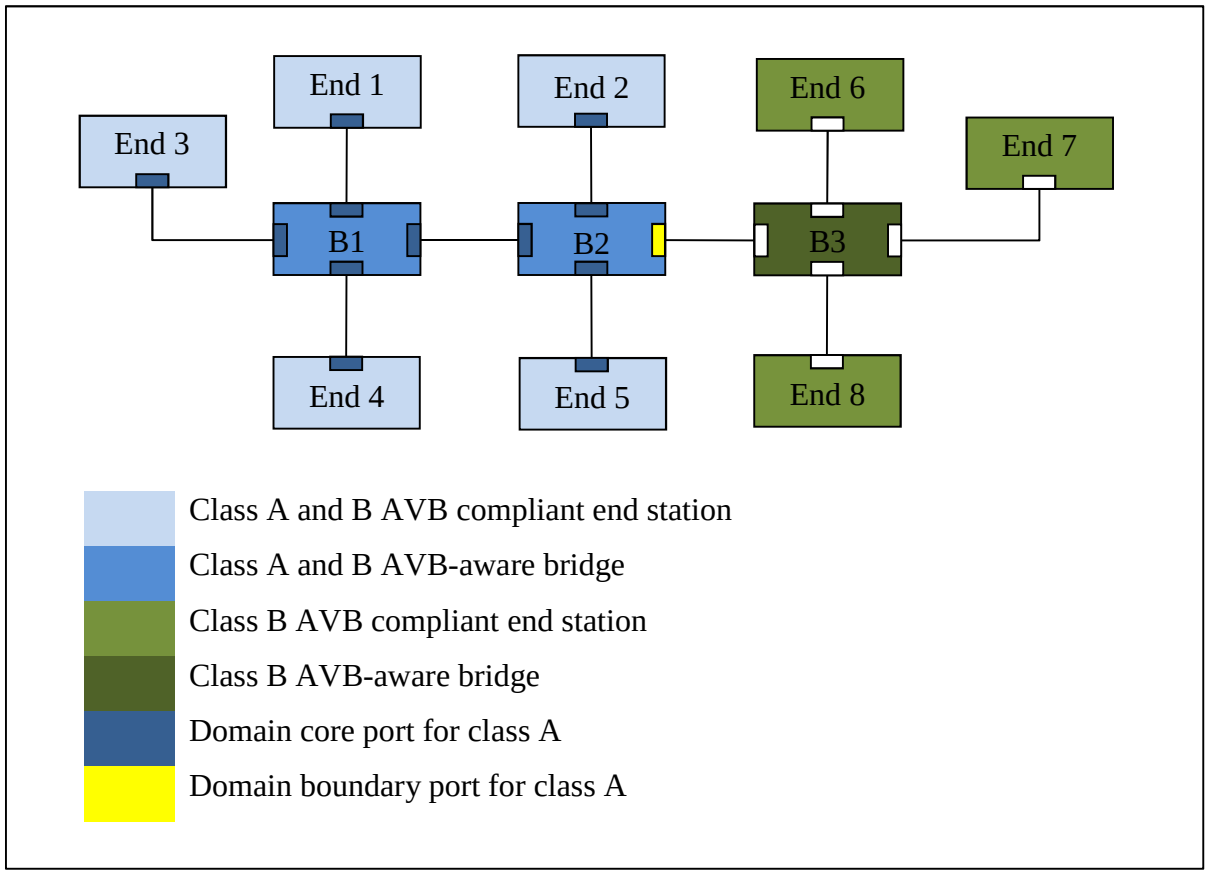

Figure 58: SRP domain for SR class A

Figure 59 shows the same example network as shown in Figure 58. In this figure, all of the ports of all of the stations are marked as *domain core ports* for SR class B as all of the stations support SR class B using priority two.

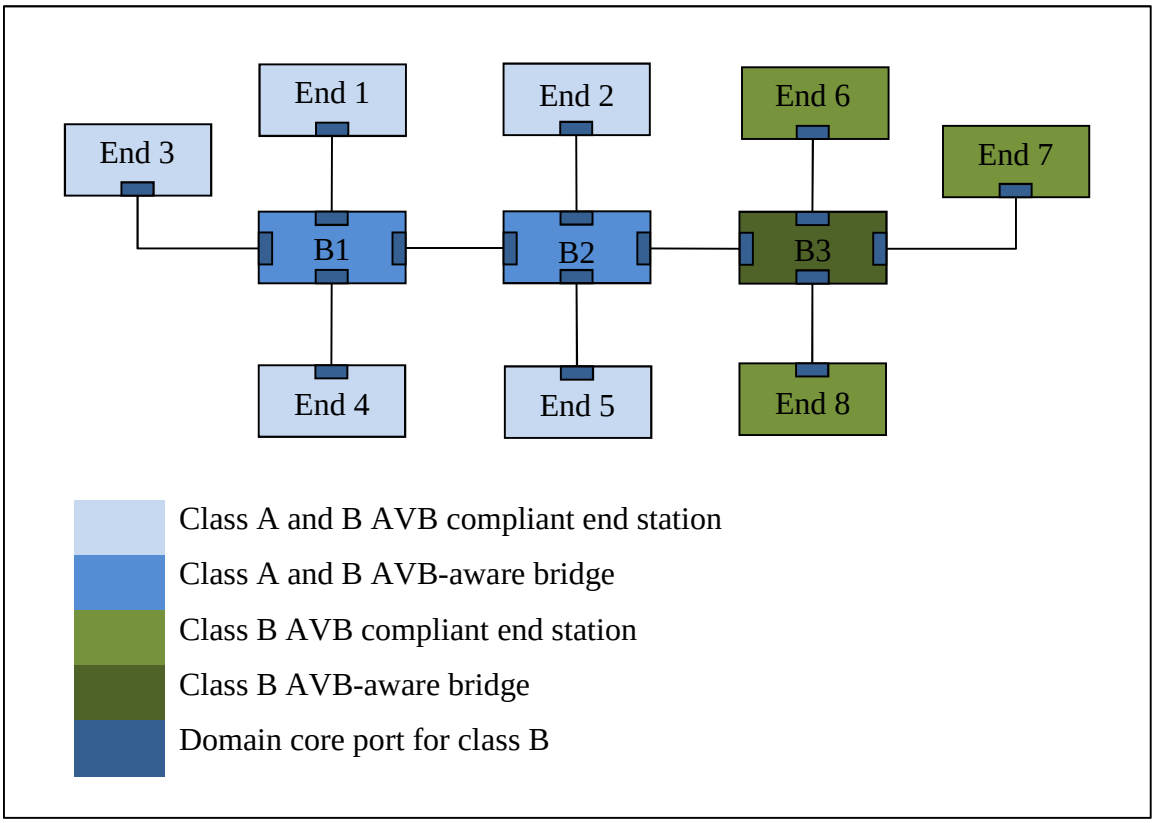

Figure 59: SRP domain for SR class B

4.2.2.4.1 Detection of a Stream Reservation Protocol Domain

MSRP is used to establish SRP domains. Neighbouring devices that have identical SR class characteristics are part of the same SRP domain and stream data may be transmitted between these devices. All systems that support a particular SR class are considered to be in the same SR domain if they use the same priority value for that SR class. A SRP domain boundary for a SR class exists at a point when neighbouring devices use different priority values for the same SR class.

In addition to the *talker* and *listener* attributes, MSRP defines a *domain* attribute. This attribute contains information that is used by a bridge port to determine the location of an SRP domain boundary. The structure of a domain attribute is as follows:

- *SR class ID* (one octet): The *SR class ID* field is used to represent an SR class that is supported by the port.
- *SR class priority* (one octet): The *SR class priority* field is used to represent the priority value that is associated with the SR class (identified by the *SR class ID* field).

- *SR class VID* (two octets): The *SR class VID* field is used to represent the VID of the VLAN that is associated with the SR class (identified by the *SR class ID* field).

Each AVB device is aware of its characteristics and functionality. When it is initialised, it is able to declare *domain* attributes for each of its supported SR classes. For each SR class, of each port of a bridge, MSRP determines whether the port is a *domain core port*, a *domain boundary port* or that the port is not part of the domain. When the *domain* attribute of the port at the other end of a link has the same value as the *domain* attribute on the local end of a link, the port is considered to be a *domain core port* for the SR class, otherwise it is not considered to be a *domain core port*.

Once registered on a bridge's port, MSRP does not propagate *domain* attributes across to the other ports of a bridge. The purpose of the *domain* attribute is to determine the SR class characteristics of a station at the other end of an attached link.

4.2.2.4.2 Priority Regeneration

When a frame is received by a bridge, the priority of that frame is regenerated using the priority information contained within the frame and the *priority regeneration table* of the port from which the frame was received

Each port has associated with it a *priority regeneration table*. This table has eight entries that correspond to the eight possible priority values: zero through to seven. Each entry specifies the received priority value and the corresponding regenerated priority value. Table 9 shows the default regeneration values. These are initially used in the *priority regeneration table* of each port.

Received priority	Default regenerated priority
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Table 9: *Priority regeneration table*

In a bridge that supports SRP, each reception port has a *domain boundary port priority regeneration override table* associated with it. This table associates a priority value with each SR class supported by the bridge and a regenerated priority value to be used when the port has been determined to be a *domain boundary port*.

Table 10 shows the default values for priority and regenerated priority for two SR classes (SR class A and SR class B) for a *domain boundary port*. This ensures that any data arriving at the boundary port using a priority value that is associated with an SR class is regenerated so that it does not interfere with any stream data.

SR class	Default priority	Default regenerated priority for SRP domain boundary ports
A	3	0
B	2	0

Table 10: Default *domain boundary port* priority regeneration override values table

Assume that an AVB-aware bridge uses the default *domain boundary port* priority regeneration override values that are defined in Table 10. If a non-AVB endpoint device transmits a frame destined to the AVB-aware bridge with a priority of three, when the frame arrives at the *domain*

boundary port of the AVB-aware bridge, the priority of that frame will be adjusted to zero before it is transmitted further. This then prevents that frame from interfering with any stream data (using priority three) in the SRP domain.

4.2.2.5 Talker Behaviour

A talker end station that wants to make use of the credit-based shaper algorithm characteristics for stream delivery has to use the priorities that the bridges in a network recognise as being associated with SR classes exclusively for transmitting stream data. A talker end station exhibits the transmission behaviour for frames that are part of time-sensitive streams. This behaviour is consistent with the operation of the credit-based shaper algorithm. This behaviour is seen for the transmission of individual streams, and for combined streams' data transmitted from the port. This is illustrated in Figure 60.

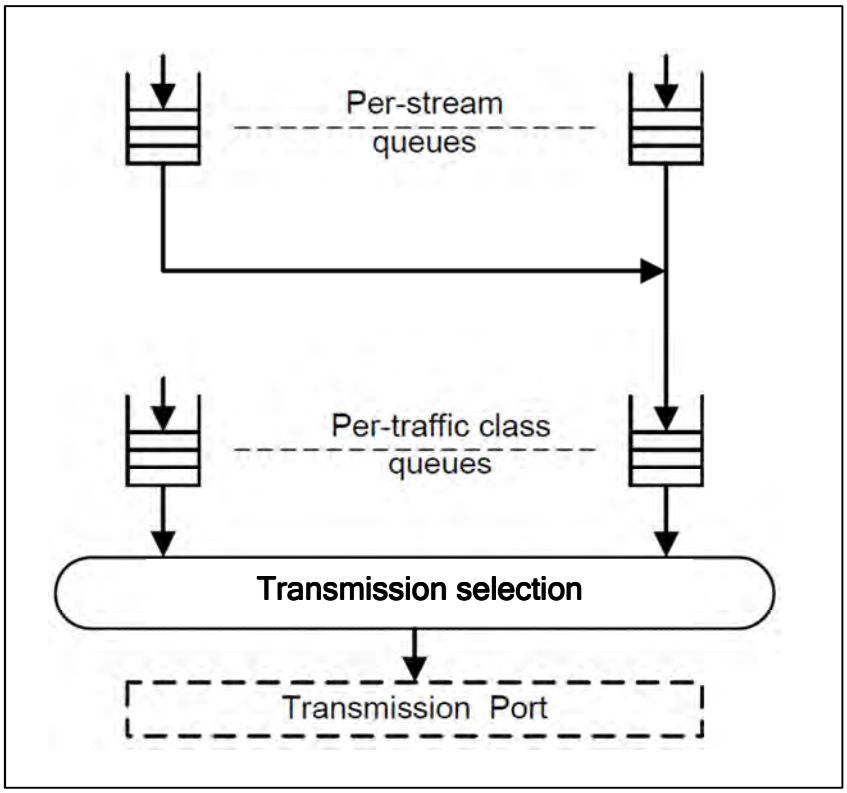

Figure 60: Queuing model for a talker station [68]

A talker end station has a per-stream queue, and it places stream frames into this queue. The placement of frames into this queue is based on the parameters of the *talker advertise* attribute for the stream. During each class measurement interval, it is allowed to place the number of frames it

advertised in the *talker advertise* into the queue, and each frame may not be larger than the maximum frame size advertised in the *talker advertise*.

Each queue associated with each stream uses the credit-based shaper algorithm to determine the rate at which data frames for the stream are placed into the appropriate port's outbound queue. The port's outbound queue then in turn uses the credit-based shaper algorithm to select which frames are to be transmitted out of the queue.

The end station exhibits the same transmission selection characteristics that are described in Section 4.2.2.3 ("Transmission Selection Algorithms" on page 92) for the transmission of frames.

4.3 Conclusion

When transmitting realtime stream data across a digital network, it is necessary for this data to be transmitted in a timely and deterministic fashion. IEEE 1394 and Ethernet AVB networks natively provide mechanisms to provide this deterministic transmission to ensure that the networks are not oversubscribed. IEEE 1394 provides deterministic transmission via a bus arbitration mechanism. This ensures that all of the devices that have acquired network resources are able to transmit isochronous data every isochronous cycle. Ethernet AVB devices make use of the credit-shaper algorithm to guarantee network resources to streams. These mechanisms ensure that data frames are not discarded due to network congestion and ensures that they arrive timeously.

Chapter 5 Timing and Synchronisation

When transmitting multiple audio and video streams across a network, it must be possible to present these streams so that they are rendered correctly with respect to each other. For example (as shown in Figure 61) a video camera could be capturing images and a microphone the audio associated with those images. The video stream could traverse a network to reach a video display, and the audio stream may traverse the network to reach a pair of speakers. The audio and video streams may take different paths through the network and this could result in their data arriving at their respective destination devices at different times. It is essential that the video and its associated audio be synchronised so that the audio corresponds to the images on the video display. It should be possible for transmitting devices to be able to tell receiving devices the time at which video and audio samples should be presented (i.e., it should be able to provide the receiver(s) with a *presentation time*).

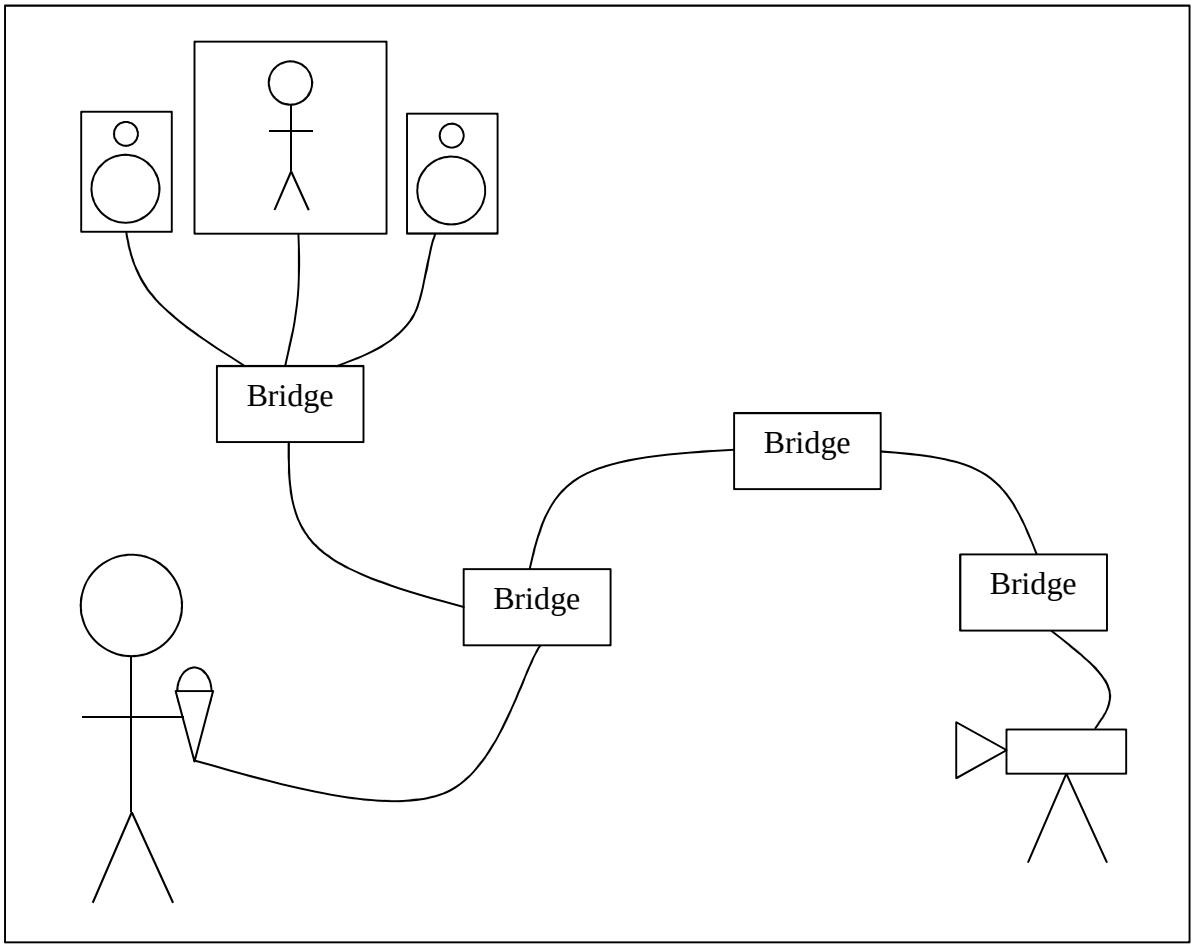

Figure 61: Example stream synchronisation

It is also necessary to achieve *wordclock synchronisation*: for example, a signal from a microphone could be fed into an analogue to digital converter (ADC) where it is sampled at a certain rate (nominally 48 kHz, for example). This signal could then be transmitted across a digital network to a speaker. Once that signal arrives at the speaker, it may be fed into a digital to analogue converter (DAC) such that it can be played back on the speaker. The DAC should run at the same rate as the ADC (48 kHz, for example), but this is unlikely to be the case. If the ADC is running faster than the DAC, it will produce more samples than the DAC is consuming causing a buffer overrun. In the opposite case, where the ADC is running at a slower rate than the DAC, the ADC is producing fewer samples than the DAC is trying to consume resulting in a buffer under-run. These conditions result in glitches in audio and video playback. There is a need to synchronise the sampling clocks of the devices that produce streams, and the sample clocks of the devices that receive streams. This is known as wordclock synchronisation.

In order to control the presentation time of audio and video samples, and to allow for wordclock synchronisation, IEEE 1394 and AVB devices need to share a common sense of time. This chapter discusses the native synchronisation mechanisms that are in place within each of these networking technologies such that devices on these networks are able to share a common sense of time. There is a need to understand these synchronisation mechanisms in order to develop mechanisms that allow for cross network stream synchronisation to take place.

5.1 IEEE 1394

On an IEEE 1394 bus, each node has a *cycle time* register that is incremented by a clock on the node. This clock has a frequency of 24.576 MHz. The format of the *cycle time* register is shown in Figure 62. It is composed of a 7-bit *second count* field, a 13-bit *cycle count* field, and a 12-bit *cycle offset* field. These fields are used as follows:

- The *cycle offset* field is updated until it reaches a value of 3071. At this point it rolls over to zero and the *cycle count* field is updated by one. The roll over happens every 125µs and is used to trigger the transmission of *cycle start* packets. Once every 125µs, the root node broadcasts a *cycle start* packet onto the bus. The cycle start packet contains the root node's *cycle time* register value. The *cycle time* register values of all the nodes are synchronised by the cycle start packet.

- The *cycle count* field is updated until it reaches a value of 7999. At this point it rolls over to zero and the *second count* field is updated by one. This happens every second.
- The *second count* field is updated until it reaches a value of 127. At this point it rolls over to a value of zero.

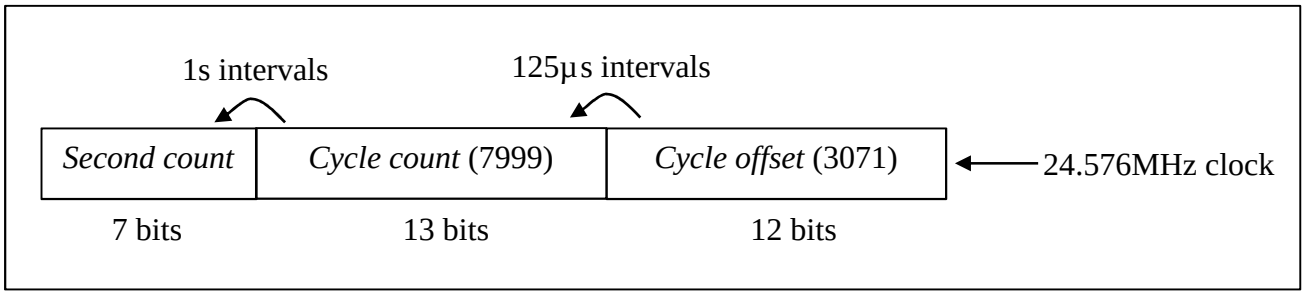

Figure 62: *Cycle time* register (adapted from [4])

An IEEE 1394 node also implements a *bus time* register. This register is composed of a *second count hi* field and a *second count lo* field. The *second count lo* field is an alias for the *second count* field of the *cycle time* register and is thus 7-bits in size. The *second count hi* field is 25-bits in size and is incremented when the value of the *second count lo* field rolls over from a value of 127.

Using this mechanism, all of the nodes on an IEEE 1394 bus share a common sense of time. This allows for transmitting IEEE 1394 nodes to timestamp their packets with presentation times, as shown in Section 6.2 “Timing and Synchronisation”.

5.1.1 Cross IEEE 1394 Bridge Synchronisation

IEEE 1394 buses (consisting of IEEE 1394 nodes) can be bridged together with IEEE 1394 bridges [62] to form a *net*. An IEEE 1394 bridge consists of two portals. Each portal is a connection from a bridge to an IEEE 1394 bus. A bridge is capable of forwarding asynchronous and isochronous subactions. The forwarding characteristics of a bridge are determined by routing information that it maintains.

Each IEEE 1394 bus forms its own arbitration domain and has a cycle master that provides uniform time to that bus. Each bridge portal has an independent 24.576 MHz cycle timer. These cycle timers may not advance at exactly the same rate resulting in one advancing at a higher rate than the other.

In order for an IEEE 1394 bridge to reliably transport isochronous stream data, it is necessary for all cycle masters within a net to maintain *frequency synchronisation* with each other. Without this, cycle time drift might grow large enough to cause isochronous buffer overruns or underruns. For example, an IEEE 1394 bridge may receive isochronous packets on a portal at a faster rate than it is transmitting them on its second portal. This would occur if the cycle time of the cycle master of the first portal's bus advances at a higher rate than the cycle time of the cycle master of the second portal's bus. The simplest method of achieving frequency synchronisation is to maintain an effective cycle offset phase difference of zero (known as *phase synchronisation*) between cycle timers on adjacent buses. Cycle offset phase synchronisation occurs when cycle start events are simultaneous.

A net has a single cycle master that provides cycle offset for the entire network. This is the cycle master of one of the IEEE 1394 buses and is known as the *net cycle master*. IEEE 1394 bridges distribute the *net cycle offset* throughout a net.

5.1.1.1 IEEE 1394 Bridge Portals

A single portal of a net is known as the *prime portal*. The location of the prime portal determines the location of the net cycle master, i.e., the net cycle master is located on the bus to which the prime portal is connected. Each bus of a net contains an *alpha portal*. This portal snoops and forwards transaction subactions addressed to the prime portal. The prime portal is also the alpha portal on the IEEE 1394 bus to which it is connected. A *co-portal*, from the perspective of a particular IEEE 1394 bridge's portal is the other portal of the same bridge. A *subordinate portal* is a portal that is neither an alpha nor a prime portal. Figure 63 shows an example net that contains two IEEE 1394 bridges and two IEEE 1394 nodes. In this example, the left hand portal of Bridge A is the prime portal. As a result of this, the right hand portal of Bridge A and Bridge B are alpha portals. The left hand portal of Bridge B is a subordinate portal.

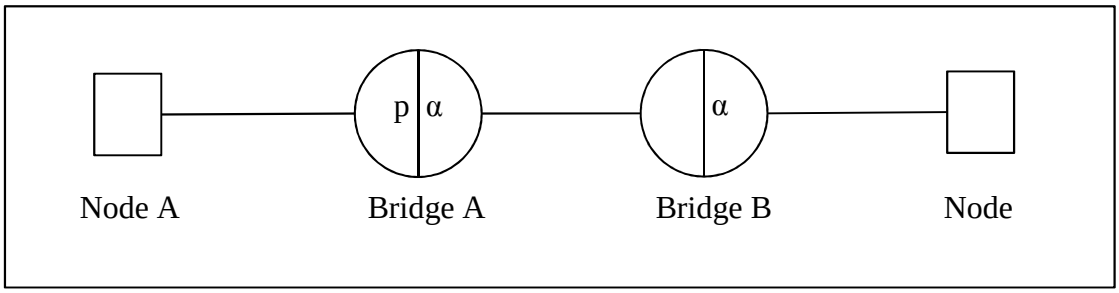

Figure 63: An example net

In the context of cycle time synchronisation, a *downstream* cycle master or a downstream portal is one that has more bridges between itself and the net cycle master than the portal to which it is compared. For example, in Figure 63, if Node A is the net cycle master, the alpha portal of Bridge B is downstream relative to the alpha portal of Bridge A. In the context of an isochronous stream, within a bridge the downstream portal is the one with more bridge portals between itself and the node transmitting the stream.

In the context of cycle time synchronisation, an *upstream* cycle master or an upstream portal is one that has fewer bridges between itself and the net cycle master than the portal to which it is compared. For example, in Figure 63, if Node A is the net cycle master, the alpha portal of Bridge A is upstream relative to the alpha portal of Bridge B. In the context of an isochronous stream, within a bridge the upstream portal is the one with fewer bridge portals between itself and the node transmitting the stream.

On the bus that contains the prime portal, the portals of the other bridges on the bus obtain cycle time directly from the net cycle master. These portals communicate their cycle offset to their co-portals which regulate cycle start events on their own buses.

5.1.1.2 Phase Synchronisation

The IEEE 1394 bridges that interconnect IEEE 1394 buses into a net are responsible for maintaining phase-locked synchronisation between the net cycle master and all other cycle masters in the net. Phase lock is achieved when the value of the *cycle offset* field of the *cycle time* register is identical for two cycle masters separated by a bridge. An IEEE 1394 bridge accomplishes phase lock by measuring the cycle offset difference between the cycle master on the upstream portal's bus and the cycle master on the alpha portal's bus. This difference is adjusted to account for propagation delays

between the portals and their respective cycle masters. When the difference (measured in ticks of a 24.576 MHz cycle timer) is nonzero, the alpha portal either adjusts its own cycle time (if it is cycle master) or transmits a cycle master adjustment packet to the cycle master on its bus. The cycle master adjustment packets (known as *go fast* or *go slow* packets) allow an alpha portal to instruct a cycle master to delay or hasten the advent of the next cycle synchronisation event by one tick of its 24.576 MHz cycle timer.

For an alpha portal to calculate whether a phase adjustment is required, four data items are sufficient:

- Simultaneous samples of the *cycle offset* field of the *cycle time* register of both bridge portals.
- The propagation delay between the alpha portal and the downstream cycle master.
- The propagation delay between the upstream portal and the upstream cycle master.

Figure 64 shows these data items between two IEEE 1394 buses in a net. The upstream and downstream (with respect to the net cycle master) cycle masters are marked CM and CM'. The upstream portal is labelled *u* and the alpha portal is labelled *α*.

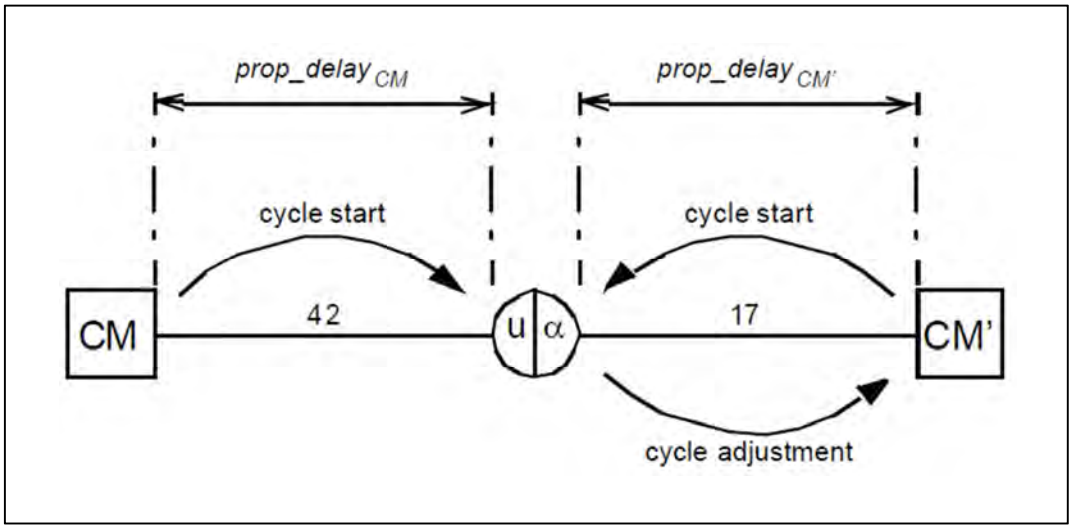

Figure 64: Phase synchronisation between two IEEE 1394 buses [62]

An IEEE 1394 bridge is able to calculate the phase difference between its two portals with the following formula

$$(cycle\ offset_{alph} + propagation\ delay_{CM'}) - (cycle\ offset_{upstream} + propagation\ delay_{CM})$$

Each portal of an IEEE 1394 bridge measures the propagation delay between itself and the cycle master on its bus by means of ping packets. The propagation delay is half the round-trip delay measured by a ping packet.

Both bridge portals have an independent 24.576 MHz cycle timer whose value can be sampled via the *cycle time* register. Once each isochronous period, the phase of both cycle timers (visible via the *cycle offset* field of the *cycle time* register) is sampled. This result of this is combined with the propagation delays according to the formula above. If the result of the calculation is a negative value, the downstream cycle master is running slower than the upstream cycle master and the alpha portal should instruct it to reduce the threshold value for the impending cycle start by one cycle timer tick. If the result of the calculation is a positive value, the downstream cycle master is too fast and the alpha portal instructs it to increase the threshold value by one cycle timer tick. Adjustments made by the alpha portal to the cycle master's threshold value are in effect only for the next *cycle count* increment. After this, the threshold reverts to the nominal 3071 cycle timer ticks.

Table 11 shows an example phase difference calculation for Figure 64. The data shown in the table is observed by a single IEEE 1394 bridge. After the calculation is performed, it is determined that the downstream cycle master (CM') is running slower than the upstream cycle master (CM) and thus it is instructed to temporarily decrease its *cycle count* increment threshold by one time tick.

Observed data	*cycle offset*$_{\text{upstream}}$	25 ticks
	propagation delay$_{\text{CM}}$	7 ticks
	cycle offset$_{\text{alpha}}$	15 ticks
	propagation delay$_{\text{CM'}}$	5 ticks
	phase difference	-12 ticks
Result	The alpha portal will transmit a cycle master adjustment packet to temporarily decrease the *cycle count* increment threshold by one cycle time tick (*go fast*).	

Table 11: Example phase difference calculation based on Figure 64 [62]

5.1.1.3 Cycle Master Adjust Packet

A cycle master adjust packet instructs the recipient to adjust the interval between successive cycle synchronisation events. The type of cycle master adjust packet indicates the adjustment to be made to the cycle master's behaviour at the next cycle synchronisation event. The adjustment causes a temporary override of the threshold value above which the *cycle offset* field of the *cycle time* register wraps around to zero and causes the *cycle count* field of the *cycle time* register to increment.

- A *go slow* packet specifies that the threshold should be set to 3072. This delays the next cycle synchronisation event by one tick of the cycle master's 24.576 MHz cycle timer.
- A *go fast* packet specifies that the threshold should be set to 3070. This hastens the next cycle synchronisation event by one tick of the cycle master's 24.576 MHz cycle timer.

Upon the next cycle synchronisation event, the threshold value is restored back to 3071.

5.2 Ethernet AVB

IEEE 802.1 AS [69] [72] [73] defines the *generalised precision time protocol* (gPTP) which allows systems on a bridged LAN to all share a common sense of time. The protocol defines two types of *time-aware systems*:

- *Time-aware end station*: A time-aware end station is a device that is capable of acting as a source of synchronised time and/or a destination of synchronised time (using gPTP).
- *Time-aware bridge*: A time-aware bridge is a device that is capable of communicating synchronised time received on one of its ports to its other ports (using gPTP).

A bridged LAN composed of these time-aware systems interconnected by LAN technologies supporting gPTP is known as a *time-aware bridged LAN*. This time-aware bridged LAN forms a *gPTP domain*. The gPTP domain defines the scope of gPTP message communication, state, operations, data sets and timescale. The time-aware systems form a synchronisation master/slave hierarchy with one of the time-aware systems selected to be the grandmaster.

Time synchronisation is performed with the selected grandmaster sending its current time (amongst other information) to all of the time-aware systems directly attached to it. Each one of the time-aware systems that receives this information corrects the received time by adding the propagation time

needed for the information to transit the communication path from the grandmaster. A time-aware bridge will forward the corrected synchronised time information (including additional delays in the forwarding process) to all of its attached time-aware systems. For this mechanism to work, two time intervals need to be precisely known:

- The time taken for the synchronised time information to transit the communication path between two time-aware systems. Each time-aware system in a gPTP domain, for each of its ports, measures the delay on the link between itself and the time-aware system that it is connected to.
- The forwarding delay through bridges (called the *residence time*). The measurement of the *residence time* is local to a bridge and must be less than or equal to 10ms.

Any time-aware system that has clock sourcing capabilities is a potential grandmaster. gPTP defines a selection process that ensures that all of the time-aware systems in a gPTP domain use the same grandmaster. This selection process is known as the *best master clock algorithm* (BMCA). All of the time-aware systems on a time-aware bridged LAN continuously participate in the BMCA (even if they are not grandmaster capable) and thus should a "better" clock enter the time-aware bridged LAN, it will be selected as the grandmaster. If a time-aware bridged LAN were to become divided, for example, the BMCA will ensure that a grandmaster exists on each segment that is created. The BCMA will select the "best" grandmaster-capable time-aware station as the grandmaster of the time-aware bridged LAN segment. When this happens, two separate gPTP domains are formed.

Time-aware systems in a gPTP domain only communicate gPTP information directly with other time-aware systems. Non-time-aware bridges cannot be used to relay gPTP information as they slow timing convergence and introduce extra jitter and wander.

5.2.1 gPTP Messages

For full-duplex point-to-point links, all gPTP communication occurs via *precision time protocol* (PTP) messages. There are two PTP *message classes*:

- *Event message class*: these messages are time-stamped on egress from a time-aware system and ingress to a time-aware system.
- *General message class*: these messages are not time-stamped.

Each PTP *message class* also has a number of *message types*. These message types will be introduced in the relevant sections.

5.2.1.1 Generation of *Event Message* Timestamps

All *event messages* are time-stamped on egress from, and ingress to, a time-aware system. The timestamp value is the time, relative to the local clock of that time-aware system, at which the *message timestamp point* of the message passes the *reference plane*. The *message timestamp point* is the point within an *event message* at which a timestamp is taken, and is defined as the beginning of the first symbol following the start of frame delimiter. The *reference plane* marks the boundary between the time-aware system and the network media. These are illustrated in Figure 65.

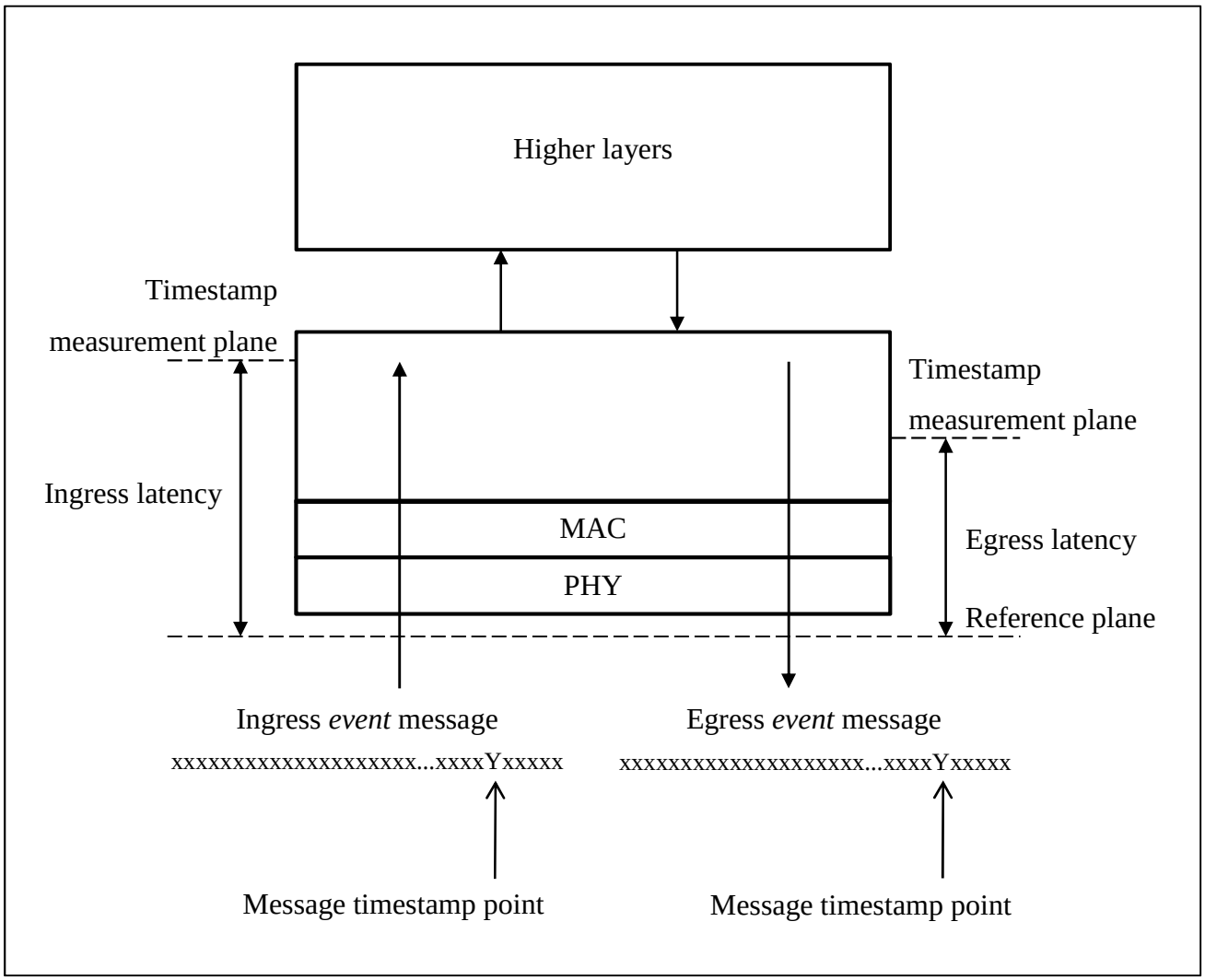

Figure 65: Definition of *message timestamp point, reference plane, timestamp measurement plane*, and latency constants (adapted from [69])

If *event message* timestamps are generated using a point other than the *message timestamp point* (within an *event message*), then the generated timestamps should be corrected by the time interval between the actual time of detection and the time that the *message timestamp point* passed the *reference plane*. If these corrections are not made, a time offset between time-aware systems results.

If timestamps are generated at a *timestamp measurement plane* that is different from the *reference plane*, the generated timestamps should be corrected for these offsets. Thus, the egress and ingress timestamps should be corrected as follows:

- *egress timestamp = egress measured timestamp + egress latency*
- *ingress time stamp = ingress measurement timestamp - ingress latency*

Where,

- *egress timestamp* and *ingress timestamp* are the timestamp values when the *message timestamp point passed* the *reference plane*,
- *egress measured timestamp* and *ingress measured timestamp* are the timestamp values measured relative to the *timestamp measurement plane*,
- *egress latency* and *ingress latency* are the latencies between these measurement planes.

The *timestamp measurement plane*, relative to the *reference plane*, may be different for inbound and outbound *event messages*, and thus the *egress latency* and *ingress latency* values are likely to be different. Failure to make these corrections results in a time offset between the slave and master clocks.

PTP peer delay measurements (see Section 5.2.4 “PTP Peer Delay Protocol”) calculates the mean propagation time on a link and thus requires that two-way delay be symmetrical for the highest accuracy. The latency corrections allow transmission delays to be measured such that they appear fixed and symmetrical to gPTP even though there might be substantial asymmetry and transmission variation.

5.2.2 Best Master Selection and Network Establishment

The BMCA is a distributed algorithm that is used to select the "best" clock of a gPTP domain as the root time-aware system of that gPTP domain, and if the root time-aware system is grandmaster capable, the grandmaster. The grandmaster will always be the root time-aware system, but the root time-aware system will not necessarily be the grandmaster (this would occur if none of the time-aware systems are capable of being grandmaster). The BMCA is also used to construct a time-synchronisation spanning tree. Synchronised time is communicated from the selected grandmaster (the root time-aware system) to other time-aware systems via the time-synchronisation spanning tree.

BMCA exchanges information between time-aware systems via *announce* messages. An *announce* message is a general message, and thus is not time-stamped. Each *announce* message contains information that identifies one of the time-aware systems as the root of the time-synchronisation spanning tree and, if the root time-aware system is grandmaster-capable, the grandmaster. Each time-aware system uses the information in the *announce* message it receives, along with its knowledge of itself, to determine which time-aware systems that it has knowledge of should be the root and, if grandmaster-capable, the grandmaster. Once an *announce* message is transmitted by a port, subsequent timing information transmitted by the port shall be derived from the grandmaster that is indicated in that *announce* message.

5.2.2.1 Time-aware System Characterisation

Each *announce* message carries information that is used to characterise the time-aware system. This information consists of a priority value, multiple clock characteristic values, and a clock identity value that is used to uniquely identify a time-aware system. When comparing two time-aware systems, these values are concatenated together to form an unsigned integer called a *system identity*. A *system identity* with a lower numeric value indicates a time-aware system that is a more capable root time-aware system than a time-aware system with a *system identity* with a higher numeric value.

The priority field can be used to determine whether a time aware system is grandmaster capable or not, and can be used by a network manager to force a certain time-aware system to be grandmaster. When comparing two time-aware systems, a time-aware system's priority value has the greatest influence on the outcome of the *system identity* comparison. The clock characteristic value includes

values that describe the class of the clock and its accuracy. When comparing two time-aware systems, if the systems have equal priority, their clock characteristics are compared. If the two time-aware systems have equal clock characteristics, then the clock identities are compared as a tie-breaker.

5.2.2.2 Examples of Grandmaster Selection

When a time-aware bridged LAN is in a steady state, the grandmaster of the gPTP domain periodically transmits an *announce* message announcing its presence and superiority. Time-aware end stations and bridges keep track of the current grandmaster. Figure 66 shows an example time-aware bridged LAN with time-aware end station A as the grandmaster. It periodically transmits *announce* messages. In the diagram, *GM* indicates *grandmaster*.

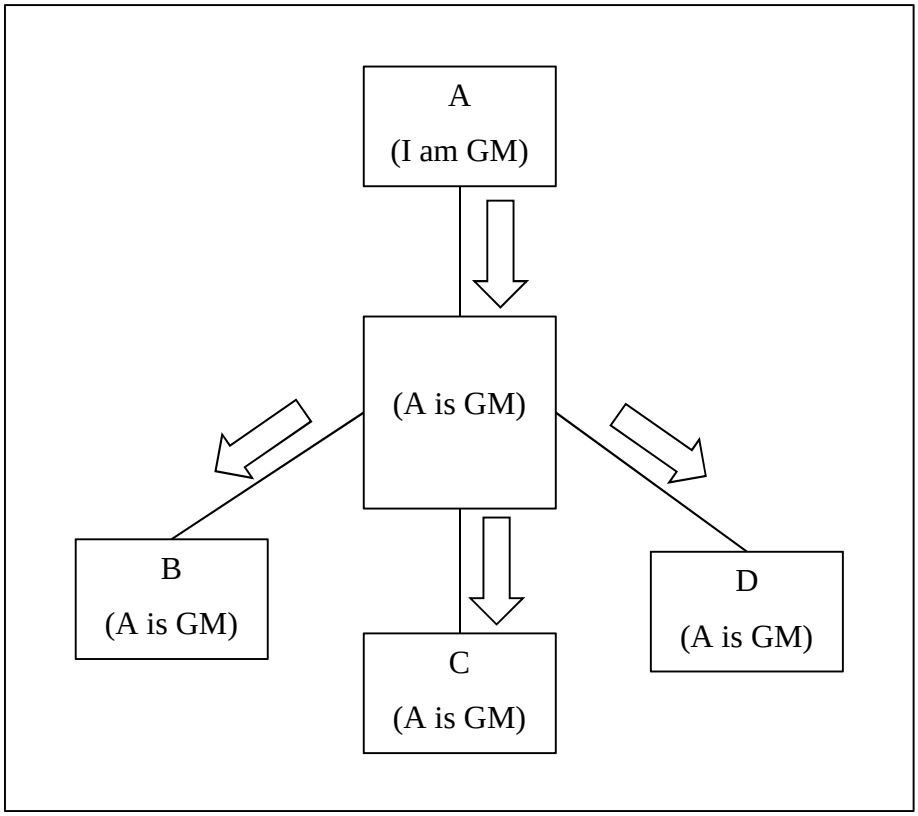

Figure 66: *Announce* message transmission in a steady state

In Figure 67, an additional time-aware end station (E) is added to the network. This end station receives an *announce* message from end station A and realises (by comparing the *system identity* in the *announce* message with its own *system identity*) that it is a better grandmaster than end station A. This could happen, for example, if the clock of time-aware system E is considered more accurate than the clock of time-aware system A.

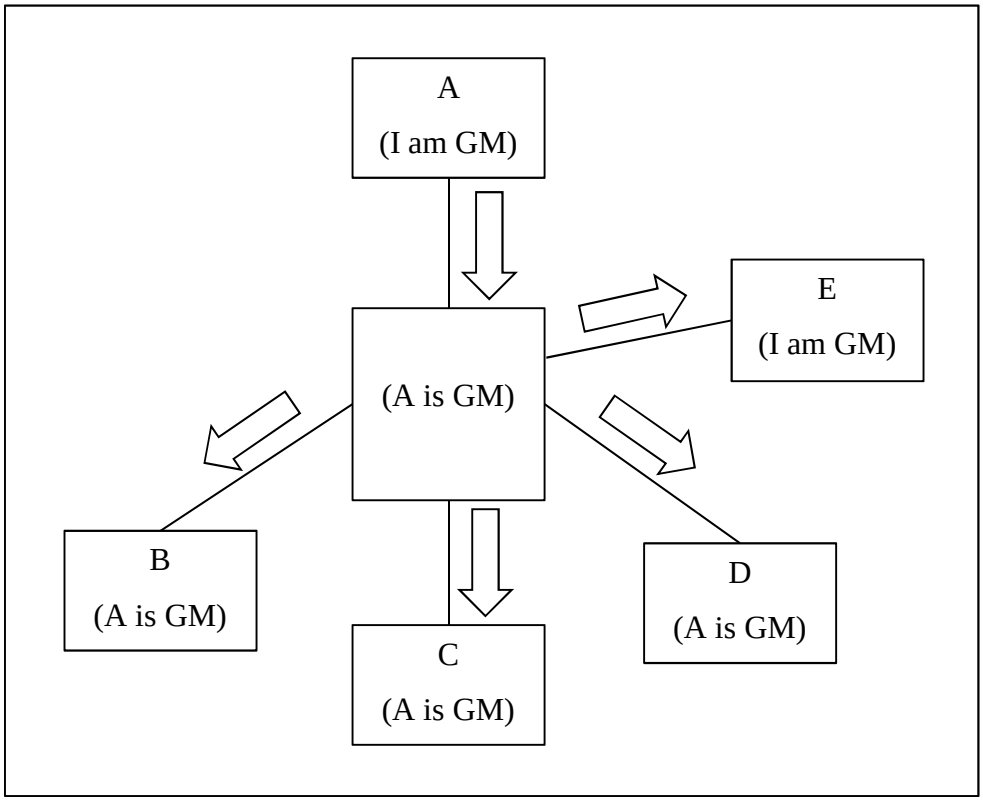

Figure 67: *Addition* of a time-aware system

As shown in Figure 68, time-aware end station E transmits an *announce* message announcing its superiority. The time-aware bridge updates its knowledge of the grandmaster. Subsequent announce messages from time-aware system A are not forwarded by the time-aware bridge as is it aware of a superior time-aware system (time-aware end station E). The bridge forwards time-aware end station E's *announce* message.

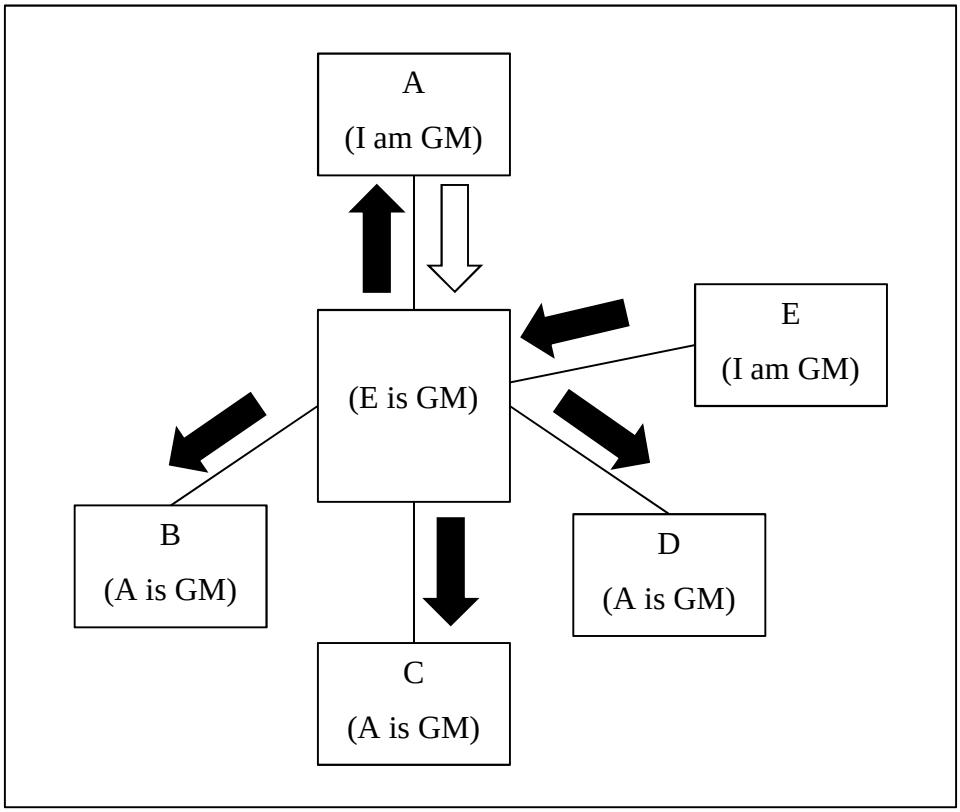

Figure 68: *Announce* message transmission after time-aware system addition

Once each time-aware system has received the *announce* message from time-aware end station E, it records that time-aware system E is the grandmaster, as shown in Figure 69.

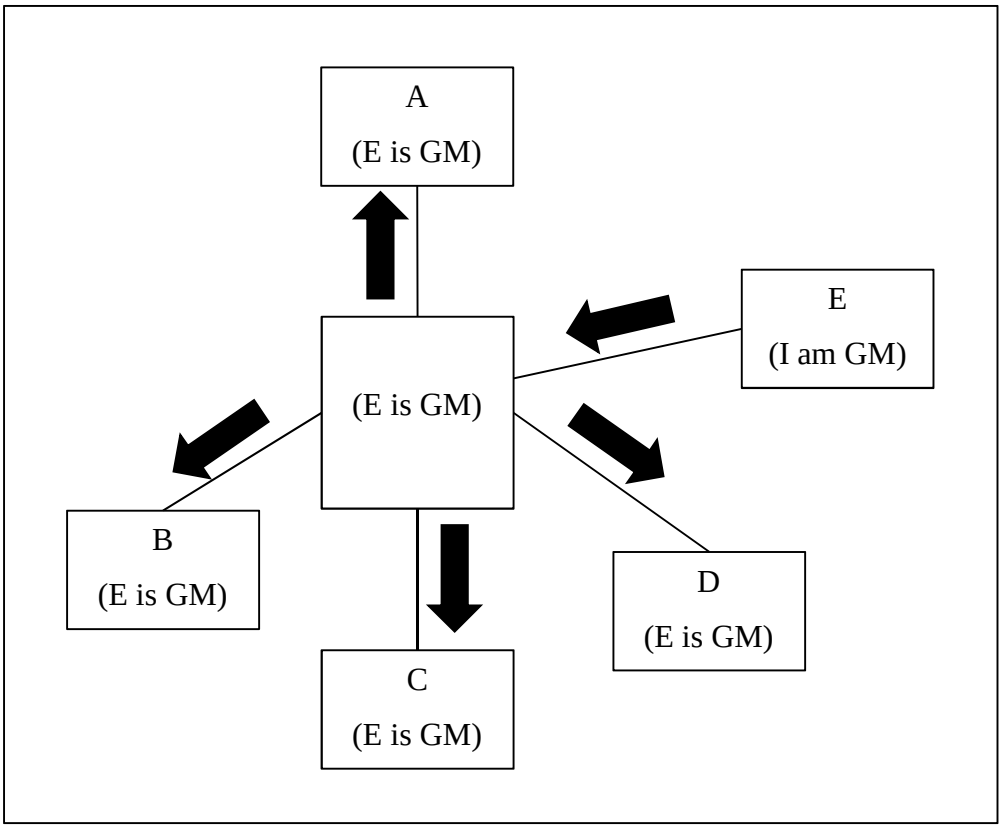

Figure 69: *Announce* message transmission in a steady state after time-aware system addition

If time-aware system E is removed from the time-aware bridged LAN, the other time-aware systems will stop receiving its *announce* messages. After a time-out period, all of the time-aware systems send *announce* messages, as shown in Figure 70. The time-aware bridge keeps track of the *system identity* with the lowest value and only forwards the *announce* message of the time-aware system with the lowest *system identity* value. In this case (assuming that the bridge processes time-aware system A's *announce* message first), time-aware system A's *announce* message is forwarded to the other time-aware systems and the rest of the other time-aware systems *announce* messages are filtered.

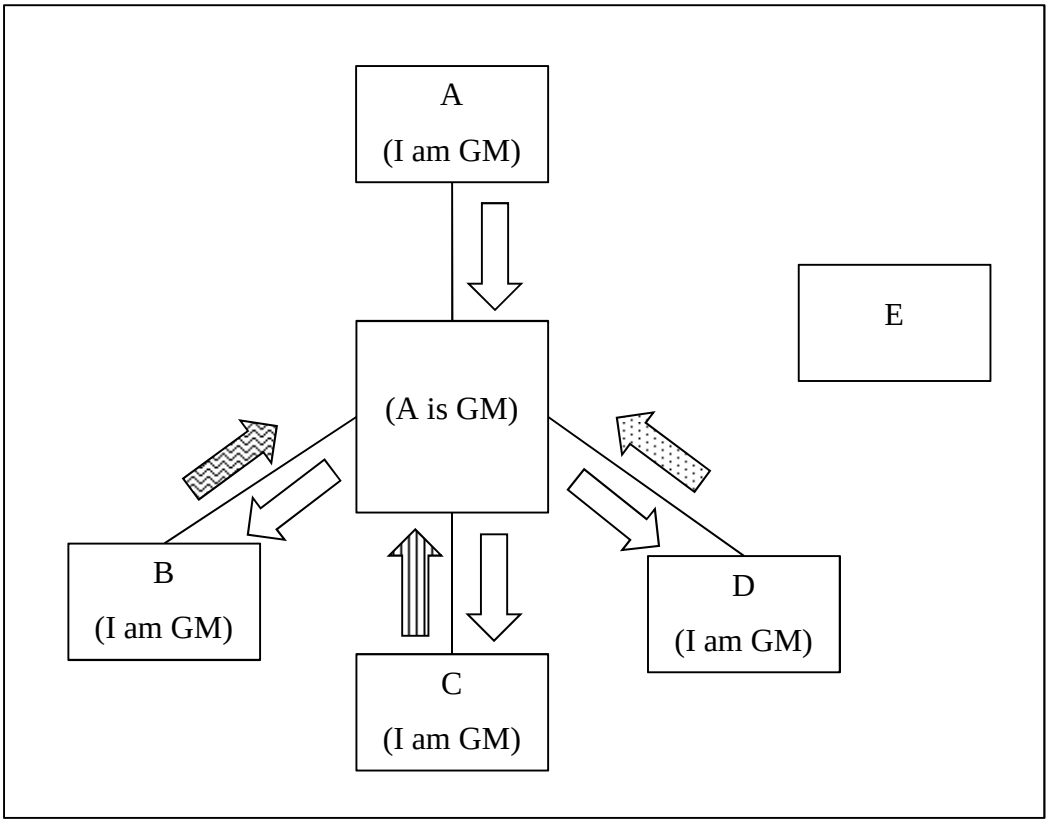

Figure 70: *Announce* message transmission after time-aware system removal

The system stabilises once again, and the grandmaster time-aware system continuously transmits *announce* messages announcing its presence and superiority, as shown in Figure 71.

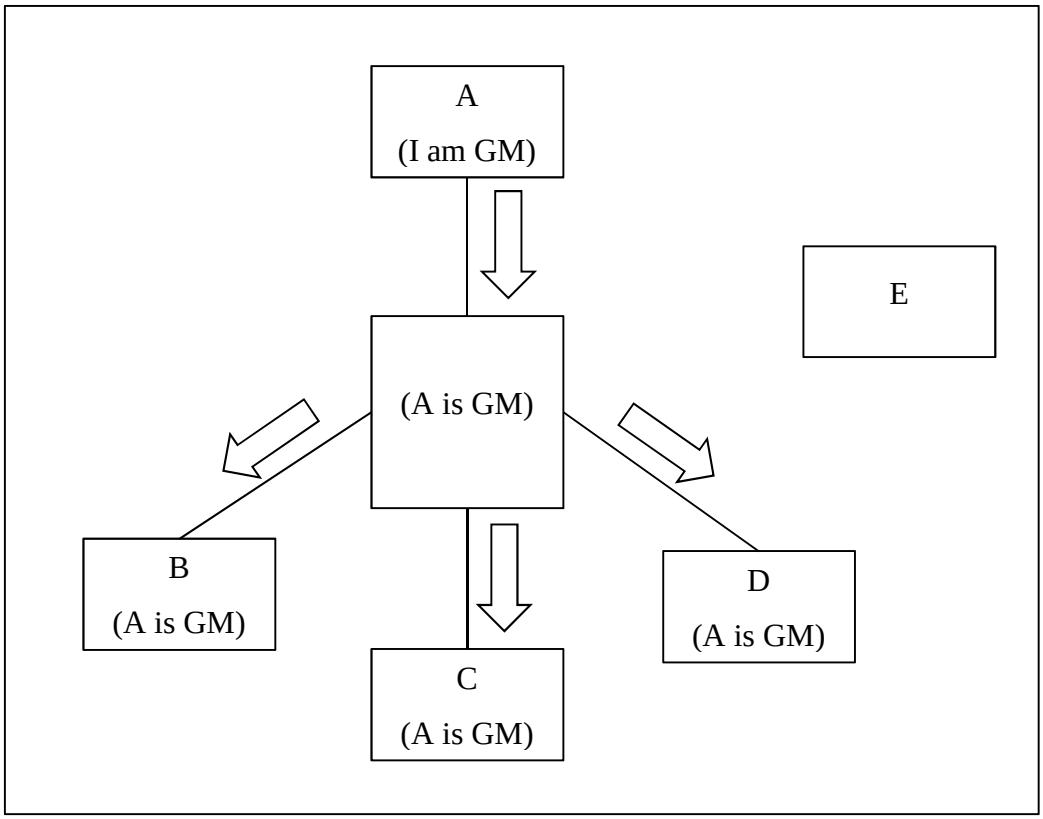

Figure 71: *Announce* message transmission in a steady state after station removal

5.2.2.3 Port Roles

As part of the construction of a time-synchronisation spanning tree, each port of each time-aware system is assigned a *port role* from Table 12.

Port role	Description
Master	Any port of a time-aware system that is closer to the root time-aware system than any other port of the gPTP communication path connected to that port.
Slave	The one port of a time-aware system that is closest to the root time-aware system.
Passive	Any port of a time-aware system whose port role is not *master*, *slave*, or *disabled*.
Disabled	Any port of a time-aware system that is either disabled, has its time-synchronisation and best master selection functions disabled, or the IEEE 802.1AS protocol is not operating.

Table 12: Port role definitions

An example master/slave hierarchy of time-aware systems is shown in Figure 72. All of the grandmaster's ports have a port role of *master*. All of the other time-aware systems have exactly one slave port. The time synchronisation spanning tree is formed by the links between ports of time-aware systems where these ports are not *passive*. The *passive* ports are used to break loops.

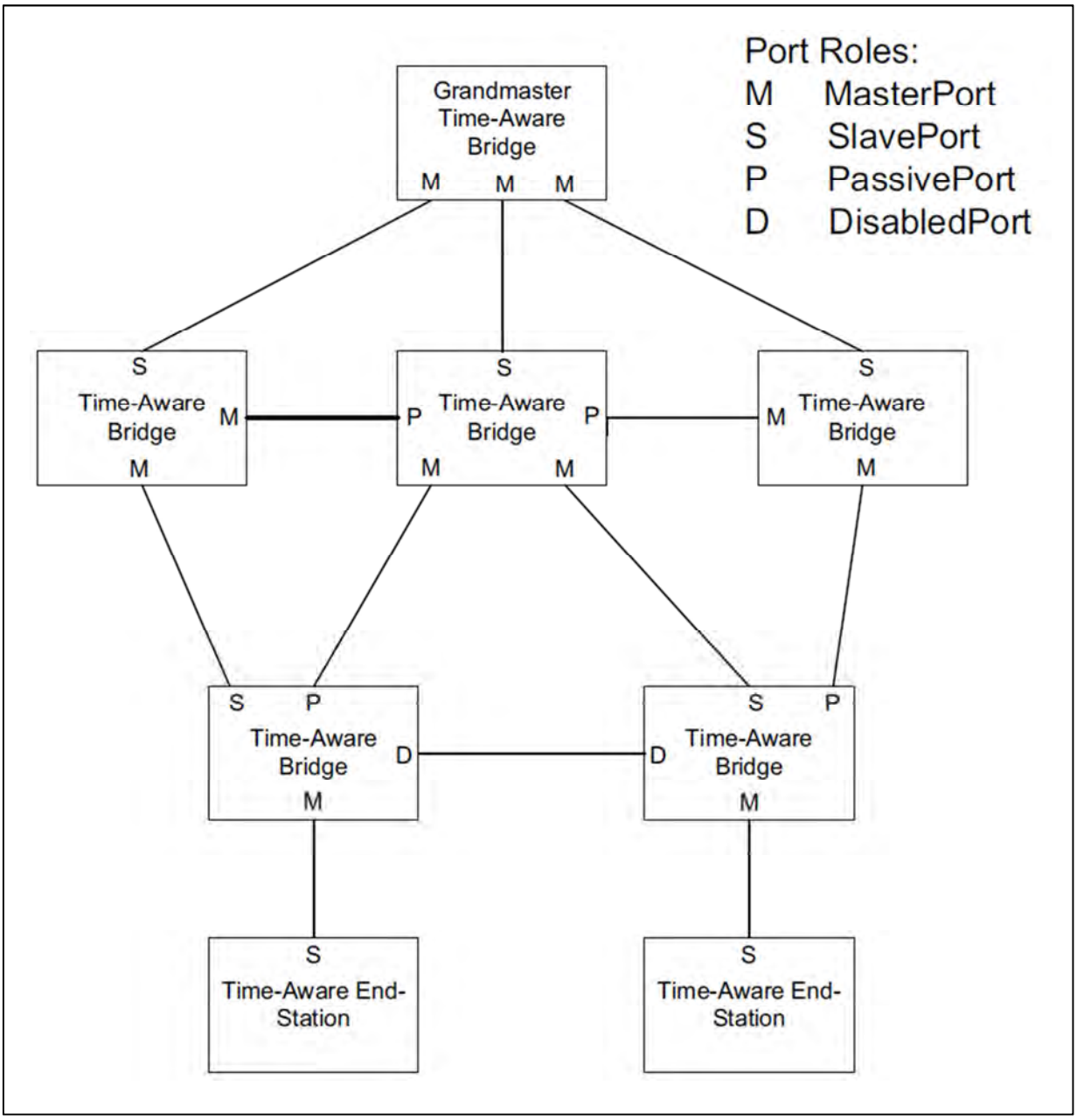

Figure 72: An example master/slave hierarchy of time-aware systems [69]

5.2.3 Logical Syntonisation

Time synchronisation correction is dependent on the accuracy of delay and residence time measurements. Ideally, the clock used for this purpose has to be frequency locked (syntonised) to the grandmaster to ensure that all of the time measurement intervals use the same time base. Two time-aware systems are syntonised if the duration of a second on both of the clocks is the same. i.e., each clock advances at the same rate (syntonised clocks do not necessarily share the same epoch).

Adjusting the frequency of an oscillator is slow and prone to gain peaking effects. gPTP measures the frequency of a clock against the frequency of the grandmaster clock. This frequency ratio is known as the *grandmaster frequency ratio*. The *grandmaster frequency ratio* allows time-aware bridges to correct time interval measurements. For each port that a time-aware system has, the ratio of the frequency of the time-aware system at the other end of the link to the frequency of its own clock is measured. This is known as the *neighbour frequency ratio*. The *grandmaster frequency ratio* is used in computing synchronised time, and the *neighbour frequency ratio* is used in correcting propagation time measurements. The *grandmaster frequency ratio* is calculated by accumulating *neighbour frequency ratios*.

All of the time-aware systems in a gPTP domain are logically syntonised, meaning that they all measure time intervals using the same frequency even though their clocks may advance at different rates.

5.2.4 PTP Peer Delay Protocol

Link propagation delay is measured by each port at the end of every full-duplex, point-to-point link. Both of the ports that share a link, independently and periodically make the measurement, and both ports know the propagation delay result. This allows time-synchronisation information to be transported irrespective of the direction it takes. The direction that time synchronisation information is transmitted could change if the grandmaster changes.

Figure 73 shows the operation of the *PTP peer delay protocol*. The propagation delay measurement is initiated by a *peer delay initiator* time-aware system. The time-aware system at the other end of the link is the *peer delay responder* time-aware system. A similar measurement occurs in the opposite direction, with the initiator and responder interchanged and the directions of the messages reversed.

The propagation delay measurement starts with the *peer delay initiator* transmitting a *p delay request event message* and generating a timestamp, t_1. The *peer delay responder* receives the message and timestamps it with time t_2. The responder returns a *p delay response event message* and timestamps it with time t_3. The responder returns the time t_2 in the *p delay response* message, and the time t_3 in a *p delay response follow up* message. The *peer delay initiator* generates a timestamp, t_4, upon receiving

the *p delay response* message. The *peer delay initiator* then uses these four timestamps to calculate the *mean propagation delay* as follows:

$$t_{ir} = t_2 - t_1$$

$$t_{ri} = t_4 - t_3$$

$$D = \frac{t_{ir} + t_{ri}}{2} = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$$

where *D* is the *mean propagation delay*.

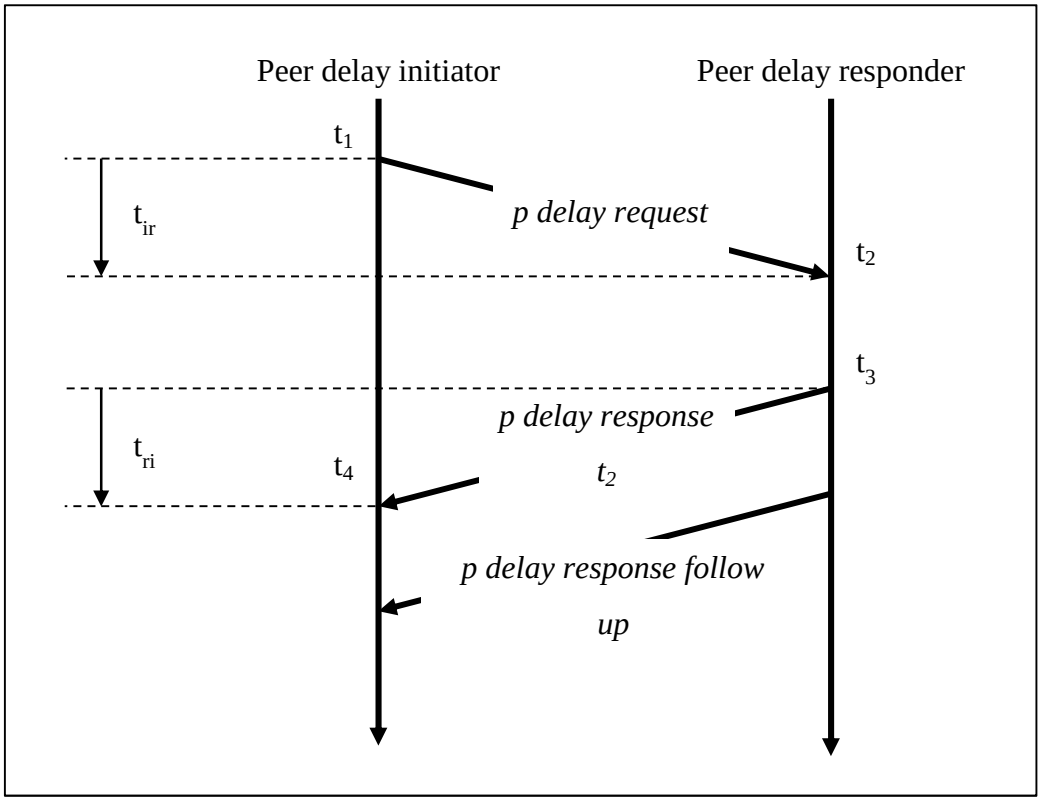

Figure 73: Propagation delay measurement using the *PTP peer delay protocol* (adapted from [69])

The accuracy of the *mean propagation delay* measurement depends on how accurately the times t_1, t_2, t_3, and t_4 are measured. The calculation also assumes that the *peer delay initiator* and *peer delay responder* timestamps are taken relative to clocks that have the same frequency. In practice, t_1 and t_4 are measured relative to the local clock of the *peer delay initiator*, and t_2 and t_3 are measured relative to the local clock of the *peer delay responder*. If the value of the *mean propagation delay* is wanted relative to the *peer delay responder's* time base, the term ($t_4 - t_1$) must be multiplied by the ratio of the *peer delay responder* relative to the *peer delay initiator*. If the value of the *mean propagation delay* is wanted relative to the *peer delay initiator's* time base, the term ($t_3 - t_2$) must be multiplied

by the rate ratio of the *peer delay initiator* relative to the *peer delay responder*. If the *mean propagation delay* is desired relative to the grandmaster's time base, each term must be multiplied by the rate ratio of the grandmaster relative to the time base that the term is expressed in.

There can also be an error in the measured propagation delay due to time measurement granularity. For example, if the time measurement granularity is 40ns, the timestamps t_1, t_2, t_3, and/or t_4 can undergo 40ns step changes. When this occurs, the measured propagation delay will change by 20ns (or by a multiple of 20ns if more than one of the timestamps has undergone a 40ns step change). The actual propagation delay has not changed by 20ns. The effect is due to time measurement granularity. The effect can be reduced, and the accuracy improved, by averaging successive measured propagation delay values.

The *p delay turnaround time* (the duration of the interval between the receipt of a *p delay request* message by a port, and the sending of the corresponding *p delay response* message) must be less than or equal to 10ms.

5.2.5 Calculating Neighbour Rate Ratio

The rate ratio of the *peer delay responder* relative to the *peer delay initiator* is known as the *neighbour rate ratio.* The *neighbour rate ratio* is calculated using the departure (from a *peer delay responder*) and arrival (to a *peer delay initiator*) times of successive *p delay response* messages. IEEE 802.1AS does not prescribe the specific algorithm to use to calculate the *neighbour rate ratio*, as long at the measurement can be made within ±0.1 ppm. As an example, the *neighbour rate ratio* can be estimated as the ratio of the elapsed time of the local clock of the responder time-aware system, to the elapse time of the local clock of the initiator time-aware system. This ratio can be calculated for the time interval between a set of received *p delay response* and *p delay response follow up* messages and a second set of received *p delay response* and *p delay response follow up* messages some number of *p delay request* message transmission intervals later.

Figure 74 shows two time-aware systems (a *peer delay initiator* and a *peer delay responder*), and it shows a *p delay response* and *p delay response follow up* message being transmitted by the *peer delay responder*, and then a second *p delay response* and *p delay response follow up* being transmitted a number of *p delay request* message transmission intervals later. Based on the departure

time of the first *p delay response* message ($t_{3,1}$) and the departure time of the second *p delay response* message ($t_{3,x}$), the time difference between these two times (td_r) can be calculated by the *peer delay initiator*. Based on the arrival times of the first *p delay response* message ($t_{4,1}$) and the arrival time of the second *p delay response* message ($t_{4,x}$), the time difference between these two times (td_i) can be calculated by the peer delay initiator. These two time differences can then be used to calculate the *neighbour rate ratio*.

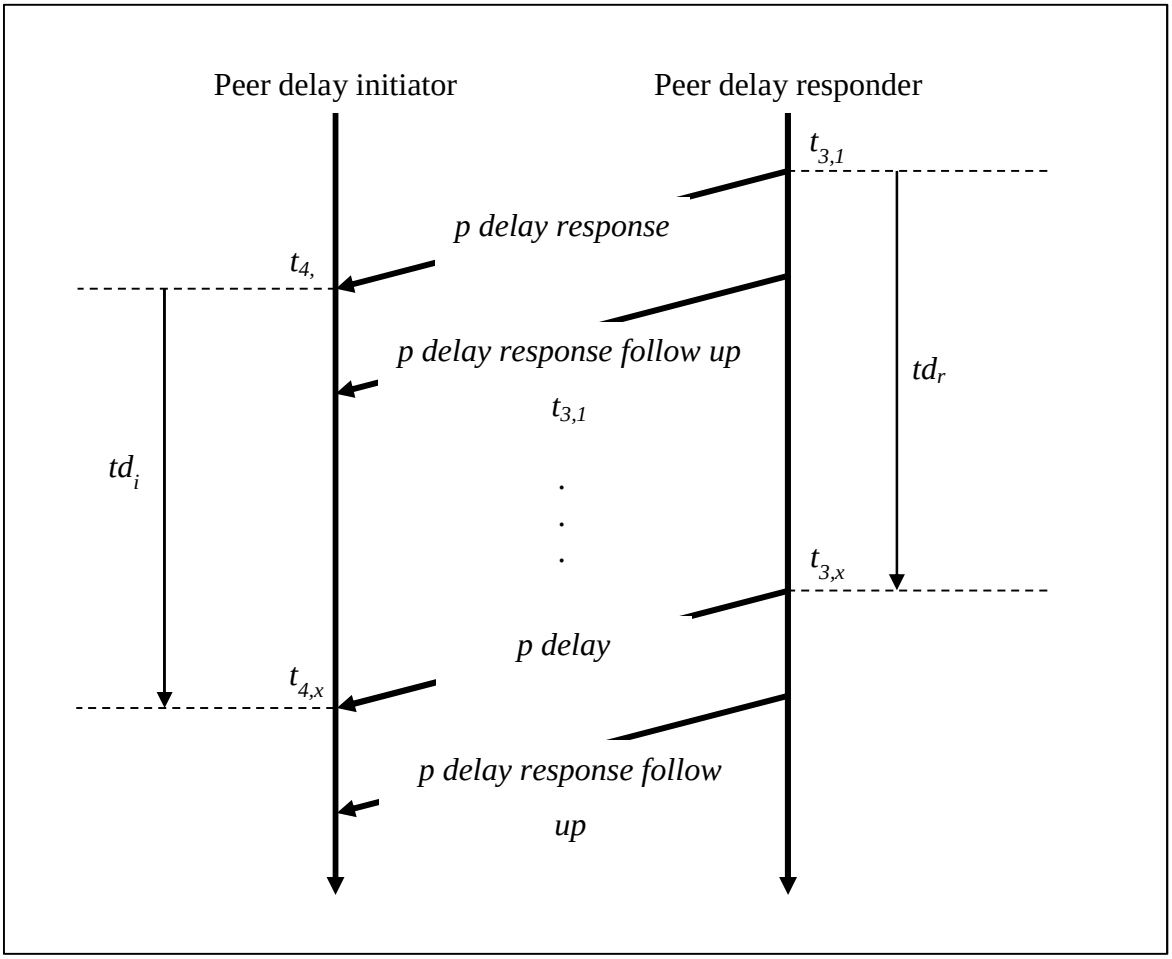

Figure 74: Example *neighbour rate ratio* calculation

5.2.6 Time-Synchronisation

In a time-aware bridged LAN, a time-aware system synchronises to the selected grandmaster using information that it receives on its slave port. In this context, *synchronise* means that the time-aware system is able to calculate the grandmaster time corresponding to any local clock time. There is no

requirement that the local clock's frequency be physically adjusted to match the grandmaster's frequency, although this is not prohibited.

The transportation of time-synchronisation information by a time-aware system that uses full-duplex Ethernet links use *sync* and *follow up event messages*. As shown in Figure 75, all master ports periodically transmit a *sync* message to ensure that all devices share a common sense of time. This time originates from the grandmaster time-aware system, and trickles through to the leaves of the time-synchronisation spanning tree.

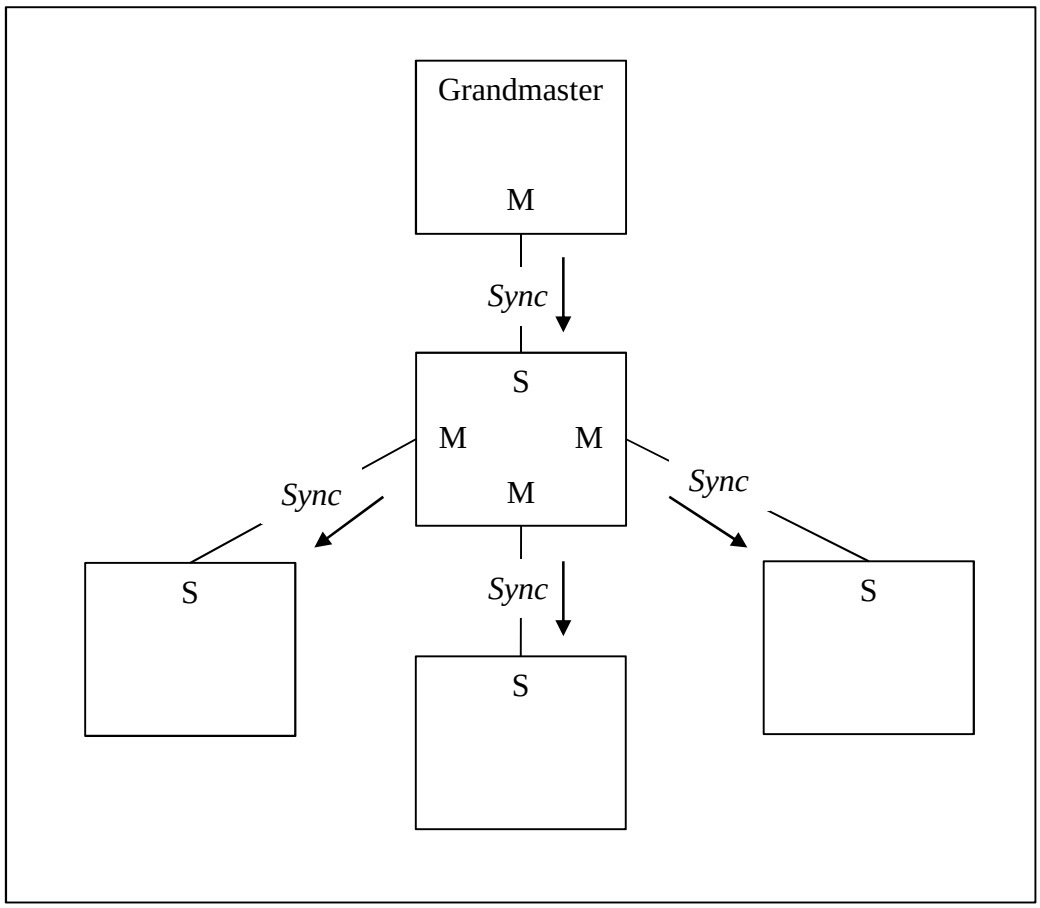

Figure 75: Transmission of *sync* messages

A non-grandmaster time-aware system receives time synchronisation information on its slave port. This information consists of the grandmaster's time, and the corresponding local clock's time. The synchronisation process is shown in Figure 76. This figure shows three adjacent time-aware systems, labelled *a*, *b*, and *c*. Time-synchronisation information is transported from time aware system *a* to time-aware system *b*, and from time-aware system *b* to time-aware system *c*.

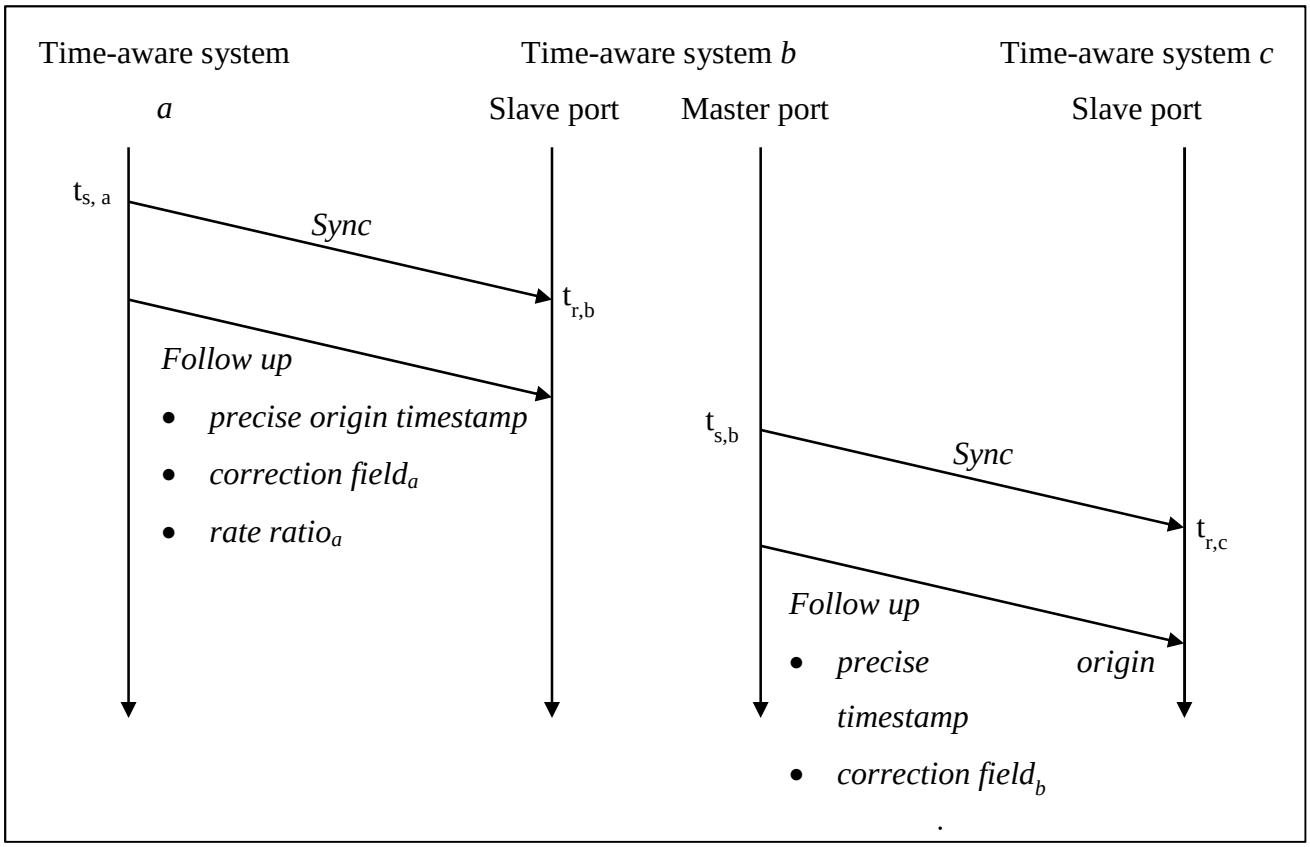

Figure 76: Transport of time synchronisation information (adapted from [69])

In Figure 76, the master port of time-aware system *a* sends a *sync* message to the slave port of time-aware system *b* at time $t_{s,a}$. $t_{s,a}$ is measured relative to the local clock of time-aware system *a*. At a later time, time-aware system *a* sends an associated *follow up* message to time-aware system *b*, which contains the following fields:

- *precise origin timestamp*: this field contains the time of the grandmaster when it originally sent the *sync* message information. If time-aware system *a* is the grandmaster, this would correspond to the time $t_{s,a}$.
- *correction field*$_a$: this field contains the difference between the grandmaster's time when the *sync* message was sent by time-aware system *a* ($t_{s,a}$) and the *precise origin timestamp*. If time-aware system *a* is the grandmaster, this value would be zero. The sum of the *precise origin timestamp* and the *correction field*$_a$ values is equal to the grandmaster's at time $t_{s,a}$.
- *rate ratio*$_a$: this field contains the ratio of the grandmaster's clock frequency to the frequency of the local clock of time-aware system *a*. It is calculated from the most recently received *rate ratio* on its slave port multiplied by the current *neighbour rate ratio* measured by the slave port. If

time-aware system *a* is the grandmaster (and hence does not have any slave ports), the *rate ratio* is one.

Time-aware system *b* receives the *sync* message from time-aware system *a* at time $t_{r,b}$. $t_{r,b}$ is measured relative to the local clock of time-aware system *b*. Time-aware system *b* receives the associated *follow up* message some time later.

Time-aware system *b* sends send a new *sync* message at time $t_{s,b}$. $t_{s,b}$ is measured relative to the local clock of time-aware system *b*. Time-aware system *b* calculates the value of the *correction field*$_b$ (the difference between the grandmaster's time that corresponds to the time $t_{s,b}$ and the *precise origin timestamp*). i.e., the sum of the *precise origin timestamp* and the *correction field*$_b$ values is equal to the grandmaster's time that corresponds to $t_{s,b}$ (the time that the *sync* message was sent by time-aware system *b*). To calculate the *correction field*$_b$, it must calculate the value of the time interval between $t_{s,a}$ and $t_{s,b}$, expressed in the grandmaster's time base. This interval is equal to the sum of the following quantities:

- The propagation delay on the link between time-aware systems *a* and *b*, expressed in the grandmaster time base.
- The difference between $t_{s,b}$ and $t_{r,b}$ (i.e., the residence time), expressed in the grandmaster time base.

The *mean propagation delay* (which time-aware system *b* stores in time-aware system *a*'s time-base) on the link between time-aware system *a* and time-aware system *b* must be multiplied by *rate ratio*$_a$ to express it in the grandmaster's time base. The residence time must be multiplied by the *rate ratio*$_b$ to express it in the grandmaster time base. These adjustments allow for all quantities used in calculations to use the same time base thus reducing inaccuracies in calculations.

With the time information that is transported in the *sync* and *follow up* messages, it is possible for a time-aware system to calculate the offset between itself and the grandmaster. Thus, it is possible for a time-aware system to calculate the grandmaster's time (the synchronised time) at any given point using its local time and the calculated offset.

5.3 Conclusion

When transmitting audio and video streams across IEEE 1394 and Ethernet AVB networks, it is necessary for these streams to be synchronised. There needs to be mechanisms in place that allow audio and video data to be time-stamped with presentation times so that this data can be presented correctly with respect to each other. These devices also need mechanisms to allow them to perform wordclock synchronisation. This requires that all of the devices on a network share a common sense of time.

This chapter discussed how IEEE 1394 and Ethernet AVB networks natively allow for all devices to synchronise their clocks to that of a master clock, thus allowing for the sharing of a common time. These networking technologies have distinct methods for achieving clock synchronisation and represent time in different formats. Chapter 6 “Media Transport Protocols” discusses how this time is used by media transport protocols to provide synchronisation. When time-stamped data streams are transferred from IEEE 1394 networks to Ethernet AVB networks, or vice versa, there is a need to understand these synchronisation mechanisms to ensure the correct presentation of the media.

Chapter 6 Media Transport Protocols

A number of media transport protocols have been standardised for transporting various forms of media over IEEE 1394 and Ethernet AVB networks. An IEEE 1394/Ethernet AVB audio gateway device needs to be able to interpret these protocols to allow for reception of streams on one network type, and for the subsequent transmission of these streams on the other network type. Audio data and timing information has to be successfully transferred across the networking technologies. This chapter presents an overview of these protocols, the packet formats that they utilise, and their associated synchronisation mechanisms.

6.1 Packet Formats

Packet formats have been standardised for transporting audio data over IEEE 1394 and AVB networks. The AVB protocols borrow heavily from the IEEE 1394 protocols, and thus the IEEE 1394 protocols are discussed first.

6.1.1 IEEE 1394 Audio Packet Formats

6.1.1.1 Isochronous Packet

Figure 77 shows the IEEE 1394 isochronous packet format used during isochronous transactions. Each node that has successfully reserved bandwidth and a channel number, and that is performing isochronous transactions, is able to transmit one of these packets onto an IEEE 1394 bus (for each channel of interest) once each isochronous cycle (once every 125 µs).

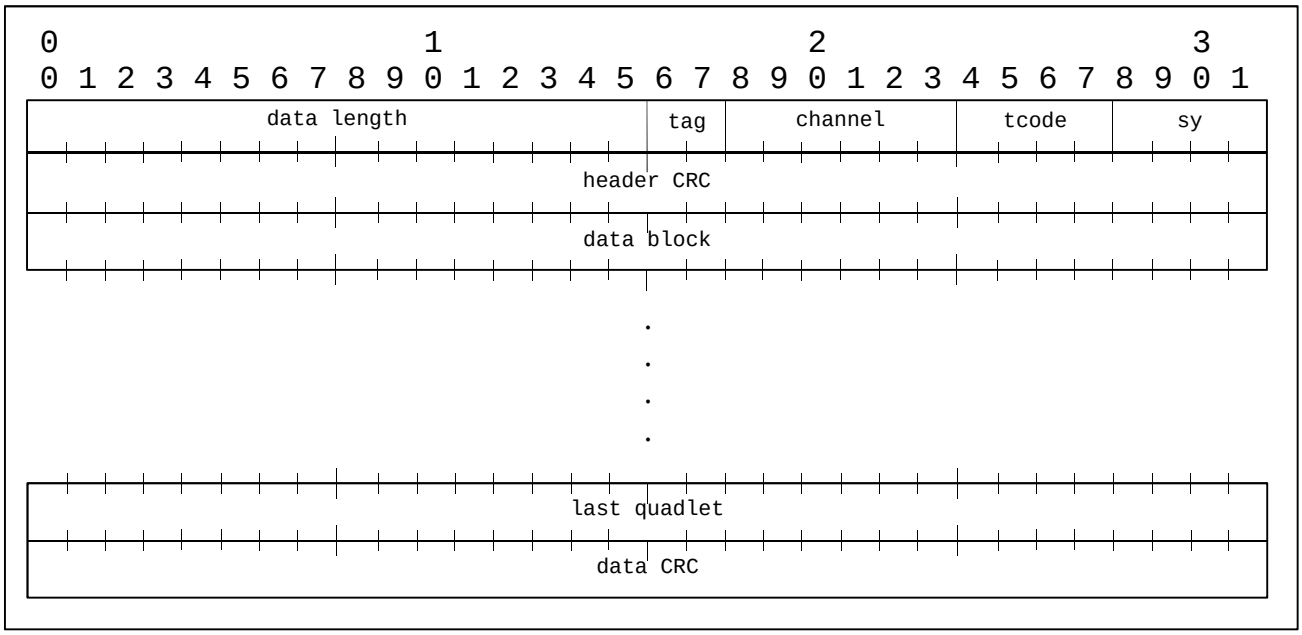

Figure 77: Isochronous packet format

Table 13 defines the fields of an isochronous packet.

Field	Size	Definition
data length	16 bits	The value of the *data length* field represents the number of bytes of data in the packet.
tag	2 bits	The *tag* field is used to identify higher level protocols carried within the isochronous packet.
channel	6 bits	The value of the *channel* field is used to represent the channel that the packet is transmitted on.
tcode	4 bits	The *tcode* field is set to a value of 0xA for isochronous transactions.
sy	4 bits	The *sy* field is application specific.
data blocks		The *data blocks* carry the isochronous packets payload.

Table 13: Isochronous packet fields

6.1.1.2 Common Isochronous Packet

The IEC [74] has prepared a series of standards that prescribe the transmission of audio, video and multimedia over IEEE 1394 networks. This series is identified as *IEC 61883*. IEC 61883 is composed of a number of parts that give formal descriptions of how audio and video data should be

transported across IEEE 1394 networks. Amongst other things, IEC 61883-1 [75] describes the general packet format, known as the *common isochronous packet* (CIP). A packet that uses this format is known as a *CIP packet*. Figure 78 shows the format of a CIP packet. A CIP header and data section appear within the data section of an isochronous packet. For an isochronous packet that contains CIP data, its *tag* field is set to a value of 01_b.

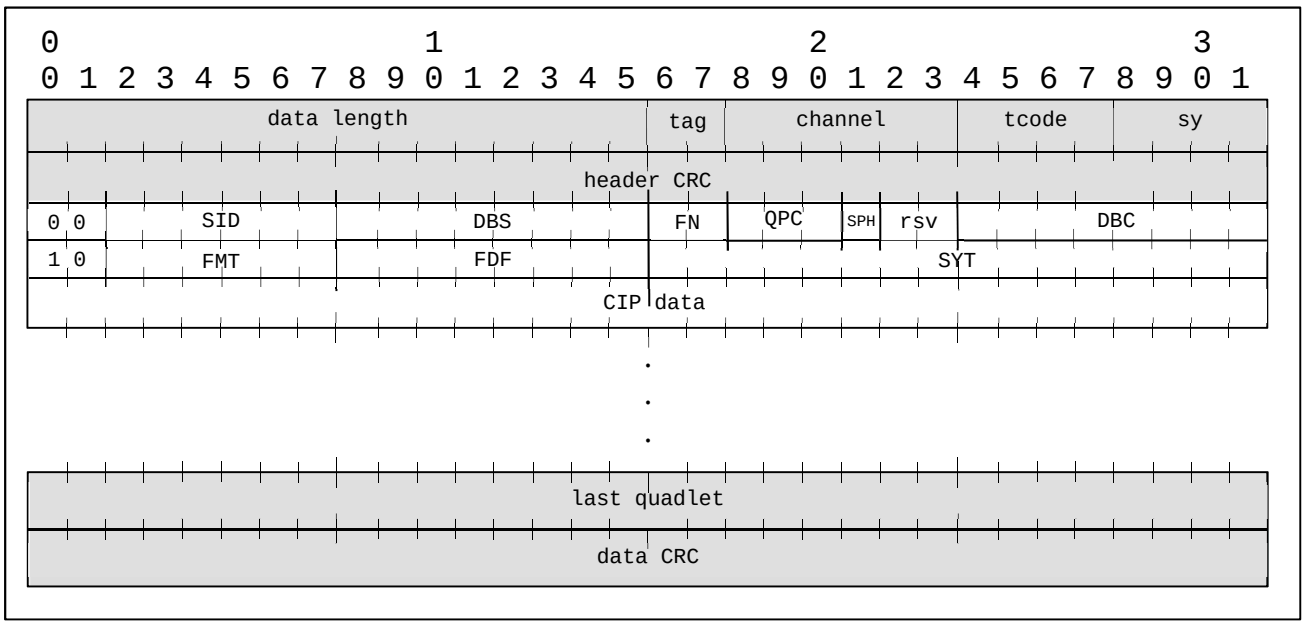

Figure 78: CIP packet format

The CIP header fields are defined in Table 14.

Field	Size	Definition
SID	6 bits	The *SID* field's value represents the 6 bit physical ID of the node transmitting the packet.
DBS	8 bits	Audio visual data is clustered into one or more data blocks. The *data block size* field represents the size of the data blocks. A value of 0 indicates 256 quadlets, and any other value represents the actual number of quadlets.
FN	2 bits	The *fraction number* field is used to indicate the number of fractions into which the original application packet was divided. This is not used for audio samples, so its value is 0.
QPC	3 bits	The *quadlet padding count* field is used to represent the number of padding bytes added to each application frame to make it divide evenly into 2^{FN} data blocks. The value of this field is 0 for audio data.
SPH	1 bit	The *source packet header* field is set to 0 for audio data indicating that the packet contains no sources packets.
rsv	2 bits	This field is reserved for future standardisation.
DBC	8 bits	The *data block count* field represents a count of data blocks.
FMT	6 bits	The *format* field is used to indicate the format of the data being transported.
FDF	8 bits	The *format dependent field* is used to identify sub-formats or convey other information.
SYT	16 bits	The value of the SYT field indicates the time that a particular data block within the packet should be presented at the receiver.

Table 14: CIP header fields

6.1.1.3 IEC 61883-6

IEC 61883-6 [53] is a standard that defines the *audio and music data transmission protocol*. This defines how audio data is transferred across IEEE 1394 networks. IEC 61883-6 allows multiple channels (or *sequences*) of audio to be transmitted simultaneously within single CIP packets. Conceptually, these sequences of audio are transferred in *streams*. Figure 79 shows this concept diagrammatically. It shows a stream that is transferring four sequences of audio. These audio sequences could correspond to four channels of audio being transmitted by an audio mixing desk, for example.

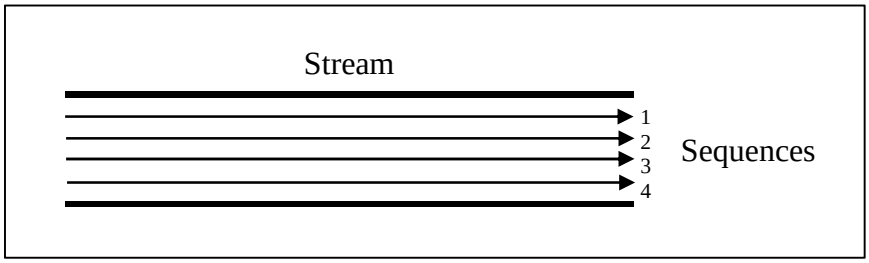

Figure 79: A stream of sequences

Audio data is transported within a CIP packet's data section. As an IEEE 1394 transmitter transmits isochronous packets at a rate of 8000 per second, and as audio is sampled at much higher rates (44.1 kHz, 48 kHz, 88.2 kHz, or 96 kHz, for example), multiple audio samples have to be packaged together into single CIP packets.

The simultaneous availability of samples from multiple sources is known as an *event*. The data that makes up an event is known as a *data block.* A CIP packet would format the audio stream in Figure 79 into data blocks each containing four quadlets (a quadlet is four octets). Each data block represents the simultaneous sampling of the four channels of audio (known as an event), and each quadlet represents a sample. Figure 80 shows an isochronous packet and how samples are packaged into data blocks within the packet. The quadlet position within a data block corresponds to a sequence. For example, each first quadlet of each data block is a sample of a series of samples that together make up a sequence of audio. A series of these packets together make up a stream. The size of each data block is communicated in the CIP header's *DBS* field. In this example, the value would be four quadlets.

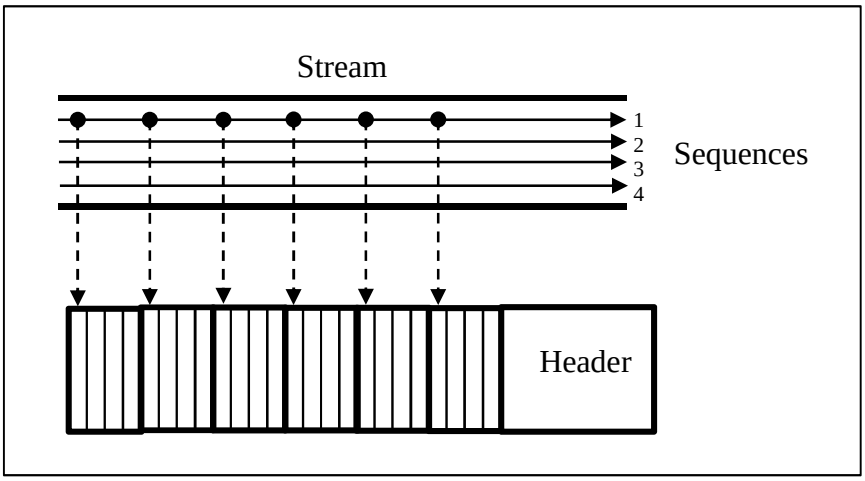

Figure 80: A representation of a stream containing sequences

CIP packets transferring IEC 61883-6 conformant data have an *FMT* field value of 0x10 (audio and music data transmission protocol). IEC 61883-6 defines a number of sub formats. Within CIP packets, this sub format is carried within the *FDF* field. The format of the *FDF* field is shown in Figure 81.

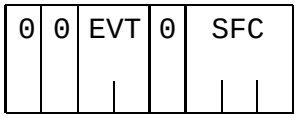

Figure 81: *FDF* field

The *EVT* field of the *FDF* field indicates the event type. Possible values for the *EVT* field for IEC 61883-6 are shown Table 15. The work in this thesis focused on the use of the *AM824* event type.

EVT (event type)	
Value	**Description**
0x00	AM824
0x01	24-bit * 4 audio pack
0x02	32-bit floating point data
0x03	Reserved

Table 15: Event type (EVT) code definitions

The *SFC* field of the *FDF* field indicates the *nominal sampling frequency code*. Possible values for the *SFC* field are shown in Table 16.

SFC (Nominal Sampling Frequency)	
Value	**Sample transmission frequency**
0x00	32 kHz
0x01	44.1 kHz
0x02	48 kHz or not indicated
0x03	88.2 kHz
0x04	96 kHz or not indicated
0x05	176.4 kHz
0x06	192 kHz or not indicated
0x07	Reserved

Table 16: SFC (Nominal Sampling Frequency Code) definition

Each sample within a data block that is formatted with the *AM824* format specification is composed of an 8-bit label, and 24 bits of audio data, as shown in Figure 82.

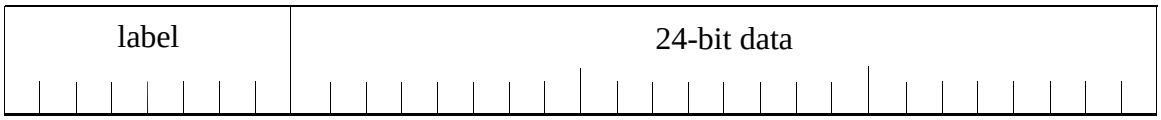

Figure 82: AM824 format

The value of the *label* field is used to specify the format of the *24 bit data* field. Table 17 shows the possible values, and their definitions. The work described in this thesis focused on the use of the *raw audio* format.

Label	
Value	**Description**
0x00 – 0x3F	IEC 60958 conformant
0x40 – 0x43	Raw audio
0x44 – 0x7F	Reserved
0x80 – 0x83	MIDI Conformant
0x84 – 0xFF	Reserved

Table 17: AM824 label definitions

Each sample using the *raw audio* format may have a sample length that is 24, 20, or 16 bits in length. The *label* values used to represent these are shown in Table 18.

Raw audio labels	
Value	**Description**
0x40	Raw audio 24-bit sample
0x41	Raw audio 20-bit sample
0x42	Raw audio 16-bit sample
0x43	Reserved

Table 18: *Raw audio* labels

Raw audio samples are expressed using 2's complement numbering format.

6.1.2 Ethernet AVB Audio Frame Formats

IEEE 1722 [55] specifies a protocol, packet formats, and presentation time procedures that can be used to transfer audio and video data between end stations on AVB bridged LANs. The protocol leverages protocols and procedures defined in the IEC 61883 family of standards. The protocol defined in IEEE 1722 is known as the *audio/video transport protocol* (AVTP). The protocol allows audio and video data that is transmitted by a talker to be reproduced on one or more listeners. AVTP data is transported directly on the underlying MAC layer. This section will only focus on the use of Ethernet as the underlying medium.

All devices that transmit, receive or forward AVTP data are required to support MSRP (see Section 3.2.4 “Multiple Stream Reservation Protocol” on page 60), the forwarding and queuing rules specified in Section 4.2.2 “Forwarding and Queuing” on page 87, and gPTP (see Section 5.2 “Ethernet AVB” on page 116). AVTP relies on these protocols in order to function properly.

MSRP is used to communicate AVTP stream resource requirements to a bridged LAN and to reserve resources for the streams. The forwarding and queuing rules ensure that stream frames are transported throughout a bridged LAN with the requested QoS. AVTP makes use of gPTP for a common sense of time to be used to convey timing information from talkers to listeners.

6.1.2.1 AVTP Frame Formats

AVTP is capable of transporting many audio and video formats. AVTP defines a common header that is common to all AVTP frames, and it defines headers that are specific to audio and video formats. An AVTP frame is contained within an Ethernet frame with the EtherType 0x22F0.

6.1.2.1.1 AVTP Common Header

All AVTP stream and control frames share a common header, as shown in Figure 83.

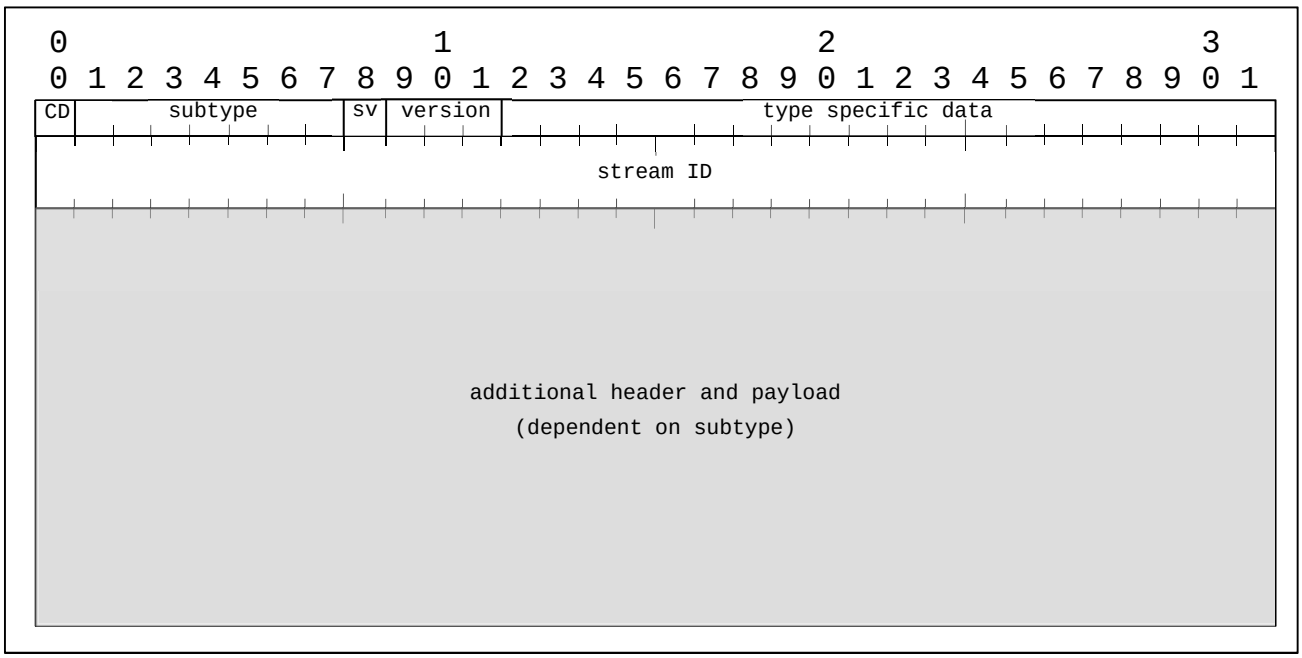

Figure 83: AVTP common header

The fields of this header are defined in Table 19.

Field	Size	Definition
CD	1 bit	The value of the *CD* field indicates whether the AVTP frame is carrying control or stream data. A value of 0 indicates stream data, whereas a value of 1 indicates control data.
subtype	7 bits	The value of the *subtype* field is used to identify the protocol being carried by the AVTP frame.
sv	1 bit	The value of the *sv* field indicates whether the stream ID field contains a valid stream ID or not. A value of 1 indicates that the stream ID field is carrying a valid stream ID, and a value of 0 indicates that the value is not valid.
version	3 bits	The value of the *version* field is used to identify the version of AVTP being transported. Currently, the only valid value is 0x00. All other values are reserved for future standardisation.
type specific data	20 bits	The value of the *type specific data* field is dependent on whether the AVTP is transporting control data or stream data.
stream ID	64 bits	If the *sv* field is set to 1, then the *stream ID* field contains a valid AVB stream ID. The stream ID field is used to identify the stream that the frame is associated with.

Table 19: AVTP common header field definitions

The possible *subtype* values are shown in Table 20. The *experimental* subtype is for testing purposes only. The subsequent interpretation of AVTP frame fields is dependent on the subtype conveyed in the frame. This work focused only on the use of the 61883 *subtype* (0x00) and the MAAP *subtype* (0x7E).

Value	Protocol	Definition
0x00	61883/IIDC	IEC 61883/IIDC protocol
0x01	MMA	Midi Manufactures Association
0x02 – 0x7D	*Reserved*	Reserved for future use
0x7E	MAAP	Multicast Address Acquisition Protocol
0x7F	*Experimental*	Experimental protocol

Table 20: Subtype values

6.1.2.1.2 AVTP Common Stream Data Header

When the *CD* field of an AVTP common header is 0 (indicating that AVTP is transporting stream data), AVTP uses an *AVTP common stream header* as detailed in Figure 84. An AVTP common stream header specifies fields in the *type specific data* field, and includes additional fields below the *stream ID* field.

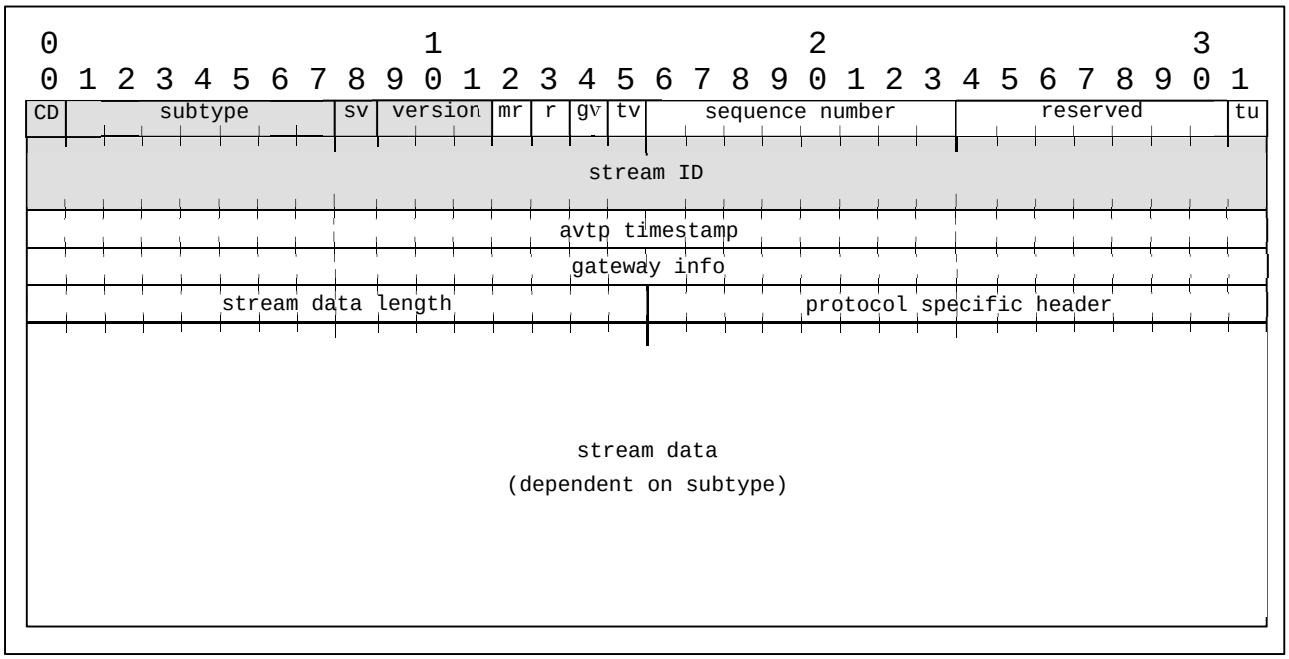

Figure 84: AVTP common stream header

The *type specific data* fields are as specified in Table 21.

Field	Size	Definition
mr	1 bit	The *media clock restart* field is used by a talker to indicate that there has been a change in the source of the media clock. For example, this could happen when there is a change from one audio input to another audio input, and this input is the source of the media clock. A listener could use this notification to allow it to quickly adjust to the new media clock. This bit is toggled by the talker each time a media clock restart is needed, and it stays at this value until a new media clock restart is needed. This bit must remain in each state for a minimum of eight AVTP frame transmissions for a stream.
r	1 bit	This field is reserved for future standardisation.
gv	1 bit	The *gateway info valid* field is reserved for use by AVTP gateways. Its use is not defined in IEEE 1722.
tv	1 bit	The *timestamp valid* field is used to indicate whether or not the value in the *avtp timestamp* field is valid or not. A value of 0 indicates that it is not, and that the value of the *avtp timestamp* field should be ignored by a listener. A value of 1 indicates that the *avtp timestamp* field contains a valid value.
sequence number	8 bits	The *sequence number* field is used to indicate the sequence of the AVTP frame in a stream. The value of this field may start at any value, but subsequence AVTP frames should have the value incremented by 1. The value of this field wraps to 0x00 after it reaches 0xFF. This field can be used by listeners to detect lost AVTP frames.
reserved	7 bits	This field is reserved for future standardisation.
tu	1 bit	The *timestamp uncertain* field is used to indicate that the attached gPTP system has reported a significant problem with its synchronisation system. For example, this could happen when the grandmaster in the system changes. This field is used to indicate that the timestamps in AVTP frames may not be globally synchronised with network time. A listener could use this information to prevent unacceptable disturbances in the recovered media streams.

Table 21: AVTP common stream header *type specific data* field definitions

Table 22 shows the details of the additional header fields that are present in AVTP common stream headers.

Field	Size	Definition
avtp timestamp	32 bits	The *avtp timestamp* field contains the AVTP presentation time (if the *tv* field is set to 1). The *avtp timestamp* field is in units of nanoseconds and has a maximum value of $2^{32} - 1$. The *avtp timestamp* field is calculated as follows: $avtp\ timestamp = (AS\ s \times 10^9 + AS\ ns) \bmod 2^{32}$ *AS s* is the gPTP seconds field *AS ns* is the gPTP nanoseconds field
gateway info	32 bits	The *gateway info* field is reserved for use by AVTP gateways. The operation of AVTP gateways is not defined in IEEE 1722.
stream data length	16 bits	The *stream data length* field is used to indicate the length of the stream's payload (in octets) of all valid data octets contained in the frame after the *protocol specific header* field.
protocol specific header	16 bits	The *protocol specific header* field is used to carry data that is specific to the protocol being carried in the AVTP frame. The definition of this field is dependent on the value of the *subtype* field.
stream data		The value of this field is dependent on the on the protocol being transported by AVTP.

Table 22: AVTP common stream header *additional header* fields

6.1.2.2 IEC 61883 over AVTP

AVTP allows for the transportation of a subset of the IEC 61883 family of protocols. This work focused on the transportation of IEC 61883-6: Audio and music data transmission protocol [53]. An AVTP frame transporting IEC 61883-6 has a *CD* field value of 0 (to indicate that the frame contains stream data) and a *subtype* field value of 0 (see Table 20 on page 147) to indicate that the frame is transporting IEC 61883 conformant data. IEC 61883 makes use of the *protocol specific header* field of an AVTP common stream header as shown in Figure 85.

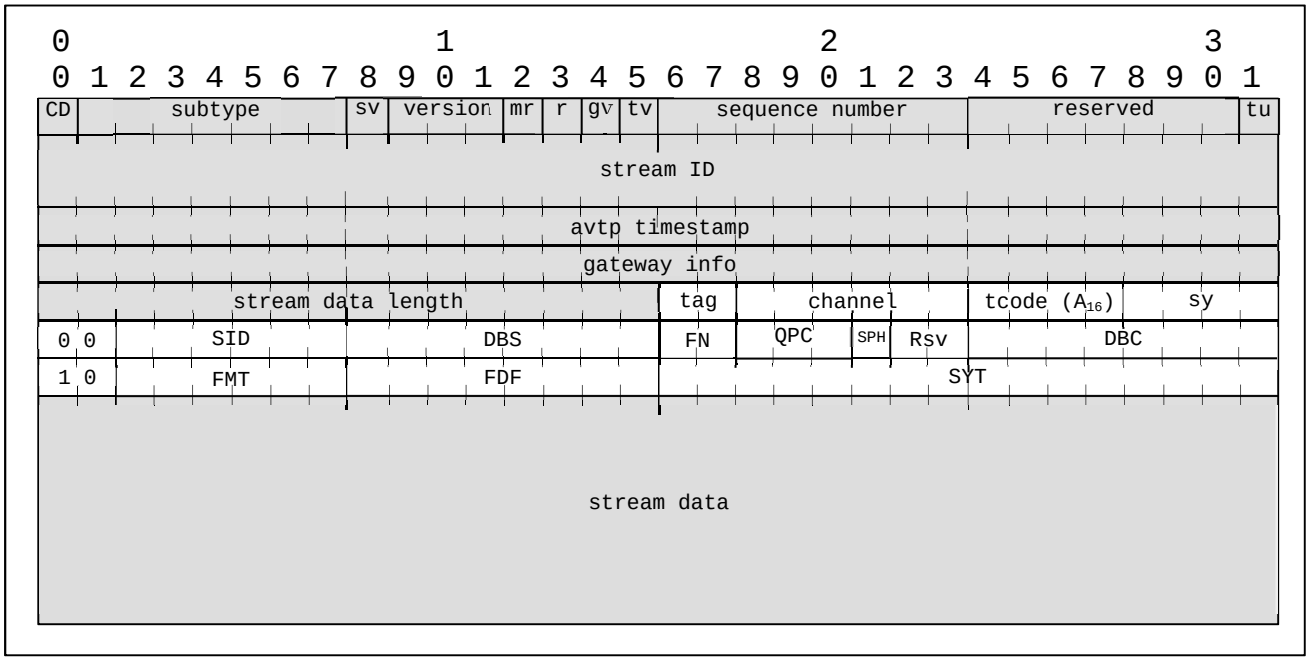

Figure 85: AVTP common stream header with CIP header

The fields of the *protocol specific header* are as defined in Table 23.

Field	Size	Definition
tag	2 bits	The definition of the *tag* field has the same definition as for IEEE 1394. For IEC 61883-6 the value of the *tag* field is set to a value of 01_2. This indicates that a CIP header is included after the *protocol specific header*.
channel	6 bits	The definition of the *channel* field has the same definition as for IEEE 1394. When IEC 61883-6 data originates from an AVB network, the *channel* field is set to a value of 31. When IEC 61883-6 data originates from an IEEE 1394 network, the *channel* field represents the channel number that was used to transmit the data onto the IEEE 1394 network via a gateway device. In this case, the valid values are 0 – 30 and 32 – 63, inclusive.
tcode	4 bits	The definition of the *tcode* field has the same definition as for IEEE 1394. The value of this field is always set to 0x0A.
sy	4 bits	The value of the *sy* field is application specific, and its use is outside of the scope of IEEE 1722. IEC 61883-6 does not use this field.

Table 23: IEC 61883 *protocol specific header* fields

The fields of the CIP header are defined in Table 24, and their definitions remain similar to those of the CIP header field definitions in IEEE 1394 isochronous packets (see Table 14 on page 140).

Field	Size	Definition
SID	6 bits	The value of the *SID* field is set to a value of 63 when the audio data originates from an AVB network. When the audio data originates from an IEEE 1394 network (via a gateway), the *SID* field's value represents the 6 bit physical ID of the node that transmitted the packet.
DBS	8 bits	Audio visual data is clustered into one or more data blocks. The *data block size* field represents the size of the data blocks. A value of 0 indicates 256 quadlets, and any other value represents the actual number of quadlets.
FN	2 bits	The *fraction number* field is used to indicate the number of fractions into which the original application packet was divided. This is not used for audio samples, so its value is 0.
QPC	3 bits	The *quadlet padding count* field is used to represent the number of padding bytes added to each application frame to make it divide evenly into 2^{FN} data blocks. The value of this field is 0 for audio data.
SPH	1 bit	The *source packet header* field is set to 0 for audio data indicating that the packet contains no sources packets.
rsv	2 bits	This field is reserved for future standardisation.
DBC	8 bits	The *data block count* field represents a count of data blocks.
FMT	6 bits	The *format* field is used to indicate the format of the data being transported.
FDF	8 bits	The *format dependent field* is used to identify sub-formats or convey other information.
SYT	16 bits	The *synchronization timing* field is not uses by AVTP end stations, but is used by gateways. If the audio data originated from an IEEE 1394 bus, the value of the SYT field indicates the time that a particular data block within the packet should be presented at an IEEE 1394 receiver.

Table 24: CIP header field formats

6.2 Timing and Synchronisation

When IEEE 1394 CIP packets and AVTP packets are transmitted across a network, they are time-stamped with a presentation time value. This presentation time value is used so that multiple receivers of a stream present the audio samples simultaneously. It is also used to recreate media

clock sampling frequencies. Chapter 5 “Timing and Synchronisation” detailed how all of the devices on an IEEE 1394 bus, and how all of the devices on an AVB network share a common sense of time. This section discusses how the timestamp information transmitted in these packets is used in conjunction with a common sense of time to perform synchronisation.

6.2.1 IEEE 1394

When transmitting IEC 61883-6 AM824 audio data across an IEEE 1394 bus, each packet is time stamped with a presentation time. This time value is a combination of the time value read from the transmitter’s *cycle time* register and a delay offset value known as a *transfer delay*. The *transfer delay* takes care of the delay incurred in the transfer of an isochronous packet on its path from a transmitter to a receiver. It allows packets travelling along different paths to be presented simultaneously, and it also takes account of a potential *short bus reset* that may occur during transmission. The *default transfer delay* is 354.17 µs. This accommodates the maximum latency of a CIP transmission through a short bus reset.

The presentation time is contained within the *SYT* field of the CIP header. The format of the *SYT* field is the same as the format of the lower 16-bits of the *cycle time* register, as seen in Figure 62 on page 111. The *SYT* field’s value applies to one of the events (data blocks) contained within the packet. The exact event is dependent on the sampling frequency of the media clock. This timestamp interval is known as the *SYT interval*. The sampling frequency (and hence the *SYT interval*) of the transmitter’s media clock is communicated in the *SFC* field of each packet’s *FDF* field, as shown in Table 25. For example, if the transmitter’s media clock is running at 48 kHz, then the timestamps are associated with each eighth event.

SFC (Nominal Sampling Frequency)		
Value	**Sample transmission frequency**	**SYT Interval**
0x00	32 kHz	8
0x01	44.1 kHz	8
0x02	48 kHz or not indicated	8
0x03	88.2 kHz	16
0x04	96 kHz or not indicated	16
0x05	176.4 kHz	32
0x06	192 kHz or not indicated	32
0x07	Reserved	

Table 25: SFC definition

A receiving device is able to extract the timestamp and use it to re-create the sampling frequency of the transmitted audio samples. When a receiver receives a CIP packet, it reads the timestamp value from that packet, and when that value is equal to its own *cycle time* register value, it indicates a match to a *phase locked loop* (PLL). The PLL receives a continuous series of match pulses, and effectively multiples the frequency of the pulses by the *SYT interval* to produce a sample clock of the transmitter's frequency. This is shown diagrammatically in Figure 86. In the figure, a transmitter is generating samples at a rate of 48 kHz, and packing the samples into data blocks within isochronous packets. At every eighth tick of the sample clock, the *cycle time* register of the transmitter is sampled, and its value included (with the addition of an offset) into the CIP header of the next isochronous packet to be transmitted.

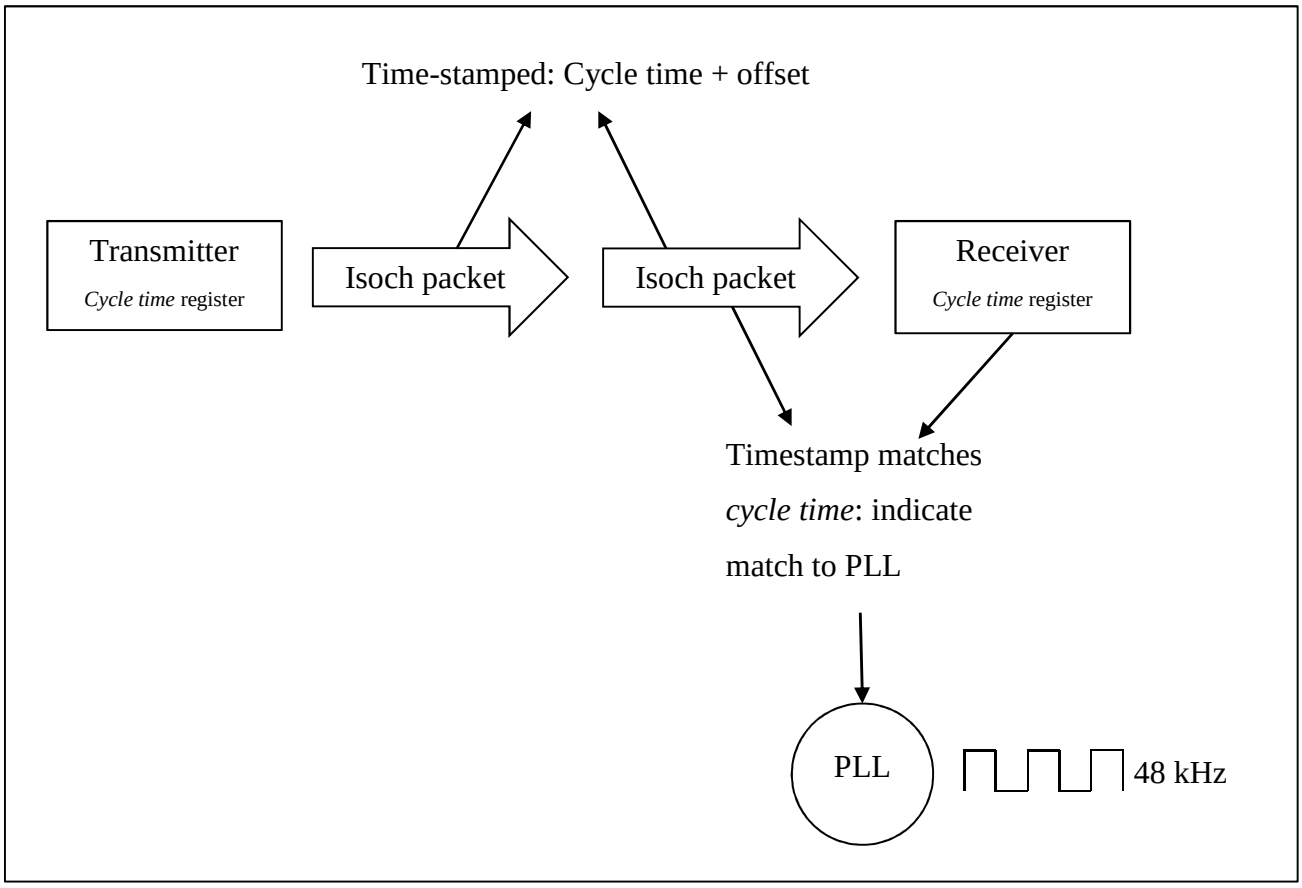

Figure 86: Sample clock synchronisation

Depending on the transmission type, every isochronous packet may not contain the number of data blocks equal to the *SYT interval*. If, for example, a transmitter is sampling at 48 kHz, it would have a *SYT interval* of eight, and it could be placing six data blocks into each isochronous packet. This implies that the timestamp will not be consistently associated with a data block at a particular position. A transmitter will prepare a timestamp for a data block that meets the following condition:

mod (*DBC*, *SYT interval*) = 0

At 48 kHz, every eighth data block has a time stamp associated with it.

At the receiver, the index of the data block to which the timestamp applies can be calculated as follows:

mod (*SYT interval* – mod(*DBC*, *SYT interval*)), *SYT interval*)

Once a receiver has received a time-stamped packet and determined the data block associated with the timestamps, it will not process this data block, and subsequent data blocks, until the cycle time in

its *cycle time* register matches the timestamp. The recovery of the timing for events between events to which the SYT timestamp applies is implementation specific.

6.2.1.1 Cross IEEE 1394 Bridge Timestamp Regeneration

An IEEE 1394 bridge is responsible for filtering and transforming isochronous subactions with respect to channel numbers. It is also responsible for transforming CIP packet physical ID and timestamp fields. A CIP packet that contains a two-quadlet CIP header has its SID field set to the physical ID of the transmitting portal.

The SYT timestamp field in a CIP packet contains an absolute timestamp value of when a time-stamped audio sample is to be presented at a receiver. The timestamp value generated at the transmitting device is relative to the transmitter's clock. When a CIP packet traverses an IEEE 1394 bridge, the timestamp value in that packet needs to be recalculated relative to the transmitting portal's 24.576 MHz clock.

The timestamp is specified as an absolute value that specifies a future cycle time. Isochronous subactions experience a constant delay when routed through a bridge. It is therefore sufficient to transform the timestamps by the addition of this constant value plus the difference in cycle times perceived by the portal and its co-portal. The difference in cycle time between the two sides of the bus, *delta*, is measured by implementation-independent means and is defined as follows:

$$delta = cycle\ count_{\text{talking portal}} - cycle\ count_{\text{listening portal}}$$

The *SYT* field is transformed as follows:

$$SYT_{\text{transmitted}} = (SYT_{\text{observed}} + ((latency + delta) << 12)) \ \&\ 0x0000FFFF$$

6.2.2 Ethernet AVB

AVTP defines a presentation time that allows for synchronisation between a talker and listeners. Each AVTP common stream header contains an *avtp timestamp* field which carries a presentation time timestamp. The value of this field represents the gPTP time when the data contained in the AVTP frame is to be available to AVTP listeners. The value of this field is expressed in nanoseconds.

The AVTP presentation time is used as a reference to synchronise any necessary media clocks and to determine when the first sample of a stream is to be presented to the client. The exact usage of the AVTP presentation time is media format dependent.

6.2.2.1 AVTP Presentation Time

The AVTP presentation time represents the time that the media sample was presented to the AVTP component of a talker plus constant transit time (known as *max transit time*) to compensate for network latency. The actual network latency is dependent on the network configuration and speed. The value of *max transit time* represents the maximum network latency assumed for a given configuration. It is possible for a talker and a listener to determine the *max transit time* value to use for a given stream but this mechanism is outside of the scope of IEEE 1722. The default *max transit time* for an SR class A stream is 2ms, and 50ms for an SR class B stream. These default times have been decided upon to support up to seven Ethernet hops for class A streams, and up to two WiFi hops plus six Ethernet hops for class B streams.

The AVTP presentation time, as received by listeners, is used to synchronise the media clock of the listener to the talker's media clock. The AVTP presentation time is directly linked to the gPTP global time and as such may be used to synchronise multiple listeners.

6.2.2.2 Presentation Time Measurement Points

Figure 87 shows the measurement points used when calculating the AVTP presentation time. The presentation time is the gPTP time that the 1722 layer of a listener transfers the time stamped data to the next processing layer in the stack. This is shown as the Presentation Time Reference Plane in Figure 87. The AVTP presentation time is used in the listener to know when to start processing (e.g., playing) a new stream and it is used to recover a stream's media clock.

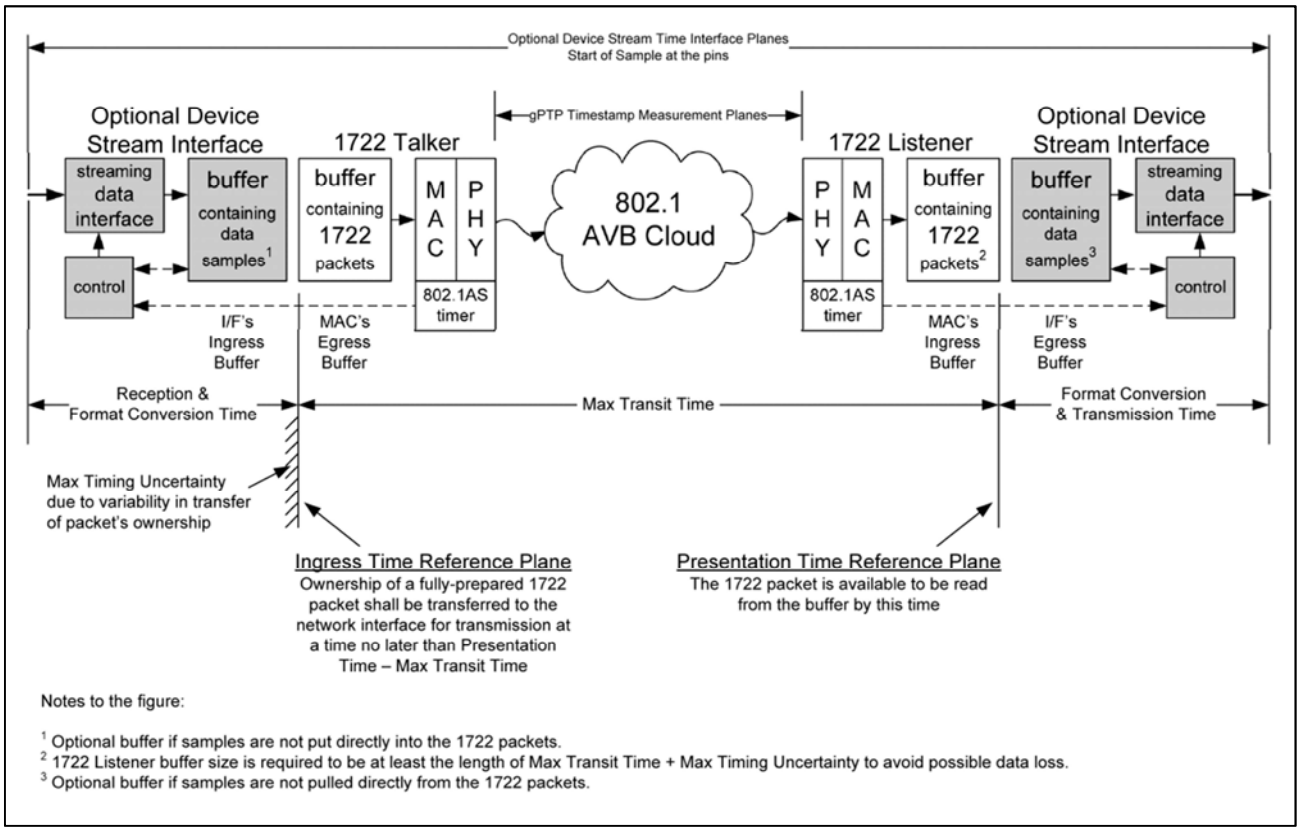

Figure 87: AVTP presentation time measurement point [55]

A talker generates an AVTP timestamp value that is in the future by the *max transit time* amount for the stream class (unless otherwise configured). In Figure 87, the Ingress Time Reference Plane is the point in a talker where an AVTP frame is transferred to the MAC for transmission. A talker transfers the frame to the MAC layer at a time that is no later than the *AVTP presentation time – max transit time*. If it does transfer the frame later than this time, the AVTP frame may not arrive at the listener before it is required for playback resulting in loss of part of a media stream.

There is always some uncertainty about exactly when this Ingress Time Reference Plane crossing will occur. A talker is responsible for knowing this *timing uncertainty* of its own design and to begin the transfer of a frame such that, in the worst case, the crossing of the Ingress Time Reference Plane will occur no later than *AVTP presentation time – max transit time*. The *timing uncertainty* of a talker is a combination of a number of factors, including its own local timer resolution and its worst-case response time to a timer event. An AVTP talker's design should be such that its *timing uncertainty* is not more than 125µs for an SR class A stream, and not more than 1ms for an SR class B stream.

6.2.2.3 IEC 61883-6 Timing and Synchronisation

The processing of presentation time, and the timing and synchronisation for IEC 61883-6 over AVTP is generally accomplished in the same manner as for IEEE 1394. AVB uses the gPTP time expressed in nanoseconds for its presentation times, whereas IEEE 1394 uses a time expressed in seconds, cycles and ticks of a 24.579 MHz clock.

If an AVTP frame consists of multiple data blocks, it is necessary to specify which data block corresponds to the AVTP timestamp. A transmitter generates a timestamp for the data block that meets the criterion

$$\text{mod}(\textit{data block count}, \textit{SYT interval}) = 0$$

The *data block count* is a running count of the transmitted data blocks. The *SYT interval* indicates the number of data blocks between two successive valid AVTP timestamps. The receiver is able to derive the index of the data block to which the AVTP timestamp applies with the following formula:

$$\textit{index} = \text{mod}((\textit{SYT interval} - \text{mod}(\textit{data block count}, \textit{SYT interval})), \textit{SYT interval})$$

A listener is responsible for determining the timing of successive data blocks between valid time stamps.

6.3 AVTP Address Allocation

A block of multicast MAC addresses has been reserved for use by AVTP for the transmission of streams. The *MAC address acquisition protocol* (MAAP) [55] is a peer-to-peer protocol that is responsible for the dynamic allocation of these addresses to devices that make use of AVTP. Table 26 shows the range of addresses that has been allocated for use by AVTP. MAAP is responsible for the allocation of the range 91:E0:F0:00:00:00 - 91:E0:F0:00:FD:FF.

Address range	Use
91:E0:F0:00:00:00 - 91:E0:F0:00:FD:FF	This range of addresses can be dynamically allocated by MAAP to devices making use of AVTP.
91:E0:F0:00:FE:00 - 91:E0:F0:00:FE:FF	This address range is reserved for static allocation to devices making use of AVTP.
91:E0:F0:00:FF:00 - 91:E0:F0:00:FF:FF	This address range is reserved for specific functionality. Currently, the address 91:E0:F0:00:FF:00 is reserved for use in MAAP frames. The rest of the addresses are reserved for future standardisation.

Table 26: AVTP address ranges

MAAP is able to allocate a single address, or a range of consecutive addresses. When a device would like to make use of one or more of the dynamically allocatable addresses, its instance of MAAP:

- Selects a random continuous range of multicast addresses from the dynamically allocatable address range.
- Sends up to three *MAAP probe* messages to the network to determine if all or part of the address range is already in use by any other devices on the network. The *MAAP probe* message contains the first address and the number of addresses that it would like to acquire. *MAAP probe* messages are transmitted at a random interval between 500 and 600 milliseconds.
- Listens for *MAAP defend* messages from other instances of MAAP. A *MAAP defend* message indicates that all or part of an address range being probed is in use.
- Repeats the above process until an address range that is not in use has been found.

Once an instance of MAAP has acquired an address range, it is responsible for defending it by:

- Periodically transmitting *MAAP announce* messages to the network to announce the fact that it is using a certain range of multicast MAC addresses. *MAAP announce* messages contain the first address of the range, and the number of addresses in the range of addresses which that instance of MAAP is using. *MAAP announce* messages allow address conflicts to be identified when networks are joined. *MAAP announce* messages are transmitted at a random interval that is between 30 and 32 seconds.
- Listening for *MAAP probe* messages and responding with *MAAP defend* messages if the address range identified in the *MAAP probe* message conflicts with an address range that it has acquired. The *MAAP defend* message contains the first address of the range and the number of addresses that were requested in the *MAAP probe* message. The *MAAP defend* message also contains the

first address that conflicts with the range that has been acquired, and the number of addresses that conflict.

- Listening for *MAAP announce* messages that conflict with previously acquired address ranges. If a conflict is detected, MAAP should discontinue the use of the address range and acquire a new address range.

MAAP probe and *MAAP announce* messages are sent to the MAC address 91:E0:F0:00:FF:00, whereas *MAAP defend* messages are sent to the MAC address of the station that transmitted the *MAAP probe* message.

6.4 Conclusion

Protocols have been developed that allow for the transmission of audio and video over IEEE 1394 and Ethernet AVB networks, and allow for stream and media clock synchronisation to occur. This chapter discussed the dominant media transport protocols for these networks, their packet formats, and the synchronisation mechanisms in place for each protocol.

These protocols use similar mechanisms for the transmission audio data. Audio data is formatted in a similar fashion on each network, and each network uses synchronisation mechanisms that are conceptually similar. When transferring audio data from one network to another, it has to be repackaged in a suitable format before transmission. Timestamp information associated with the audio data has to be regenerated such that it is relevant on the network it is being transmitted on. Chapter 10 “Ethernet AVB Devices” proposes a mechanism for achieving this.

Chapter 7 Standards-Based Command and Control Protocols

A number of command and control protocols exist that allow for remote control over parameters of networked devices. One of the goals of this work was to provide remote control over disparate networked audio devices from a single command and control protocol, with a focus on the ability to perform connection management within audio devices, and between audio devices. A protocol had to be selected for this task. This chapter provides an overview of a subset of the command and control protocols that were considered for the task. These protocols have all been used, or proposed, to provide control over network audio devices. The chapter concludes with an overview of the XFN protocol which was used in this study for control over audio devices.

7.1 Simple Network Management Protocol

The *simple network management protocol* (SNMP) [57] is a UDP-based [76] protocol used for managing devices on IP [61] networks. SNMP can be used to monitor and control network devices remotely. At the core of SNMP is a set of operations that enables the modification of parameters on SNMP-based devices. SNMP is usually associated with the monitoring and controlling of bridges and routers, but it can be used for managing many types of devices. SNMP has been used to control networked audio devices that reside on CobraNet [77] networks, for instance, as discussed in [78], and has also been proposed to control AVTP capable AVB devices, as discussed in [79] (see Section 7.1.7 “Connection Management”).

7.1.1 Managers and Agents

SNMP defines a *manager* and *agent* entity. A manager entity runs software that can handle management tasks for a network, and an agent is a piece of software that runs on a device being monitored and controlled. A manager is responsible for *polling* and receiving *traps* from agents on a network. When a manager polls a device for certain information, it performs a query on an agent for that information, whereas a trap is a mechanism used by agents to inform managers that an event has

occurred. Once a manager has received a piece of information, it may use it. For example, a manager may be monitoring a bridge and it may discover that it has gone down. When this happens, it could alert the network administrator that this has happened.

7.1.2 UDP Transmission

UDP is a low overhead protocol when compared to the *transmission control protocol* (TCP) [80]. SNMP uses UDP as a transport protocol for its messages. Since UDP is a connectionless unreliable protocol, SNMP deals with lost packets. If a response to a request is not received after a certain amount of time, it is able to retransmit the request. The number of retransmissions is configurable.

This mechanism works when polling, but not for traps. When an agent sends a trap to a manager, and it does not arrive, the manager has no way of knowing that it was sent a trap, and the agent does not know that the trap did not arrive because the manager is not required to send a response back to the agent acknowledging the receipt of the trap.

7.1.3 SNMP Communities

SNMPv1 and SNMPv2 use *communities* to establish trust between managers and agents. An agent is configured with three community names:

- *Read-only*: allows for reading data values only.
- *Read-write*: allows for reading and writing data values.
- *Trap*: allows managers to receive traps from an agent.

7.1.4 Structure of Management Information

The *structure of management information* (SMI) provides mechanisms to define how managed objects are named, and their associated data types. A managed object is a piece of information that can be monitored or controlled via SNMP. For example, this could be something like the operational status of a network interface, the number of octets that a network interface has transmitted, or a text string describing the physical location of a device. Managed objects are defined using a subset of OSI's *Abstract Syntax Notation One* (ASN.1) [81]. ASN.1 is a notation used for describing data

transmitted by telecommunications protocols. It is concerned with the structural aspects of information (the encoding of data), and not the semantics related to this data. The SMI defines the subset of ASN.1 used by SNMP.

Each managed object has a *name*, *type* and *syntax*, and *encoding*. The name, or *object identifier* (OID), uniquely identifies a managed object. Managed object names usually appear in numeric form and in human readable form. A managed object's data type specifies how its data is represented and transmitted between managers and agents. A managed object is encoded into a string of octets such that it can be transmitted over a transport medium.

Managed objects may be arranged into tables where each row, or entry, in a table is made up of one or more related managed objects. For example, a device may have a number of network interfaces. Each of these network interfaces has a number of attributes, such as a description, a type, a transmission speed, and a physical address. A table can be used to describe this data such that each entry in the table defines a network interface with its properties. Table 27 shows an example table describing two network interfaces (along with their attributes). Section 7.1.4.1 "Naming OIDs" describes how SNMP represents tables.

Description	Type	Speed	Physical Address
10/100BaseTX Port 7/1	Ethernet CSMA/CD	1000000000	00:23:45:67:89:AB
10/100BaseTX Port 7/2	Ethernet CSMA/CD	100000000	00:11:22:33:44:55

Table 27: An interfaces table

7.1.4.1 Naming OIDs

Managed objects are arranged in a hierarchical tree structure. This structure is the basis for SNMP's naming scheme. An OID is a series of integers separated by dots (.). These integer values are used to identify nodes in the tree structure. The human readable form of an OID is a series of names separated by dots. Each name is used to identify a node in the tree structure. Figure 88 shows an example of a few top levels of an object tree structure. Each node is identified by a number and a name (the actual numbers and names in the figure are unimportant at this stage).

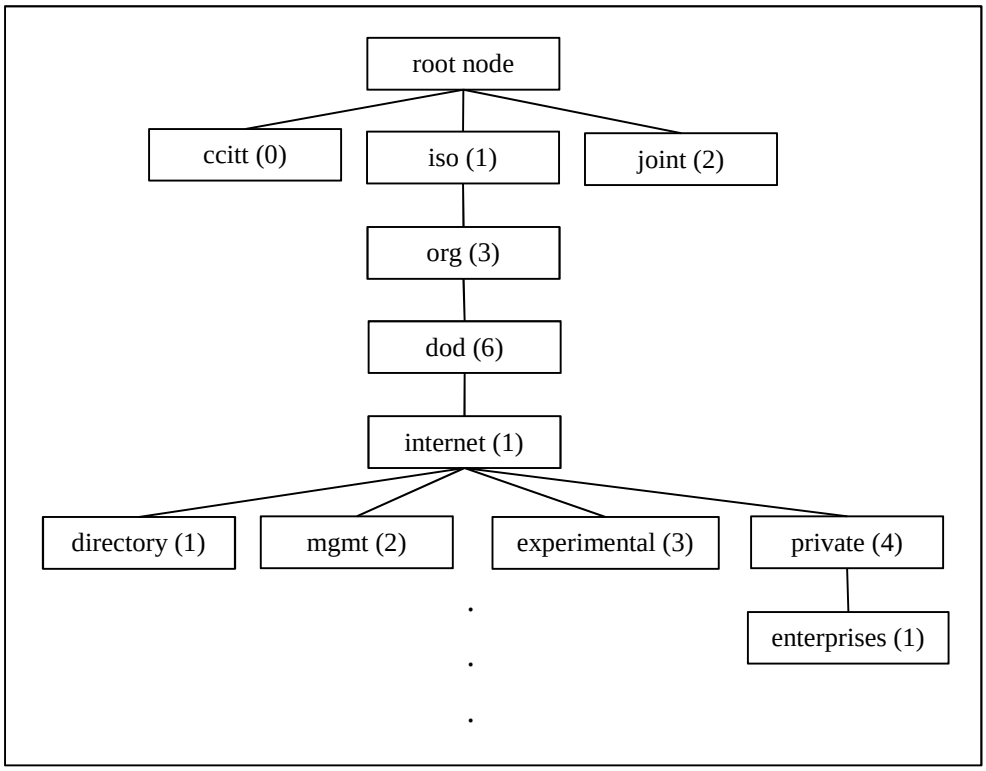

Figure 88: Example portion of an object tree hierarchy

The node at the top of an object tree is called the *root*, any node with children is called a *subtree*, and any node without any children is called a leaf node. The iso(1).org(3).dod(6).internet(1) subtree may be represented in OID form as 1.3.6.1 or iso.org.dod.internet.

The OID of a managed object has a number attached to the end of it. It uses the convention *x.y*, where *x* is the OID of the object type and *y* identifies an instance of that managed object. Scalar objects (that are not defined as a row in a table) use the identifier 0. If the objects are defined in a table, the instance identifier lets a row in the table be identified. The first row is identified by the identifier 1, the second row by the identifier 2, and so on.

For example, each attribute of a network interface (as seen in Table 27 on page 164) could be allocated an OID to uniquely identify it, as shown in Table 28. In order to query the managed object representing the description of the first interface, the OID .1.3.6.1.2.1.2.2.1.2**.1** would be used to address it. The description of the second interface would be addressed with the OID .1.3.6.1.2.1.2.2.1.2**.2**, and so on.

OID	Attribute
.1.3.6.1.2.1.2.2.1.2	Description
.1.3.6.1.2.1.2.2.1.3	Type
.1.3.6.1.2.1.2.2.1.5	Speed
.1.3.6.1.2.1.2.2.1.6	Physical Address

Table 28: Interface attributes

Figure 89 shows how SNMP would represent the interfaces table shown in Table 27 in the SNMP address hierarchy. The leaf nodes are used to address managed objects representing the various attributes of the network interfaces. The top level nodes in the address hierarchy have the following meanings:

- iso(1): OIDs that fall below the iso(1) base OID are assigned by the *International Organisation for Standardisation* (ISO) [82].
- org(3): OIDs that fall below the iso(1).org(3) base OID are assigned by organisations that are acknowledged by ISO.
- dod(6): OIDs that fall below the iso(1).org(3).dod(6) base OID are assigned by the US Department of Defence [83].
- internet(1): OIDs that fall below iso(1).org(3).dod(6).internet(1) are used to address managed objects related to the Internet.
- mgmt(2): OIDs that fall below iso(1).org(3).dod(6).internet(1).mgmt(2) are used to address Internet management managed objects.
- MIB-2(1): iso(1).org(3).dod(6).internet(1).mgmt(2).MIB-2(1) is the base OID for managed objects defined in MIB-2. MIB-2 defines managed objects that allow for the management of TCP/IP based networks.
- interfaces(2): iso(1).org(3).dod(6).internet(1).mgmt(2).MIB-2(1).interfaces(2) is the base OID for managed objects related to the IP interfaces of a device.
- ifTable(2): iso(1).org(3).dod(6).internet(1).mgmt(2).MIB-2(1).interfaces(2).ifTable(2) is the base OID for a table listing the interfaces of a device. Conceptually, the table is composed of ifEntry(1) entries. Each ifEntry has a number of fields, such as the description (ifDesc(2)), type (ifType(3)), speed (ifSpeed(5)), and address (ifPhyAddr(6)) of an interface.

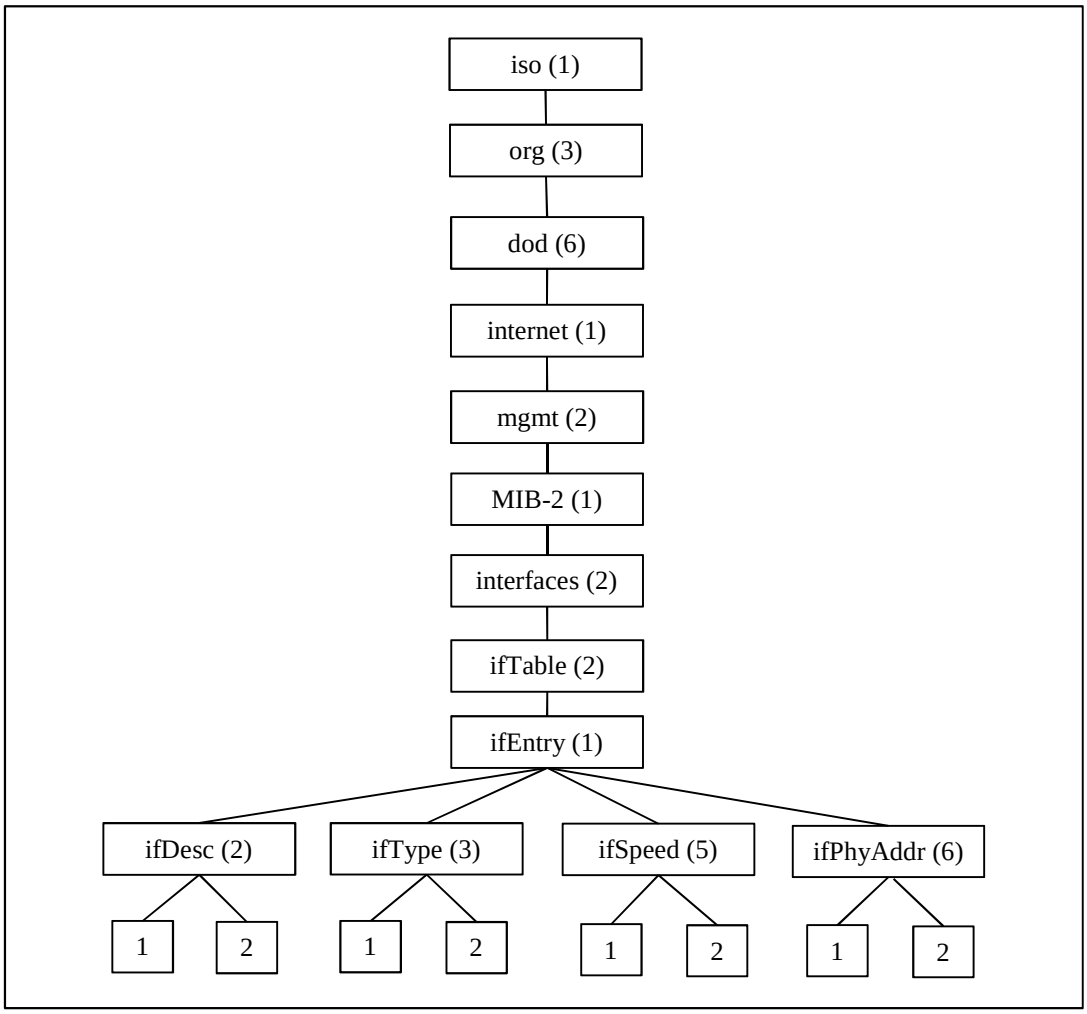

Figure 89: Representing the interfaces table in the SNMP address hierarchy

It is possible for anyone to implement their own object tree hierarchy for their own needs under the *enterprises* (1.3.6.1.4.1) base OID. An application can be made to the *Internet Assigned Numbers Authority* (IANA) [84] for the assignment of a *private enterprise number*. The owner of this number is then free to define their own object tree structure under this base OID. For example, the owner of the enterprise number 2680 (Peak Audio, Inc) is free to define the object tree under the 1.3.6.1.4.1.2680 base OID. Parameters relating to CobraNet devices fall under this base OID, for example.

7.1.4.2 Object Data Type

SNMP defines a number of data types that managed objects may take on. These define the type of data that the object can hold. Examples of data types are:

- *Integer*/*Integer32*: This data type is a 32-bit number that is often used to represent enumerated types. For example, a managed object may be in either an *up*, *down*, or *testing* state represented by the numbers 1, 2, and 3 respectively.
- *String*: A string of zero or more octets generally used to represent text strings.
- *Counter*/*Counter32*: This data type is a 32-bit unsigned number. When its maximum value is reached, it wraps to zero and starts over again. The value of a counter type should never decrease in value.
- *OID*: An OID for a managed object within an object tree.
- *IP address*: This is a 32-bit data type used to hold an IPv4 address.
- *Gauge*/*Gauge32*: This data type is a 32-bit unsigned number. The gauge type (unlike the counter type) is able to increase and decrease in value.
- *Time ticks*: This data type is a 32-bit unsigned number that is used to measure time in hundredths of a second.

7.1.5 Management Information Bases

The *management information base* (MIB) is a collection of managed objects (arranged in a tree structure) that an agent tracks. Any information that can be accessed by a manager is defined in a MIB. The SMI provides mechanisms to define managed objects, while a MIB is the definition of the objects themselves. An agent may implement many MIBs. Vendors are free to implement MIBs for their own use.

7.1.6 SNMP Operations

It is possible for a manager to gather information from an agent. SNMP defines a number of operations. The core operations are:

- *Get*: The *get* operation allows a manager to send a request to an agent for a specific piece of information. The request contains the OID that the requestor is interested in. The agent processes this request and responds with the requested information in a *get response* message. If the request cannot be performed, the *get response* message contains a meaningful error code. The *get* operation is able to attempt to retrieve more than one managed object at once, but message sizes

are limited by agent capabilities. If a particular agent is unable to return all of the requested responses, it returns an error message with no data.

- *Get next*: The *get next* command allows a manager to retrieve a group of object values from an agent. For each managed object of interest, a separate *get next* request and *get response* is generated. A *get next* command is used to traverse a subtree in lexicographical order. When an agent responds to a *get next* command (and this command is received by the manager), the manager issues the next *get next* command. This process continues until the agent returns an error, which signifies the end of the MIB has been reached. The *get next* command is useful for retrieving information out of a table where the length of the table is not known. It is possible to issue a *get next* for the start of a table (using an instance identifier of 0) which will return the first entry in the table (which is identified with the instance identifier 1), if the entry exists. To retrieve the next entry in the table, the instance identifier 1 is specified, which will return the second entry (identified with the instance identifier 2), if the entry exists. If the manager attempts to read past the end of the table, an error is returned by the agent.
- *Get bulk* (SNMPv2 and SNMPv3): The *get bulk* operation allows for the retrieval of a large section of a table at once. With the *get bulk* command, as much of the response as possible is sent to the requesting manager (it is possible to send incomplete responses to a manager).
- *Set*: The *set* operation is used to change the value of a managed object, or to create a new row in a table. Only objects that are defined *read/write* or *write only* can be modified or created using *set*. It is possible for a manager to set more than one object at a time. When a *set* command is sent to an agent, it responds with a *get response* to specify whether the command was successfully carried out. It is possible to set more than one managed object at a time, but if any of the sets fail, all of them fail.
- *Trap*: A trap is sent from an agent to a manager. The agent is configured with the trap destination address. A manager does not acknowledge receipt of trap messages. A trap informs a manager when a managed object changes state. A trap message contains managed objects and their values. It is the responsibility of the manager to respond to the trap message appropriately.

7.1.7 Connection Management

SNMP enables object tree hierarchies to be built that represent the parameters of a device, and allow their values to be obtained and adjusted.

Connection management within a device occurs when audio signals are routed from signal source points to signal destination points within a device. For example, an audio signal arriving at the analogue input of a device could be patched through to an AES-3 interface, or to the channel of an AVTP stream, of the same device. Connection management between audio devices occurs when audio signals are patched from an output of one device to the input of another device. For a connection to be established, certain parameters need to be configured on the devices that are part of the connection.

The use of SNMP was proposed in [79] for control over AVTP capable AVB devices. The document proposes to represent AVTP stream sources and stream sinks with rows in SNMP tables, where each row in a *source* table represents a source AVTP stream, and each row in a *sink* table represents a sink AVTP stream. Table 29 shows an example table representing two source streams with a few of their properties.

Name	**Stream ID**	**Multicast MAC**	**State**
“Drum Kit”	01:23:45:67:89:AB:00:01	01:23:45:67:89:AB	Streaming
“Stage Mics”	01:23:45:67:89:AC:00:02	01:23:45:67:89:AC	Advertised

Table 29: An example source table

Each row in one of the tables contains properties (each property represented with a column) associated with the stream, such as:

- Name: A textual name for the stream.
- Stream ID: The stream ID associated with the stream. For a source stream, this represents the stream ID allocated to the stream by the device. For a sink stream, this represents the stream ID of the stream that the device would like to/is receiving.
- MMAC: The MAC address allocated by MAAP for the stream. For a source stream, this represents the MAC address allocated to the stream by MAAP. For a sink stream, this represents the MAC address of the stream that the device would like to/is receiving.
- State: The state of the stream. This property represents the streaming state of the stream. It is useful to determine whether MSRP successfully reserved resources for the stream.
- Format: The format of the stream. AVTP is able to transport a number of media formats, and this argument allows the format of the stream to be specified. This can allow a hardware device to configure itself to receive the particular stream format, for example.

- Channels: The number of channels contained within the stream.
- Map: The map field is used for connection management internal to a device. For a source stream, the map property is used to determine where the channels in a stream are receiving their signals from within a device. For example, the map property could map the first and second analogue input of a device to the first and second channel in an AVTP stream. For a sink stream, the map property is used to determine where the channels in a stream are routed to within a device. For example, the map property could map the first and second channel of a stream to the first and second analogue outputs of a device.
- Presentation: The presentation time offset. The value of this field determines the presentation time offset that is added to the presentation time contained in AVTP frames. This field allows the presentation time to be fine-tuned.

The creation of a row in the *source* table allows an AVTP stream source to be created. In SNMP, this can be achieved via a read-only managed object that indicates the next available row index. When this managed object is read (with a *get request*), a new row is created in the table and the index of the new row is returned to the caller. A row can be deleted by setting its *state* field to a value indicating that the stream is no longer required. Creating a row in the *source* table results in the stream being advertised to the attached AVB network (via MSRP). The creation of a row in the *sink* table allows an AVTP stream sink to be created. This results in the device requesting attachment (via MSRP) to a stream advertised by another device.

7.1.8 Tools

A number of tools exist that allow for the use and development of applications that make use of SNMP. Net-SNMP [85], for example, is a suite of tools and libraries that allow for:

- The retrieval and manipulation of information on SNMP-capable devices
- Graphical MIB browsing of SNMP enabled devices
- The reception of SNMP notifications (traps) from agents
- Applications to be built to respond to SNMP queries. These applications can define their own MIBs and respond as required to queries on the managed objects defined in those MIBs.

7.1.9 Conclusion

SNMP provides the means for address hierarchies to be built that reflect the natural structure of a device. The leaves of these hierarchies point to the parameters (*managed objects*) of a device. The native commands of the protocol allow for the values of the managed objects to be obtained and adjusted. The SNMP table structure offers a convenient mechanism for representing streams that a device has to offer, or that a device would like to receive. Each row of a table represents a stream, and the columns of the table represent the properties of that stream.

7.2 IEC 62379

IEC 62379 [86] is a set of specifications that specify a *control framework* for control over networked audio visual equipment. The framework allows for control over network transmission, and for control over internal device parameters. The control framework provides a consistent interface to the functionality provided by audio/visual units. IEC 62379 allows systems to be built that are plug-and-play. Devices are able to discover units that are connected to a network, and interrogate them to determine their capabilities. The natural functional groups of a device are represented in a consistent and structured manner. IEC 62379 is designed such that it is extensible. IEC 62379 is composed of parts 1 through 6. IEC 62379-1 [75] specifies aspects that are general to all equipment, and IEC 62379-2 [87] specifies aspects of the control framework that are specific to audio devices. IEC 62379-5 specifies control over the transmission of various realtime media over networking technologies. IEC 62379-5 sub-part 1 [88] specifies management of aspects that are common to all network technologies and IEC 62379-5 sub-part 2 [89] specifies protocols which can be used between networking equipment to enable the setting up of connections.

7.2.1 Equipment Structure

The natural structure of a device forms the basis of the control framework. A device, or a *unit*, tends to be structured hierarchically as a set of functional *blocks*. These blocks are usually linked to each other through some form of internal routing and patching mechanisms. Blocks may have inputs, outputs, and internal parameters, as shown in Figure 90.

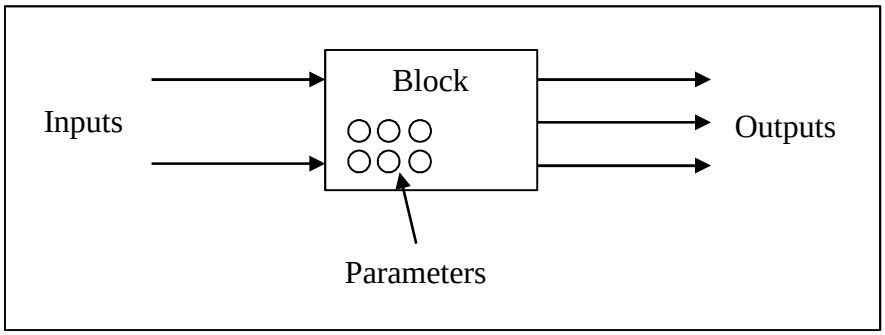

Figure 90: A block

The output of one block usually forms the input of another block. An input to a block may or may not be connected to one output of another block. The output of a block may be connected to zero or more inputs of other blocks. Groups of blocks that are connected together are called *processing chains*. Usually, a processing chain represents what a device does as a whole.

A *port* provides an external connection to another piece of equipment. An *input port* allows data (for example, audio and video) to enter a unit from an external entity, and an *output port* allows data to leave a device to an external entity. A port may correspond to a physical connector (for example, an XLR socket for analogue audio), or to a virtual connector (for example, the end point of a connection across a network). With respect to a unit, an input port has no inputs, and an output port has no outputs (these are external to the device). An input port has an output which is used to supply data to a block. An output port has an input which is used to receive data from a block.

Figure 91 shows an example unit that is composed of a number of blocks and ports. The outputs of the input ports (1, 2) are connected to the inputs of a mixer block (3). The output of the mixer block is connected to the input of a limiter block (4). The limiter block's output is connected to the output port's (5) input. These blocks and their connections form a processing chain. The numbers shown in brackets are block identifiers (see Section 7.2.3 "Control Framework").

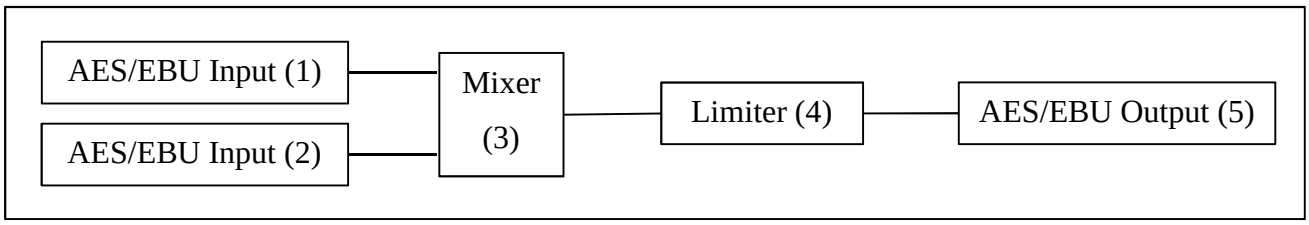

Figure 91: An example unit with blocks [87]

7.2.2 Managed Objects

IEC 62379 utilises SNMP, and communication between a controlling application and a unit takes place via SNMP (SNMPv1, SNMPv2, or SNMPv3 may be used). As indicated in Section 7.1 “Simple Network Management Protocol”, SNMP defines read and write operations that may be performed on a set of managed objects. The managed object types are arranged in a hierarchical manner and are accessible via OIDs. Managed object type OIDs that are part of the IEC 62379 series of standards begin with 1.0.62379.*p* if they are defined in part *p* of IEC 62379, or 1.0.62379.*p*.*s* if they are defined in sub-part *s* of part *p* of IEC 62379.

Each block is described by a group of managed objects. These managed objects represent the parameters that exist within a functional block. The structure of each type of block is specified in one of the parts of IEC 62379, or elsewhere for product-specific or application-specific blocks. Connections between blocks are described by a table containing identification of the output to which each input is connected. As an example, IEC 62379 defines an audio limiter to have a structure as shown in Figure 92. This block limits the incoming audio signal to a pre-set maximum level. The limiter block is defined as having the following parameters: threshold, attack time, gain makeup, recovery time, and recovery mode.

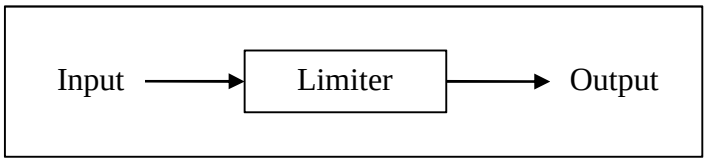

Figure 92: An audio limiter block

Instances of limiter blocks are defined as rows in SNMP tables. The root node for a limiter block type is identified with the OID .1.0.62379.2.1.5. The parameters of a limiter are identified with the following values appended to the OID:

- Block ID: 1
- Threshold: 2
- Attack time: 3
- Gain makeup: 4
- Recovery time: 5
- Recovery mode: 6

Therefore, in order to access the attack time parameter of the second limiter, the following OID will be used:

- iso(1).standard(0).IEC 62379(62379).part 2(2).audioMIB(1).audioLimiter(5).attackTime(3).2

Each managed object type within IEC 62379 is defined by the following attributes:

- *Identifier*: This specifies the name and number that identifies the object type relative to the group in which it belongs. For example, the value 2 identifies the threshold parameter in a limiter block.
- *Syntax*: This specifies the syntax of the abstract data structure representing the object value. This could be an integer, for example.
- *Index*: This specifies whether the object is used to uniquely identify a row in a managed object table. For example, the block ID of the limiter is used to uniquely identify a row in the table representing the limiters of a device.
- *Readable*: This specifies the privilege level for read access to the associated object. It is only permissible to read the managed object (with a *get* or *get next* SNMP operation) with a privilege level equal or greater to this privilege level. It is possible to set this to *none* to completely disable read access.
- *Writeable*: This specifies the privilege level for write access to the associated object. It is only permissible to write to the associated managed object (with a *set* SNMP operation) with a privilege level equal or greater to this privilege level. It is possible to set this to *none* to completely disable write access.
- *Volatile*: This attribute specifies whether the current value of the object is retained after a hard reset or period of power loss.
- *Status*: This attribute specifies the required level of implementation support for the object. This attribute is able to specify whether:
    - It is *mandatory* to implement the managed object in a unit that implements the object's parent group.
    - It is *optional* to implement the managed object in a unit that implements the object's parent group.
    - It is *deprecated*, indicating that the managed object will not be implemented in any newly designed units.

7.2.3 Control Framework

The control framework consists of a list of blocks that make up a unit, and a list of connections between these blocks. The list of blocks is contained in a *block table* and the list of connections is contained in a *connector table*. A number of blocks currently exist, and new ones continue to be introduced all of the time. Units can be developed from a combination of existing blocks and new blocks to provide desired functionality.

A defined block type is identified by an OID. When a new block type is specified, the OID of the block type identifies a MIB table, or group of MIB tables, with each table containing a variable number of rows. An instance of a block type in a table is indexed using a block identifier (*block ID*). A block ID is a unique number (with respect to an individual unit) that is used to identify individual blocks within a unit.

Table 30 shows the block table for the example device shown in Figure 91. As the unit is made up of five blocks, the table contains five rows (one for each block). Shown in each row is the block ID that a device has assigned to the instance of the block, and the OID that is used to identify the type of block.

Block ID	Block type
1	1.0.62379.2.1.1 (Audio port)
2	1.0.62379.2.1.1 (Audio port)
3	1.0.62379.2.1.2 (Mixer block)
4	1.0.62379.2.1.5 (Limiter block)
5	1.0.62379.2.1.1 (Audio port)

Table 30: An example block table [87]

Table 31 shows an example connector table for the example unit shown in Figure 91. A connection is formed between an output of a block to the input of another block. Each row in the table represents a connection. The first column of the table represents the block ID of the block containing the input, and the second column represents the actual input of that block. The third column represents the block ID of the block containing the output, and the fourth column represents the actual output of

that block. For the unit shown in Figure 91, the first row of the table indicates that AES/EBU Input (1) is connected to the first input of the mixer (3).

Connection Rx Block ID	Connection Rx Block Input	Connection Tx Block ID	Connection Tx Block Output
3	1	1	1
3	2	2	1
4	1	3	1
5	1	4	1

Table 31: An example connector table [87]

A controlling application may only need to be able to represent and control certain blocks within a unit. A controlling application is able to discover units that exist on a network, and is able to discover the blocks that each unit contains. It does not have to have knowledge of the unit itself. A controlling application may also retrieve any connections that may exist between blocks.

7.2.3.1 Media Formats

A *mode table* lists all of the media formats that are valid for each block's outputs. Table 32 shows an example mode table for the example unit shown in Figure 91. In this example, each of the outputs within the unit are capable of running at either 44.1 kHz or 48 kHz. The table displays the possible formats for each output of each block.

Block ID	Block output	Media format	Enabled
1	1	1.0.62379.2.2.1.3.2.2.24.44100	True (1)
1	1	1.0.62379.2.2.1.3.2.2.24.48000	True (1)
2	1	1.0.62379.2.2.1.3.2.2.24.44100	True (1)
2	1	1.0.62379.2.2.1.3.2.2.24.48000	True (1)
3	1	1.0.62379.2.2.1.3.2.2.24.44100	True (1)
3	1	1.0.62379.2.2.1.3.2.2.24.48000	True (1)
4	1	1.0.62379.2.2.1.3.2.2.24.44100	True (1)
4	1	1.0.62379.2.2.1.3.2.2.24.48000	True (1)

Table 32: An example mode table [87]

The media formats above are represented using OIDs, such as 1.0.62379.2.2.1.3.2.2.24.48000. The components of this OID are defined as shown in Table 33.

OID Part	Definition
1.0.62379	IEC 62379
1.0.62379**.2**	IEC 62379 part 2: audio
1.0.62379.2**.2**	Audio format
1.0.62379.2.2**.1**	Audio signal format
1.0.62379.2.2.1**.3**	The signal is encoded using Pulse Code Modulation (PCM)
1.0.62379.2.2.1.3**.2**	The signal is in stereo format
1.0.62379.2.2.1.3.2**.2**	The signal consists of two channels of audio
1.0.62379.2.2.1.3.2.2**.24**	Each sample is 24-bits in length
1.0.62379.2.2.1.3.2.2.24**.48000**	The sampling frequency is 48 kHz

Table 33: An example media format

7.2.3.2 Audio Ports

The ports that exist on a unit are represented in a *port table*. There exists an entry in the table for each port on the unit. Table 34 shows an example port table for the example unit shown in Figure 91. The table lists the block ID for each port, the direction of the port, the media format of the port, the port transport (for example, analogue or AES 3), and a name for the port.

Port block ID	Port direction	Port data format	Port transport	Port name
1	Input (1)	1.0.62379.2.2.1.3.2.2.24.48000	AES3 (1.0.62379.2.2.2.2)	AES/EBU
2	Input (1)	1.0.62379.2.2.1.3.2.2.24.48000	AES3 (1.0.62379.2.2.2.2)	AES/EBU
5	Output (2)	1.0.62379.2.2.1.3.2.2.24.48000	AES3 (1.0.62379.2.2.2.2)	AES/EBU

Table 34: An example port table [87]

7.2.4 Status Broadcasts

IEC 62379 has a *status broadcast* mechanism. A status broadcast reports on the values of a group of managed objects. Examples of managed objects that could be reported are the peak level of an audio signal, the media format being received at an input, or information about what is connected to a network port. The status broadcast mechanism is implemented using point-to-multipoint transmission.

Each one of the parts of IEC 62379 defines groups of managed objects that are available for status broadcasts. All of the managed objects in a group are reported periodically, and any managed objects whose values change are reported immediately. The periodic reporting ensures that units have the latest managed object values, even if they miss a report.

Status broadcasts are initiated by a unit in response to an appropriate command. A source unit transmits one or more *status pages* of status information. A *status page* is a message that contains structured values representing some internal state of a unit. Each page is organised into a fixed format of related information. A unit may define and support multiple types of status page. Related status pages are grouped together in *status broadcast groups*. When a remote entity requests a status broadcast, it specifies which group it is interested in receiving. A status broadcast group is identified by an OID.

7.2.5 Connection Re-establishment

If a piece of equipment fails for some reason, and as a result it drops a connection, it is important that these connections are re-established (this is especially important in a broadcast studio, or during a live sound concert, for example). A compliant network interface unit is required to reconnect stored connections. Connections should be stored in non-volatile memory, and these connections should be re-established after a temporary power failure. This is only intended to cover short-term failures, after which attempting to re-establish a connection should not be attempted. This prevents units that are redeployed from one area to another from attempting to re-establish a connection that existed in another location.

7.2.6 Privilege Levels

IEC 62379 uses *privilege levels* to distinguish between different kinds of users. This mechanism allows network resources to be set aside for more privileged users, and to prevent inappropriate control tasks being performed by less privileged users, for example. Four privilege levels are defined:

- *Listener*: This is the lowest privilege level, and it is intended to be used by those interested in monitoring audio or video signals passing through a unit. A listener is able to route signals from remote devices to themselves, but is unable to change anything that would affect the experience of other users.
- *Operator*: The operator privilege level is intended for use by those who are controlling the day-to-day operation of a unit. An operator is able to change parameters that may affect other users.
- *Supervisor*: The supervisor privilege level is intended for use by those who are controlling and maintaining a network.
- *Maintenance*: The maintenance privilege level is the highest privilege level, and it is intended to be used by those who need to perform tasks that might disrupt the normal operation of a unit.

7.2.7 Automation

The automation mechanism allows for single or multiple values to be set at a given time. An automation event consists of a time, the OID of an object, and the value to set the object to at the time. This allows events to occur simultaneously, or for events to occur one after the other.

7.2.8 Connection Management

IEC 62379-5 specifies control over the transmission of realtime data over digital networks. It uses the concept of a *call*. Before stream data can be transmitted, a call has to be set up between the parties involved in the transmission. In order to set up a call, a management terminal (running a control application, for example) sends commands to a destination unit, and the destination unit then asks the network to make the connection.

Some networks offer a connection-oriented service. In these cases, a call maps naturally onto this service. IEEE 1394 and Ethernet AVB both have the concept of connections, thus an IEC 62379 call would map to this.

A physical connection to a network is described with a block. The block has an input for each media flow going to the network, and it has an output for each media flow coming from the network. Figure 93 shows a block representing a physical network port. It is transmitting three streams, or *flows*, onto a network, and has an input for each one of these flows. It is also receiving three flows from the network, and has an output for each one of these flows.

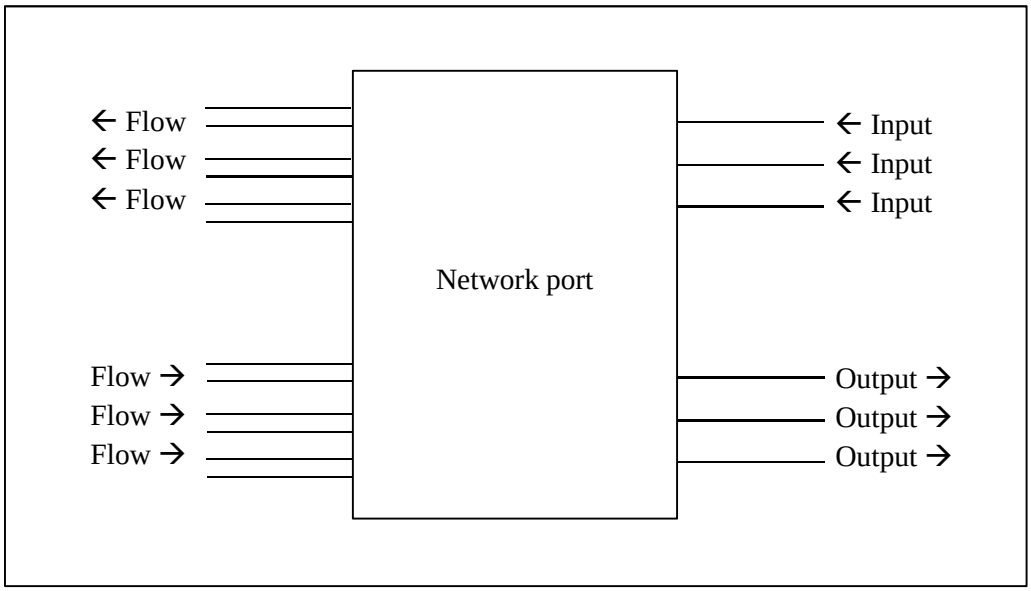

Figure 93: Inputs and outputs

Each device has a *unit destination list* and a *source destination list*. The *unit destination list* is a list of streams being received by a device, and a *unit source list* is a list of streams being transmitted by a device. Each entry in a list keeps track of properties that pertain to each stream. These include:

- The size of the payload of the maximum sized packet that forms part of the stream, and the rate at which packets are transmitted by the device. These values can be used to calculate the bandwidth required of a stream.
- The state of the stream. This value can be adjusted to create and teardown calls. For example, setting this field to *ready to connect* instructs the device to initiate a call. While the device is attempting to make the call, the state of the stream could be set to *call proceeding*. Once a call has been made, the state of the stream is set to *active*.
- A block identifier identifying the port through which the stream is flowing.
- In the case of a source stream, the input of the port block through which the stream is flowing.
- In the case of a destination stream, the output of the block through which the stream is flowing.

When a device would like to receive a particular stream, a new entry is created in the *unit destination list* of the unit that is to receive the stream. At this stage, the network port on which the stream is to arrive may not be known. A management terminal, once it has created the entry, has to fill in the fields of the new entry. This includes identifying the input of a block that is to receive the stream. Once this has happened, the management terminal requests the unit to make the connection on the network.

If a connection is successfully created, a new output is created (or an existing unused output is used) on the network port block through which the stream is to flow. This output is connected to an input of another block. In the case of an AVTP stream carrying sequences of audio data, the output of the network port block could be connected to an input of an AVTP multiplexing/ de-multiplexing block. The multiplexing/de-multiplexing block will be responsible for extracting the individual sequences of audio out of the incoming streams. Streams can be formed from the individual channels entering the block. This process is shown in Figure 94. The connections between the blocks are described in the connector table.

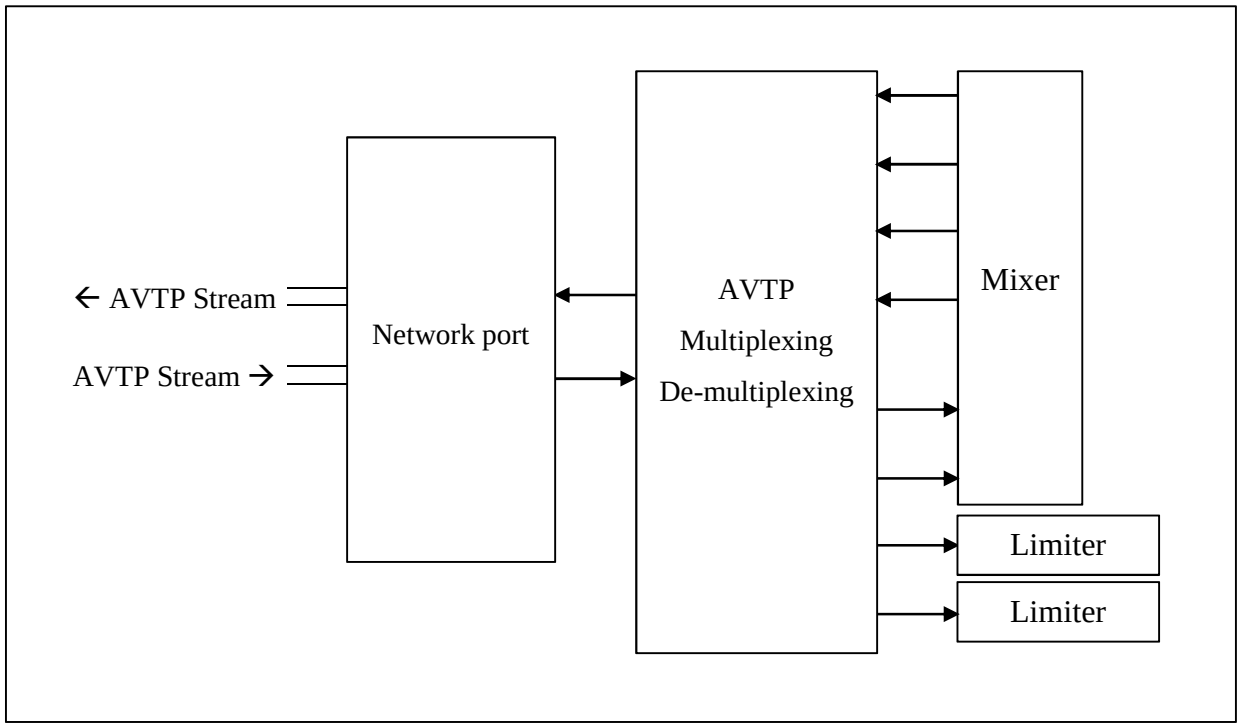

Figure 94: AVTP multiplexing/de-multiplexing

If a stream source is already transmitting the stream, and the network supports multicasting, the network equipment (a switch, for example) simply copies the existing stream to the new destination device. If a source device is not transmitting the stream of interest, the device will create (or assign) a new input to the relevant network port block, and connect it to the source.

7.2.9 Conclusion

IEC 62379 uses SNMP and makes use of its hierarchical addressing scheme and table concept in order to represent connections internal to a device, and connections between devices. Each entry in the connector table represents an internal connection between an output of a block, and an input of another block. In a similar way to that proposed in Section 7.1.7 “Connection Management”, each stream is represented with an entry in a table, where the fields of each entry represent the properties of the stream. Creating and breaking stream connections involves changing the state of the stream.

7.3 OSC

Open sound control (OSC) [58] is an open, transport neutral, message-based protocol that allows for communication between computers, sound synthesisers, and other multimedia devices. OSC communication occurs between an *OSC client* and an *OSC server* via *OSC packets*.

An OSC server has a set of *OSC methods*. OCS methods are methods that are triggered with the arrival of OSC packets from OSC clients. OSC methods may expect to be passed arguments from the OSC client. OSC methods are arranged in a tree structure called the *OSC address space*, with the leaves of the tree being the OSC methods. Branch nodes are referred to as *OSC containers*. The contents and the shape of the OSC address space may change over time.

7.3.1 OSC Address Space and OSC Addresses

Each OSC container (except the root OSC container) and each OSC method has a symbolic name. This name is composed of an ASCII character string consisting of printable characters other than the following characters: *space* # * , / ? [ ] { }

An *OSC address* of an OSC method is the symbolic name giving the full path to the OSC method in the OSC address space, starting with the root of the tree. An OSC address starts with the ‘/’ character, followed by the names of all the containers, in order, along the path from the root of the tree to the OSC method. Each container’s symbolic name is separated by a ‘/’.

Figure 95 shows an example OSC address space with each OSC container and method having a name. From this tree structure are formed the following OSC method addresses: “/first/this/one”, “/second/1”, “/second/2”, “/third/a”, “/third/b”, “/third/c”.

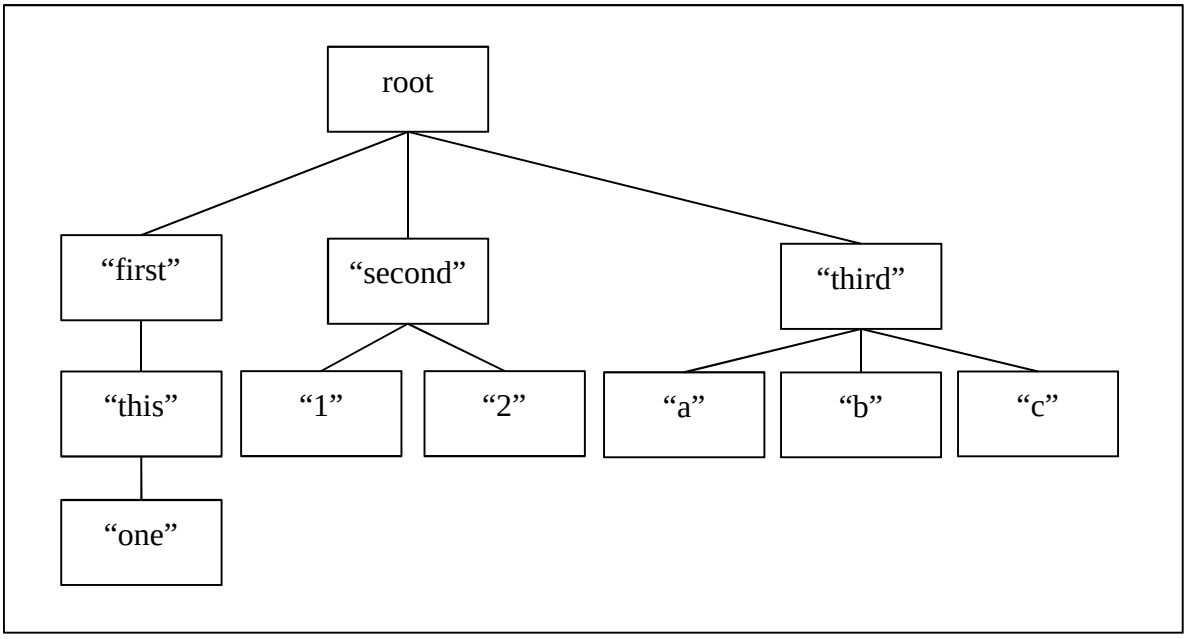

Figure 95: An example OSC address space

7.3.2 OSC Data Types

All OSC data are composed of a set of core types, as defined in Table 35.

Type	Definition
32 bit integer	32 bit big endian two’s compliment integer
OSC timetag	64 bit big endian fixed-point time tag
32 bit float	32 bit big endian floating point number
OSC string	A sequence of non-null characters, followed by a null character
OSC blob	A 32 bit integer count, followed by that many octets of arbitrary data

Table 35: Core OSC types

The size of each of the core types is a multiple of 32 bits to ensure 32-bit alignment.

7.3.3 OCS Packets

OSC packets are transmitted by OSC clients, and received by OSC servers. These packets either contain an *OSC message* or an *OSC bundle*.

7.3.3.1 OSC Message

An OSC message consists of an OSC address, followed by an *OSC type tag string*, followed by zero or more *OSC arguments*. The OSC address is the address of the method that should be invoked. The OSC type tag string is an OSC string that begins with the character ',' followed by a sequence of characters corresponding exactly to the sequence of OSC arguments in the OSC message. Each one of these characters is called an *OSC type tag* and is used to represent the type of each corresponding OSC argument. Table 36 lists the possible OSC type tags.

OSC type tag	Definition
i	32 bit integer
f	32 bit floating point number
s	OSC string
b	OSC blob

Table 36: OSC type tags

The OSC arguments are a sequence of arguments that will be supplied to the method addressed by the OSC address. Listing 1 shows an example OSC message. This message is targeted at the "input/level" OSC method which is used to adjust the audio level of an input. This OSC message contains two arguments, an integer and a float (as indicated with the *,if* part of the message). The first argument is a number used to identify the input whose audio level should be adjusted, and the second argument is the value that the audio level should be set to.

```
"/input/level" ,if (input number) (value)
```

Listing 1: An example OSC message

7.3.3.2 OSC Bundles

An OSC bundle consists of the OSC string "#bundle" followed by an OSC time tag, followed by zero or more *OSC bundle elements*. The OSC time tag is a 64-bit fixed time tag. Each OSC bundle element is composed of a 32-bit integer size indication followed by that number of octets as contents, and will always be a multiple of four. The contents of an OSC bundle element are either an OSC message or another OSC bundle.

7.3.4 OSC Message Processing

When an OSC server receives an OSC message, it invokes the OSC methods addressed by the OSC address pattern of the message. All of the methods that match the OSC address are invoked with the same argument data supplied in the OSC message. Any part of an OSC address may contain special characters with special meaning:

- '?': Matches any single character in an OSC address
- '*': Matches any sequence of zero or more characters
- String of characters in square brackets (e.g., "[string]"): Matches any character in the string. A '-' or a '!' character in the square brackets has the following meanings:
    - '-': Two characters separated by a minus sign indicate a range of characters between the two given characters on either side of the '-'.
    - '!': An exclamation mark at the beginning of a bracketed string negates the sense of the list (i.e., the list matches any character not in the list).
- Comma separated list of strings in curly braces (e.g., "{foo,bar}): Matches any of the strings in the list.

When an OSC server receives a single OSC message, the corresponding OSC methods should be invoked immediately. If the OSC server receives a bundle (which contains a time tag), the message methods are invoked when the time of the time tag is reached. If the time tag represents a time before or equal to the current time, the OSC server invokes the methods immediately.

7.3.5 Connection Management

The use of an OSC based protocol for connection management in AVTP capable AVB networks was proposed in [90]. This protocol is known as AVBC. The AVBC protocol proposes an OSC address space, and the use of OSC methods to allow for the remote control of AVTP capable AVB devices.

In AVBC, AVTP source streams are listed under the `/avb/source/…` OSC address, and AVTP sink streams are listed under the `/avb/sink/…` OSC address. Two OSC methods are provided that allow for the dynamic creation of streams. These are `/avb/source/create` and `/avb/sink/create`. These methods accept a number of parameters.

7.3.5.1 Creating Source Streams

The `/avb/source/create` method expects the following OSC arguments:

- The name of the stream: This argument allows human readable names to be assigned to a stream and can be used to refer to the stream at a later stage. For example, this could be set to something like “Movie Audio”.
- Media format: AVTP is able to transport a number of media formats, and this argument allows the format of the stream to be specified. This can allow a hardware device to configure itself to receive the particular stream format, or allow it to reject the stream creation request if it does not support the stream format.
- Presentation: This argument allows for the presentation time offset for a stream to be configured. By default, the presentation time offset for class A AVTP streams is 2ms. This argument allows this to be fine-tuned.
- Number of channels: This argument specifies the number of channels that the stream should contain.
- Media source channel code: The media source channel code argument is a map of a device’s media sources to the channels of the stream. For example, if a device has eight analogue inputs, this argument could map these analogue inputs to the first eight channels of a stream.

When the OSC `/avb/source/create` method is invoked, a stream is created and a stream ID and multicast MAC addresses are allocated to it. The stream is then advertised to the AVB network

via MSRP. The stream is represented in the OSC address space with an addition of a *stream ID* node to the `/avb/source` address. The value of this node is the value of the stream ID associated with the stream. The parameters of the stream are represented under the individual *stream ID* nodes. For example, `mmac`, `state`, `format` and `map` nodes are added as child nodes to each individual *stream ID* node, as shown below.

- `/avb/source/`*`stream ID`*`/mmac`
- `/avb/source/`*`stream ID`*`/state`
- `/avb/source/`*`stream ID`*`/format`
- `/avb/source/`*`stream ID`*`/map`

These nodes address OSC methods that allow for the retrieval of the multicast mac address allocated to the stream, the streaming state of the stream, the format of the media being conveyed in the stream, and the mapping of media sources to stream channels.

The OSC `/avb/source/create` method returns a number of arguments to the caller, including:

- Stream ID: This argument represents the stream ID that was generated for the stream when it was created.
- Multicast MAC address: This argument represents the multicast MAC address that the stream will be transmitted with. This is a multicast MAC address allocated by the MAAP protocol for the stream (see Section 6.3 “AVTP Address Allocation”).

7.3.5.2 Creating Sink Streams

There are three approaches to creating stream sinks:

1. An application is able to specify the stream ID and the multicast MAC address of the stream to attach to.
2. An application is able to specify the stream ID, and the sink device is expected to look up the multicast MAC address of the stream.
3. The controlling application specifies the device and stream name, causing the sink device to lookup the network address of the device, and well as the stream ID and multicast MAC address.

If the first approach of creating a sink stream is used, the `/avb/sink/create` method expects the following OSC arguments:

- Stream ID: The value of this argument represents the source stream ID that the sink stream should attach to.
- Multicast MAC address: This argument represents the multicast MAC address that the source stream will be transmitted with. When MSRP is making use of *talker pruning* (see Section 3.2.4.2 “Talkers Advertising Streams”), a listener device needs this value in order to request membership to the multicast group.
- Presentation: The value of this parameter specifies the presentation time offset for the stream. This allows for the fine-tuning of this value.
- Number of channels: The value of this argument informs the sink device of the number of channels in the source stream. This allows the device to prepare for the reception of the stream.
- Media sink source code: The value of this parameter is a map of incoming stream channels to media sinks within a device. If, for example, a device contains eight analogue outputs, the incoming stream channels could be mapped to these analogue outputs.

This method returns a number of arguments to the caller, including a *sink ID*. The value of this argument is a unique identifier generated by the sink device for the sink stream. It can be used later to refer to the stream.

When a sink stream is created using the `/avb/sink/create` method, the sink device will request attachment to the source stream via MSRP by declaring a *listener* attribute for the stream. The created sink stream is represented in the OSC address space with an addition of a *stream ID* node to the `/avb/sink` address. The value of this node is the value of the stream ID associated with the stream. The parameters of the sink stream are represented under the individual *stream ID* nodes. For example, `mmac`, `state`, `format` and `map` nodes are added as child nodes to each individual *stream ID* node, as shown below.

- `/avb/source/`*`stream ID`*`/mmac`
- `/avb/source/`*`stream ID`*`/state`
- `/avb/source/`*`stream ID`*`/format`
- `/avb/source/`*`stream ID`*`/map`

As with source streams, these nodes address OSC methods that allow for the retrieval of the multicast mac address allocated to the stream, the streaming state of the stream, the format of the media being conveyed in the stream, and a map of stream channels to media sinks.

7.3.5.3 Destroying Streams

Each source and sink stream has a *destroy* method. For a source stream, this would be accessed with the following OSC address `/avb/source/`*`stream ID`*`/destroy`. When the *destroy* method is invoked, the stream associated with it is torn down. For a source stream, a request is made to MSRP to withdraw the *talker advertise* attribute for the stream. For a sink stream, a request is made to MSRP to withdraw the *listener* attribute for the stream.

7.3.6 Tools

Programming libraries exist which allow for the development of OSC based clients and servers. oscpack [91], for example, provides a set of C++ classes for packing and unpacking OSC packets. It also provides the ability to transport OSC messages using the UDP/IP protocol.

7.3.7 Conclusion

OSC uses a hierarchical addressing scheme that allows methods on remote device to be invoked. In essence, the representation of streams is similar to those protocols based on SNMP. Each stream can be represented using a table abstraction where streams are represented in the address hierarchy with a unique address (representing a row in a table), and the properties of each stream are addressed under that (each one representing a field in a row). In AVBC, the mapping of stream channels to internal media sources and sinks is performed with a bitmap, and thus is less verbose than using a table. Alternatively, the hierarchical addressing scheme of the protocol allows for table abstractions to be constructed allowing for the representation of more explicit patching matrices.

7.4 XFN

The XFN protocol [59] [60] is an IP [61] based peer-to-peer command and control protocol that makes use of UDP [76]. Any device on a network that participates in the protocol may initiate or accept control, monitoring and connection management commands from any of its peers. The protocol incorporates three core concepts:

- Structuring: Each parameter in an XFN system is part of a hierarchical structure, and any level of this structure may be addressed to allow for addressing of multiple parameters simultaneously.
- Joining and grouping: Disparate parameters may be joined into groups allowing for control over these parameters from single control sources.
- Indexing: Every parameter has an index value associated with it, and it may be addressed via this index value. This mechanism has been put in place to lower network bandwidth consumption.

Due to the fact that the XFN protocol is based on UDP/IP, it has been implemented independent of any networking architecture. This allows for this protocol to work across multiple networking architectures without any changes.

7.4.1 Structuring

Multi-level addresses were created for a number of professional audio-related devices, and it was found that most parameters conform to a similar hierarchical structure. A device tends to be structured as a series of functional groupings, with the parameters of a device positioned at the lowest level of these groupings. For example, as shown in Figure 96, an audio mixing desk may be structured as follows:

- The audio mixing desk has an input block that is composed of number of signal inputs. These signal inputs may be analogue or digital in nature (number 1 in the figure).
- These signal inputs may then be patched to the channels of the audio mixing desk (number 2 in the figure).
- Each of the audio mixing desks channels may have a number of signal processing components (for example, equalisers and dynamics processors) associated with them. Each one of these signal processing components is composed of a number of adjustable parameters (number 3 and 4 in the figure).

- The signals from each channel may be patched onto the various buses that the audio mixing desk has (number 5 and 6 in the figure).
- Signals from buses may be patched through to the output sections. The output section may be composed of further signal processing components, and each of these components may be further composed of adjustable parameters (number 7 in the figure).
- The signals from the output sections may be patched through to the various outputs that the audio mixing desk has to offer (number 8 in the figure).
- The audio mixing desk has a number of analogue and digital signal outputs (number 9 in the figure).

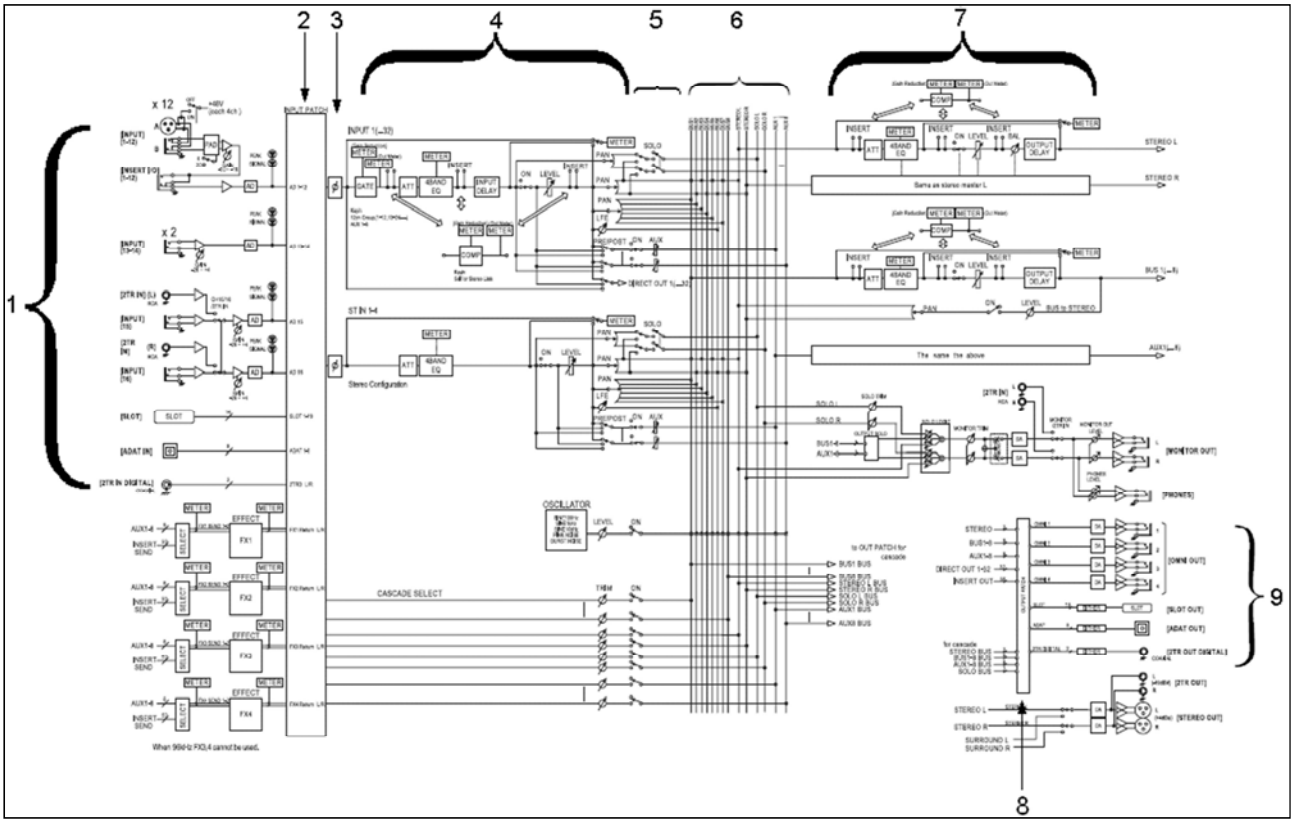

Figure 96: The block diagram for the Yamaha 01V96 Digital Mixing Console [10]

Based on the hierarchical structuring of devices, it was decided that every parameter in an XFN system be addressed via a fixed seven-level address. These addresses intuitively reflect the hierarchical structure of a device. Each level in the hierarchy has a unique value assigned to it. Associated with each one of these values is an alias used to describe the level's functionality. This hierarchical addressing scheme is shown in Table 37. Also shown are the hierarchical address aliases that would be used to describe an equalisation parameter of an audio mixing desk.

Address level	Address level name	Example level alias
1	Section block	Input block
2	Section type	Analogue inputs
3	Section number	Analogue input 1
4	Parameter block	Equaliser
5	Parameter block Index	Equalisation band 1
6	Parameter type	Equalisation parameter
7	Parameter index	Equalisation parameter 1

Table 37: The XFN seven-level addressing scheme

An XFN address consists of the following address hierarchy:

1. *Section block*: A device is usually comprised of a number of sections. An audio mixing desk has an input section, output section, and various patching matrix sections, for example. This part of an XFN address is used to identify the section in which the parameter resides.
2. *Section type*: Every section of a device has a type, and may be composed of many types. An input section of an audio mixing desk may be composed of an analogue input section type and a digital input section type. This part of an XFN address is used to identify the section type of the section block in which the parameter resides.
3. *Section number*: The section number is used to identify a specific instance of the section type. An analogue input section type may be composed of a number of analogue inputs. The section number is used such that each one of these may be individually addressed.
4. *Parameter block*: A number of parameters may be grouped together. An equalizer is a signal processing component that is composed of a number of sections and parameters. This part of an XFN address is used to identify a specific parameter block of the section number.
5. *Parameter block index*: A parameter block may be composed of a number of sections. For example, an equaliser may have low, low-mid, high-mid, and high sections. This part of the XFN address is used to identify the sub-section of a parameter block.
6. *Parameter type*: Each one of the sections of a parameter block may contain a number of parameter types. For example, each one of the sub sections of an equaliser parameter block may contain a Q, frequency, and gain parameter. This part of the XFN address is used to identify the specific parameter type being addressed.

7. *Parameter index*: Each parameter block may contain a number of parameters of the same type. The parameter index section of an XFN address is used to address these individual parameters. A channel of an audio mixing desk may, for example, contain multiple gain parameters.

Address level identifiers are defined within the XFN specification. New devices describing existing parameter types (e.g., gain parameters) use the same addresses used by existing devices. This allows new devices to be easily integrated into existing networks.

A device that participates in the XFN protocol builds up an internal seven-level tree structure that reflects the actual device. Each of the leaf nodes of the tree is associated with a user supplied callback function that allows the device to respond to commands from its peers (see Section 7.4.2 “Messaging”). Figure 97 shows an example XFN stack with a built up tree structure. Each level in the tree structure has a unique address (and an address alias) used to identify it. The leaf nodes of the tree structure are associated with user supplied callback functions for a specific XFN application.

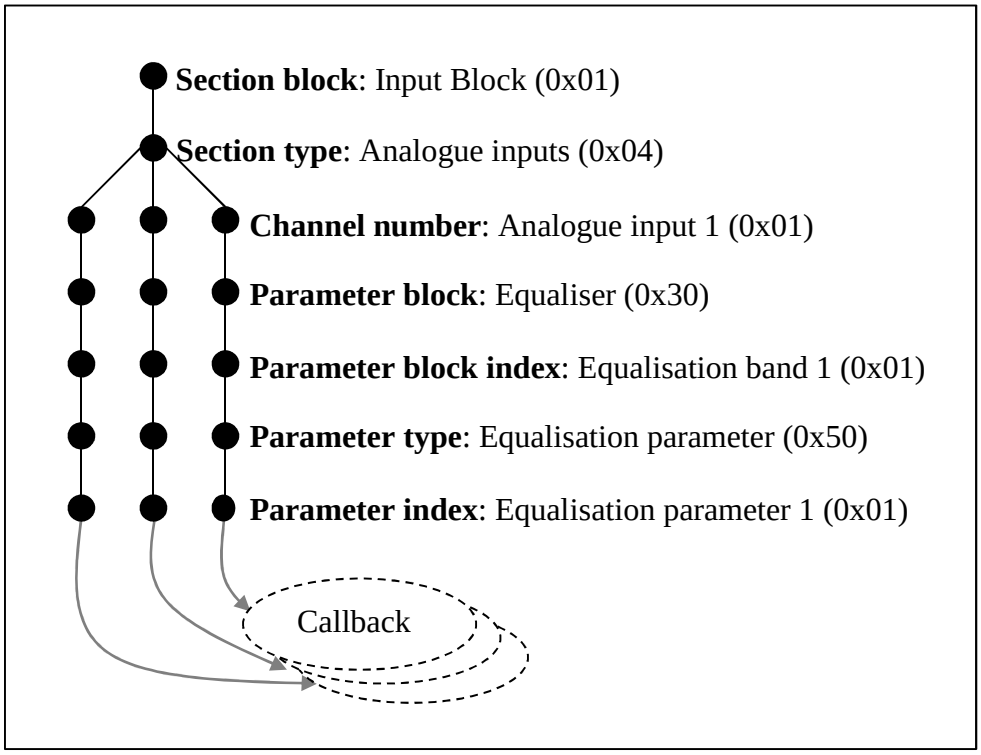

Figure 97: An example hierarchical address for an equalisation parameter

Some devices may not need all seven levels to address their parameters. This may be the case for simple devices with a few high-level parameters. When this is the case, a *dummy* value is placed into the address level in place of a standard level identifier.

7.4.2 Messaging

Any device on an XFN network is allowed to transmit and receive XFN messages to and from its peers. Each message has a *message type*, *command executive*, and a *command qualifier*. Amongst other things, the *message type* field is used to distinguish between request and response messages. A request message is sent from one device to another to request it to perform some action, and a response message is sent in response to a request message. Not all request messages require a response message.

The specific action of a request message is indicated in the *command executive* and *command qualifier* fields. The *command executive* field indicates the fundamental nature of the message, and the *command qualifier* field allows the *command executive* to be directed to a certain attribute of a parameter (a parameter contains a number of attributes, such as a value, name, and flags). The two most common command executives are *get* and *set*, and the most common command qualifier is *value*. In order to obtain the value of a parameter on a second device, a device will send a request message to that device specifying a *get value* command, and will specify the seven-level address of the parameter. The device that receives the request will respond to the requester with a response message containing the value of the parameter. In order to set the value of a parameter on a second device, a device will send a request message to the device specifying a *set value* command, the seven-level address of the parameter, and the value that the parameter should be set to.

Figure 98 shows how an incoming XFN message is processed. This message contains the seven-level address of a parameter (Parameter address), the action to perform on that parameter (Command), and a value (the value field is used for *set value* commands). When a device receives an XFN message from another device, the parameter that is identified by the XFN seven-level address is located (indicated by the number 2). Certain commands (such as *get value* and *set value*) require further action by the application using the XFN protocol. If one of these commands is indicated in the message, the application-supplied callback is called (indicated by the number 3) and is requested to perform the required action (indicated by the number 4). For example, if a *get value* command is

specified to the callback, the callback returns the current value of the parameter to the caller. If a *set value* command is specified to the callback, the callback sets the value of the parameter to the supplied value.

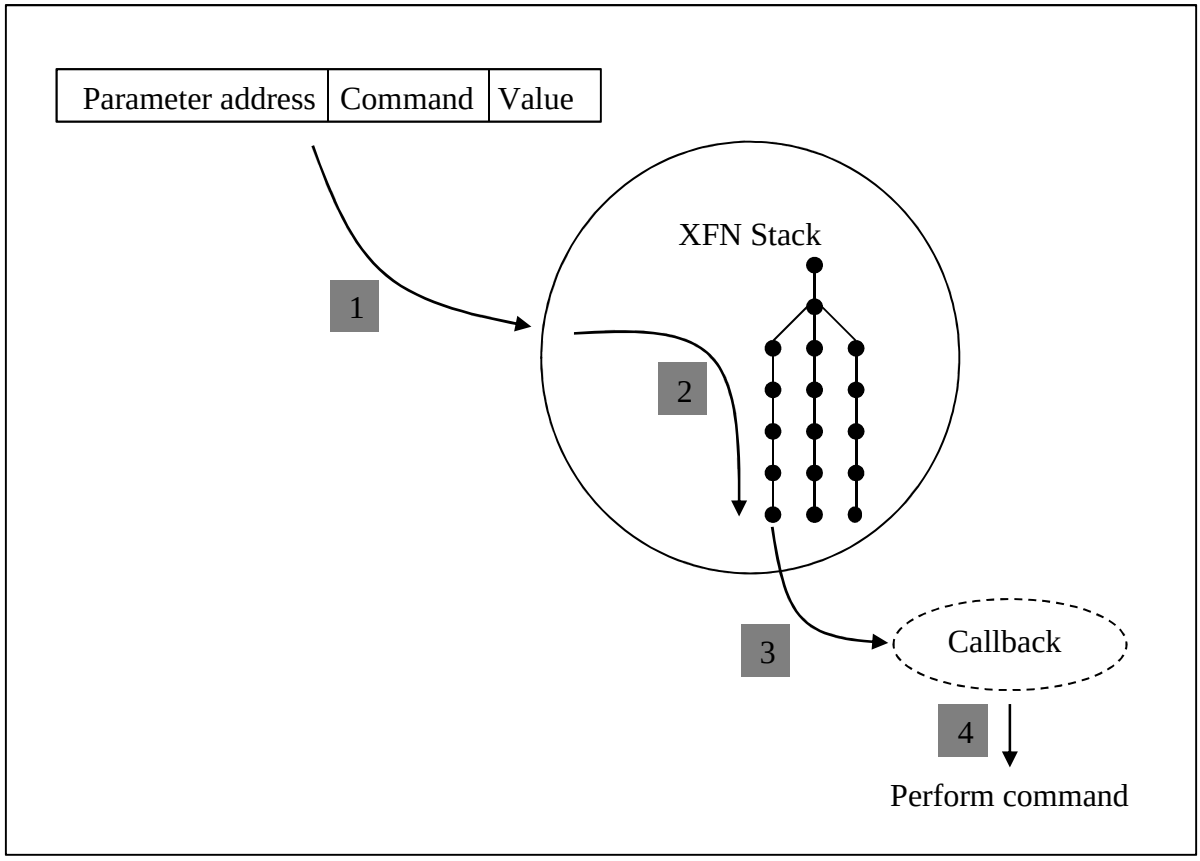

Figure 98: XFN message processing

When a value is transported in an XFN message, it has associated with it a *value format* specified in a *value format* field. The *value format* field is an integer field that indicates the type and length of the value.

XFN messages may be broadcast or multicast onto a network. This allows single messages to control parameters on multiple devices.

7.4.3 Indexing

XFN also allows for parameters to be addressed via a unique ID (as opposed to a seven-level address). Every parameter in an XFN device has associated with it a unique ID, and this unique ID may be obtained for a parameter by addressing it with its seven-level address and requesting the ID. Subsequent messages to the device may contain the unique ID for the parameter. This mechanism has been put in place to reduce network bandwidth usage.

7.4.4 Wildcarding

Parameters may be addressed individually using a full seven-level address, or groups of parameters on a device may be addressed by replacing address level values with wildcards. If, for example, a controller would like to adjust all of the gain parameters of a particular channel, it would send a *set value* message to the device and wildcard the parameter index part of the XFN address. The device will then cycle through all of the parameter's indexes and set the value of each parameter (via the callback function). Wildcards may appear at more than one level. If, for example, a controller would like to adjust all of the gain parameters of all of the channels of a device, it will wildcard the section number and the parameter index portions of an XFN seven-level address. A receiving device will then cycle through each channel, and for each channel will cycle through each gain parameter and set its value. A wildcard replaces the level identifier value with a value that has all of its bits set to one.

XFN messages that make use of the wildcard mechanism may also use broadcast and multicast transmission mechanisms. This allows multiple parameters on multiple devices to all be addressed with a single message.

7.4.5 Pushing

When a device would like to know the value of a parameter, it is able to request the value of that parameter via a *get value* command. It may however become inefficient to consistently poll for the value of a parameter when it is desirable to frequently monitor the value of a parameter. This is true of level metering parameters, for example. To alleviate this, the XFN protocol has the concept of a *push* mechanism. A requesting device is able to instruct another device to tell it whenever one of its parameter values change by sending it a *set push* command for the parameter. A requesting device

may also instruct another device to stop informing it when the parameter value changes by sending it a *set push off* message for the parameter.

Some parameter values may change at a high frequency, and it may be undesirable or unnecessary to be notified each time that this happens. XFN has mechanisms that allow updates to be sent at a regular interval. If this functionality is enabled, at each interval, if a parameter's value has changed, it sends a notification of the change in value. It is also possible to instruct a device to send updates only once a parameter's value has change by a given delta.

A requesting device has to continually register its continued interest in receiving updates on parameter value changes. If, after a timeout time, a device does not receive notification that a requesting device is still interested in receiving parameter value updates, it will cease sending updates. This mechanism is in place to guard against devices losing network connectivity before withdrawing their interest in parameter updates.

7.4.6 Joining and Grouping

Disparate parameters may be joined together into groups. Grouped parameters may exist on the same device, or the parameters may be on difference devices. This mechanism allows groups of parameters to be controlled via single control messages. For example, two audio mixing desks may exist on a network, and a fader of the one audio mixing desk may be joined to the fader of another audio mixing desk. When one of the faders is moved, it will result in the joined fader on the second audio mixing desk being moved.

There are a number of relationships that may exist between parameters that are joined into groups. These are master/slave, and peer-to-peer. These relationships may then be further described as absolute or relative.

In a master/slave relationship, a single parameter in the group is a master parameter, and all of the other parameters in the group are slave parameters. The master parameter has control over the slave parameters. For instance, when the value of the master parameter is adjusted, the values of the slave parameters are adjusted. However, when the value of a slave parameter is adjusted, the master is not

influenced. In a peer-to-peer relationship, whenever any of the parameters that are part of the group are modified, the other members of the group are also modified.

When parameters have an absolute relationship, and a parameter of the group is adjusted, its slaves or peers (depending on the relationship that exists between the parameters) take on the same value as the parameter. When parameters have a relative relationship, and a parameter of a group is adjusted, its slaves or peers (depending on the relationship that exists between the parameters) take on a value that is an offset to the slave's or peer's current value.

7.4.6.1 Example

Assume that three faders on an audio mixing desk exist, each one controlling a gain parameter. The gain parameters can take on the values 0 through 20. These faders are shown in Figure 99, and are labelled A, B, and C and currently have the values 12, 5, and 17 respectively.

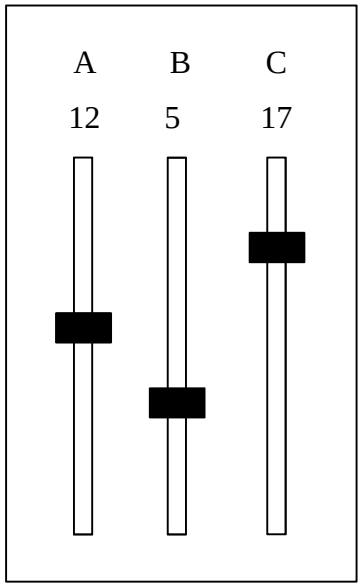

Figure 99: Three faders with initial values

If these gain parameters are grouped together into an absolute master/slave relationship (with gain A the master), and fader A is adjusted such that it takes on the value 14, its slaves will also take on the value 14, as shown in Figure 100. If fader B of the master/slave group is adjusted, it will not have any influence over any of the other parameters in the group.

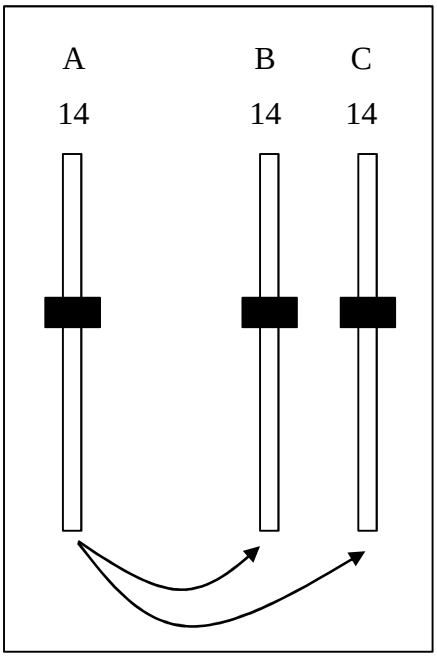

Figure 100: Absolute master/slave relationship, master adjusted

Assume that the faders in the example are set to the initial values shown in Figure 99 and they are joined into a relative peer-to-peer group. If fader A is adjusted to a value of 14, faders B and C will be adjusted by a value of 2, as shown in Figure 101. Similarly, if fader B is adjusted to a value of 4, faders A and C will be adjusted by a value of -3.

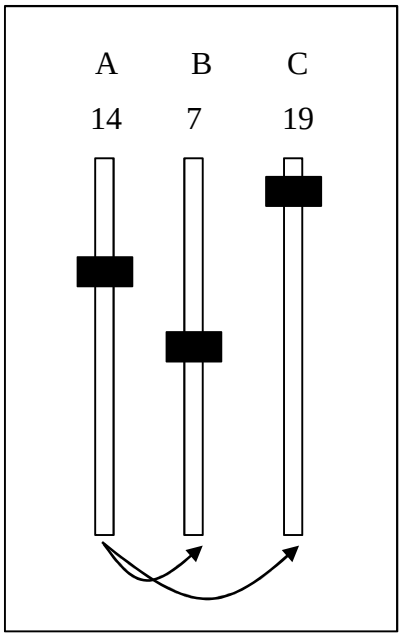

Figure 101: Relative peer-to-peer relationship, fader A adjusted

7.4.7 Modifiers

The XFN specification incorporates a *modifier* mechanism. This mechanism allows a device to adjust values that it transmits and receives, allows for the modification of seven-level address blocks, and allows for the modification of the timing of a message. Modifiers are classified as:

- *Value modifiers*: This modifier type allows a transmitted or received value to be modified. For example, it may be desirable to have one fader reflect the inverse value of a second fader. In this example, the two parameters need to be joined together, and the values received from each other need to be modified such that the values are inverted.
- *Level modifiers*: This modifier type allow for the modification of transmitted or received XFN addresses. For example, it may be useful to be able to use a single control (for example, a fader) on a small device to control a number of parameters on another larger device by selecting the channel that it has control over. Before any messages are transmitted by the controlling device, its address block is modified to reflect the channel that it currently has control over.
- *Event modifiers*: An event modifier type allows the timing of a message to be adjusted. During parameter adjustment automation, received messages need to be stored and only interpreted at a later time. This modifier allows for this to take place.

7.5 Conclusion

There are a number of command and control protocols available that allow for remote control over networked audio devices. From a high-level perspective, all of the protocols discussed above provide the ability to address unique parameters on a device, and allow for the access and modification of parameter values. The various protocols all provide a form of hierarchical parameter addressing which allows the natural hierarchical structure of a device to be modelled. Conceptual tables can be formed with their hierarchical addressing schemes, which allows for a convenient representation of patching matrices for connection management internal to devices. The table representation also allows for stream sources and sinks (and their properties) to be conveniently represented.

The XFN protocol was selected for use in this study as it was under development as an *Audio Engineering Society* (AES) standard [92] at the time that this work was performed. It provides a rich set of features and functionality geared towards networked realtime audio and video equipment.

Source code of an implementation of the XFN protocol was also made available. This source code was designed such that it could be built on Windows platforms, Linux platforms, as well as other platforms. The XFN protocol is easy to use due to its simple design and natural address hierarchy. The API functions provided by the source code are simple to use. They provide the ability to build XFN address hierarchies, and to associate application specific callback functions with the leaves of the hierarchies. All message processing is handled by the provided source code, and responses to commands are dealt with in the application supplied callback functions. This enables minimal application code in order to implement an XFN controllable application.

The XFN specification defines level hierarchies and parameters that allow for connection management to take place within audio devices, and between audio devices. These parameters allow for pre-defined hierarchies to be built allowing for connection management to take place. Chapter 11 “XFN Control and Representation” details how the protocol has been used to perform connection management in both IEEE 1394 and Ethernet AVB devices.

Chapter 8 Tunnelling

The preceding chapters introduced a number of protocols. The work that was carried out in this study is based on the use of a combination of these protocols in order to allow for control (from a single control protocol) over realtime audio devices that operate on disparate networks. The dominant standards-based networking technologies that provide a suitable environment for the transmission of realtime audio data are IEEE 1394 and Ethernet AVB, and were thus used in this study.

As an initial proof-of-concept, an application that allows for Ethernet traffic to be streamed (tunnelled) across IEEE 1394 networks was developed. Due to the incompatible addressing mechanisms and frame formats of each network type, Ethernet traffic has to be encapsulated within IEEE 1394 compatible packets. The tunnelling of Ethernet traffic across these networks allows Ethernet devices to take advantage of pre-existing IEEE 1394 networks for Ethernet frame transmission. Ethernet end stations and Ethernet LANs can be conjoined using the tunnelling application. This tunnelling application allows for remote Ethernet control over device parameters via the XFN protocol. IEEE 1394 natively provides an environment whereby audio and video data may be deterministically transmitted over a digital network. Traditional Ethernet and Ethernet bridging do not provide these mechanisms. Allowing for the transmission of Ethernet frames over IEEE 1394 networks provides a mechanism to allow for deterministic transmission of this data.

8.1 Tunnelling Ethernet Traffic over IEEE 1394

Given in Figure 102 is a sample set up of two IEEE 1394 Ethernet tunnelling devices (*tunnel nodes*). These devices are used as entry and exit points for tunnelling Ethernet frames over IEEE 1394 networks. These devices are used to capture all incoming Ethernet frames that are not addressed to the node itself, and to forward this traffic to a second tunnel node within an isochronous stream. This captured Ethernet traffic is encapsulated within IEEE 1394 isochronous packets and is sent onto the network. The second tunnel node is responsible for receiving these isochronous packets and for extracting the Ethernet frames contained within them. These Ethernet frames are then transmitted out of the second tunnel node’s Ethernet interface. From the perspective of the transmitting and receiving Ethernet devices, the tunnelling mechanisms are transparent. These devices are not aware of the presence of the tunnel nodes.

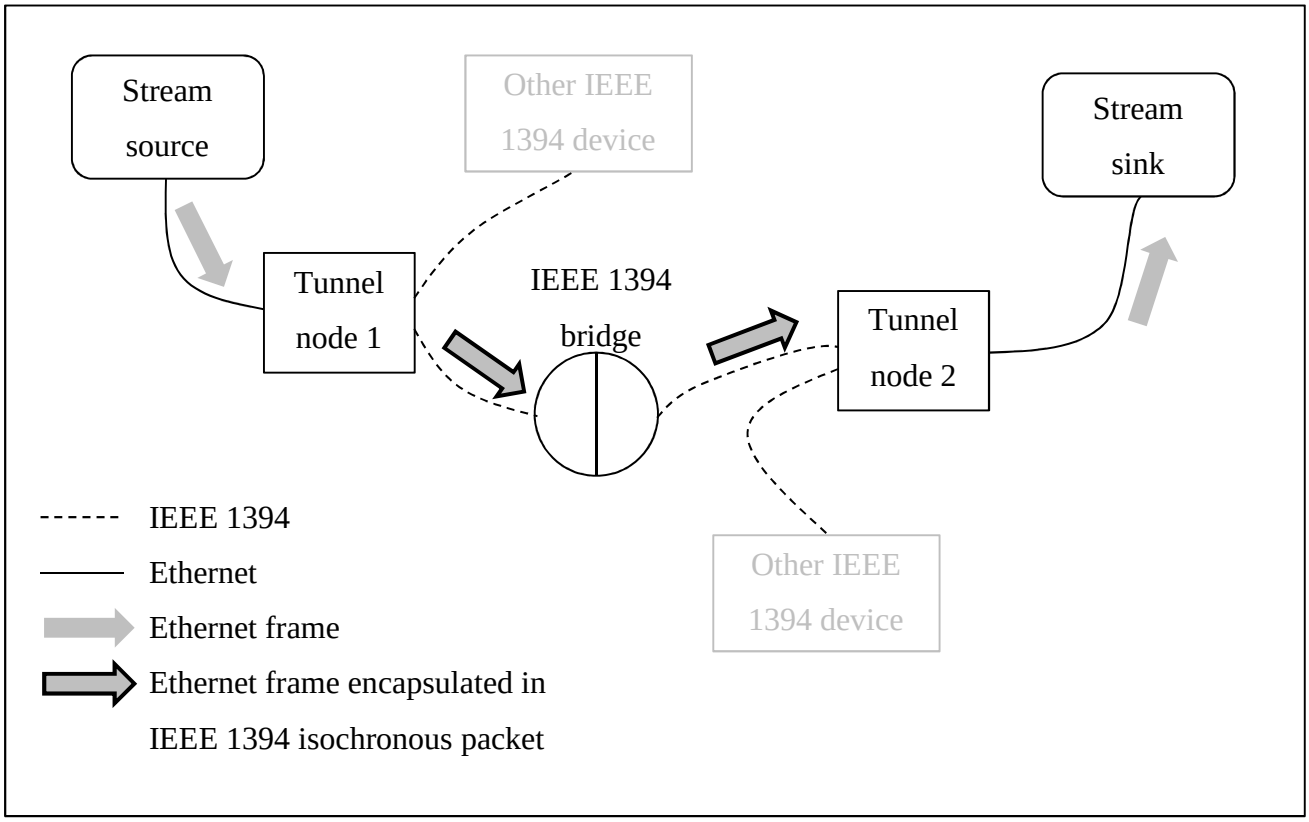

Figure 102: Tunnelling of Ethernet traffic over IEEE 1394

The tunnelling capabilities provided by the tunnel nodes allow for the deterministic transmission and routing of Ethernet frames between Ethernet devices. Devices that connect (via Ethernet) to the tunnel nodes use full-duplex links. This has the implication that the transmitting device is the only device contending for access to the medium, collision detection is not needed, and so it is able to transmit frames at will. Before a tunnel node transmits any isochronous packets, it allocates the resources (bandwidth and an isochronous channel number) needed to tunnel the Ethernet frames over the bus.

In the scenario presented in Figure 102, a stream source is directly attached to the Ethernet interface of the first tunnel node. Directly attached to the Ethernet interface of the second tunnel node is a stream sink. The stream source is able to transmit its Ethernet frames to the first tunnel node. These frames can then be deterministically routed through a complex network of IEEE 1394 nodes until they reach a destination tunnel node. At the second tunnel node, the frames are directly transmitted out of the Ethernet interface to the stream sink. If the stream source and sink were connected together through a traditional bridged Ethernet network, they would have to contend for network resources

with other devices on the network. This could lead to unpredictable frame loss and unacceptable latency causing jitter in stream playback. With the tunnel node application, existing Ethernet devices are able to take advantage of the characteristics of IEEE 1394 networks.

8.2 Limitations

The isochronous bandwidth allocation on an IEEE 1394 bus affects the maximum size that an isochronous packet may be (see Section 3.1 “Resource Reservation for IEEE 1394”). It is possible that insufficient bandwidth has been reserved to cater for Ethernet frames over a certain size. Also, an IEEE 1394 node transmits isochronous packets at a rate of 8000 packets per second. It is possible that a tunnel node receives more than 8000 Ethernet frames per second and thus is not able to transmit all of the frames that it receives onto an IEEE 1394 network.

To counter these limitations, if there is more than one Ethernet frame waiting to be transmitted onto the IEEE 1394 bus, the tunnel node is able to pack multiple Ethernet frames into a single isochronous packet (if there is sufficient space in the packet for this to happen). At the receiving tunnel node, these frames are unpacked and individually sent onto the Ethernet network.

If, on the other hand, an Ethernet frame waiting to be sent onto the IEEE 1394 network is larger than the maximum allowed isochronous packet payload size, the tunnel node will fragment the frame and send the fragments across to the receiving tunnel node. The receiving tunnel node will rebuild the Ethernet frame as it receives the fragments. Once it has a complete Ethernet frame, it will transmit it out of its Ethernet interface.

8.3 Tunnel Header

To communicate the packing and fragmenting of Ethernet frames within isochronous packets, a *tunnel header* was designed. This header is used to communicate whether the packet contains packed Ethernet frames or fragmented Ethernet frames, amongst other information. This header appears before each Ethernet frame. If the isochronous packet contains packed Ethernet frames, then the header appears before each Ethernet frame. If an isochronous packet contains an Ethernet frame fragment, then the header will appear before the fragment. Figure 103 shows the structure of an

isochronous packet containing a tunnel header and an Ethernet frame fragment, as well as an isochronous packet containing multiple tunnel headers for multiple packed Ethernet frames.

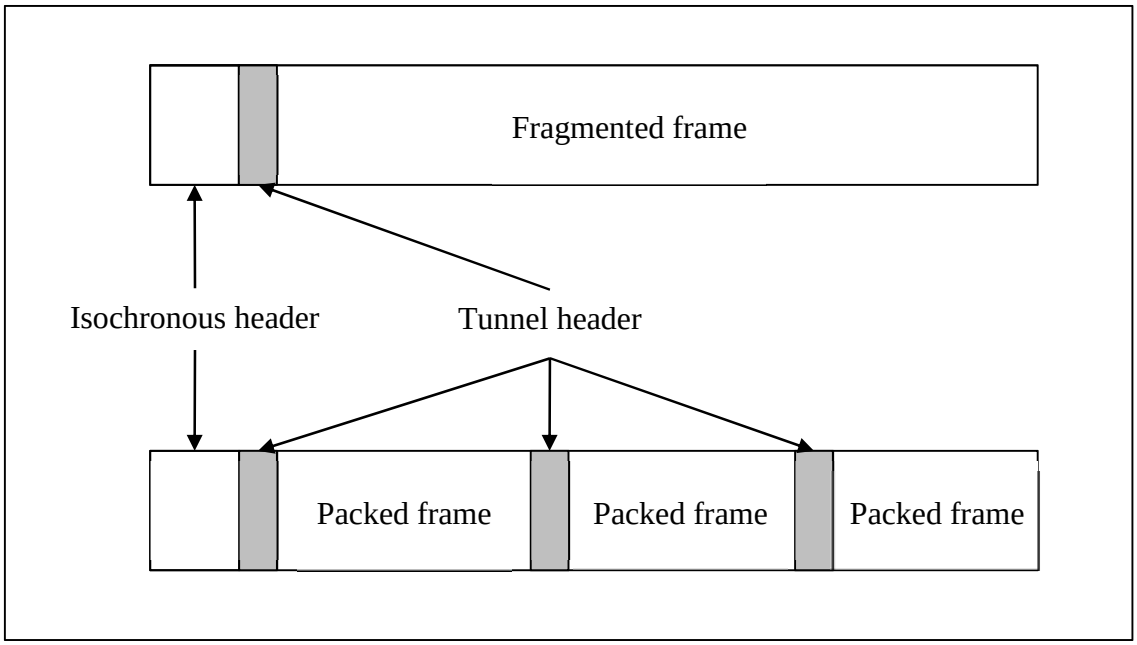

Figure 103: Tunnel node header positions

Figure 104 shows the structure of the tunnel header used for packed Ethernet frames.

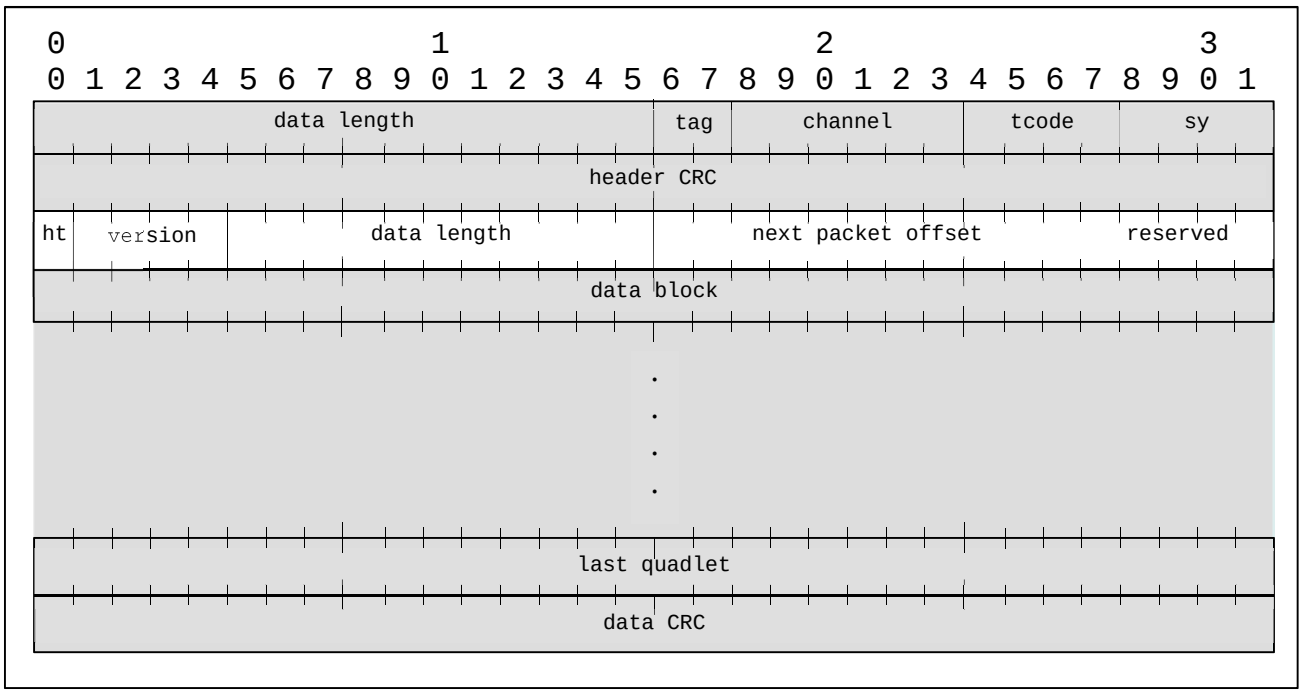

Figure 104: An isochronous packet with a tunnel node header for packed Ethernet frames

The fields of the tunnel header used for packed Ethernet frames are defined in Table 38.

Field	Definition
Header type (ht)	The *header type* field is used with packed Ethernet frames and with fragmented Ethernet frames. It is used to distinguish between a tunnel header appearing before a packed Ethernet frame, and a tunnel header appearing before a fragmented Ethernet frame. A value of one indicates packed, and a value of zero indicates fragmented.
Version	The *version* field is used with packed Ethernet frames and with fragmented Ethernet frames. It is used to identify the version of the tunnelling protocol. Currently only one version is defined, and its value is set to one.
Data length	The *data length* field is used to specify the total length of the Ethernet frame (including the header and the body) that is under the tunnel header.
Next packet offset	The *next packet offset* field is used to specify the offset (in octets) from the beginning of this tunnel header to where the next tunnel header can be found. A value of zero indicates that there are no more tunnel headers in the packet (and hence no more Ethernet frames).
Reserved	The *reserved* field is reserved for future use.

Table 38: Tunnel header fields for a packed Ethernet frame

Figure 105 shows the structure of the tunnel header used for fragmented Ethernet frames.

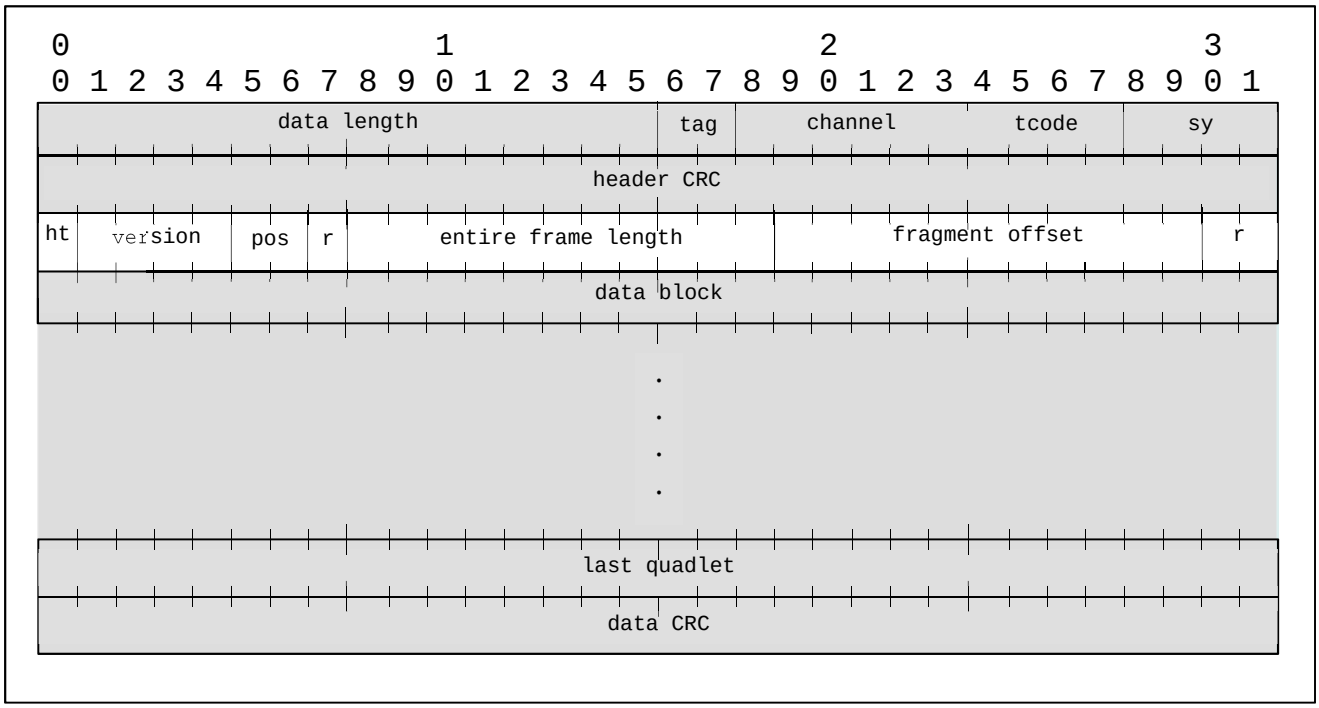

Figure 105: An isochronous packet with a tunnel node header for fragmented Ethernet frames

The fields of the tunnel header used for fragmented Ethernet frames are defined in Table 39.

Field	Definition
Header type (ht)	This has the same definition as for packed Ethernet frames.
Version	This has the same definition as for packed Ethernet frames.
Position (pos)	The value of the *position* field is used to identify the position of the fragment in relation to the Ethernet frame which the fragment is part of: • A value of one indicates that the fragment is the first fragment in the series of fragments. • A value of two indicates that the fragment is the last fragment in the series of fragments. • A value of three indicates that the fragment is an interior fragment (i.e., it is in between the first and last fragments).
Reserved (r)	The *reserved* field is reserved for future use.
Entire frame length	The value of the *entire frame length* field is used to communicate the length of the Ethernet frame from which the fragment came.
Fragment offset	The *fragment offset* is the offset of the first octet of the fragment relative to the first octet of the Ethernet frame from which the fragment came.
Reserved (r)	The *reserved* field is reserved for future use.

Table 39: Tunnel header fields for a fragmented Ethernet frame

When the *position* field indicates that the fragment underneath the tunnel header is the first fragment in a series of fragments, it allows a receiver to prepare for receiving a frame over a few isochronous packet receptions. It allows the receiver to prepare a buffer, and instructs it to not transmit the contents of the buffer until all of the fragments have been received. When the *position* field indicates that the fragment underneath the tunnel header is an interior fragment, this indicates to the receiver to place the received fragment within the previously allocated buffer. When the *position* field indicates that the fragment is the last fragment of a series of fragments, it knows that it will not receive any more fragments for this frame. Using the *entire frame length* value, the *fragment offset* values of the received fragments, and the *data length* values (of the isochronous header) of the received fragments, the receiver is able to determine whether or not it has received an entire Ethernet frame. Once the receiver has received an entire Ethernet frame, it can prepare it for transmission out of its Ethernet interface.

8.4 Providing Control over the Tunnel Nodes

The tunnel nodes provide mechanisms for tunnelling Ethernet frames in isochronous packets across IEEE 1394 networks between IEEE 1394 nodes. There was a need to provide remote control over these mechanisms and as such parameters were created to enable this control. It was necessary to provide parameters to allow for isochronous stream establishment between tunnel nodes, and to allow for control over the maximum isochronous payload size (and hence isochronous bandwidth allocation).

8.4.1 Connection Establishment

On an IEEE 1394 bus, isochronous packets are broadcast onto the bus on a particular isochronous channel (which is identified by an isochronous channel number). An IEEE 1394 node that is interested in receiving these stream packets is configured to receive the packets on that channel. In order to establish a connection between two nodes, the receiving device needs to know the isochronous channel number of the stream it should receive. It should be possible to instruct transmitting devices to start and stop streaming, and it should be possible to instruct receiving devices to start and stop receiving a stream.

8.4.2 Maximum Payload Size

The maximum size of an isochronous packet is defined by the transmission speed of the IEEE 1394 bus and by the amount of bandwidth that has been reserved for the particular stream. Given the variable nature of Ethernet frame sizes and transmission frequency, it was desirable to allow configuration of the maximum payload size of the isochronous packets to allow for optimal bandwidth allocation and usage.

8.4.3 Tunnel Node Parameter Control with XFN

To allow for control over the tunnel nodes, the XFN protocol was built into the devices and an XFN address hierarchy was developed to allow for control over their parameters. Each tunnel node builds

up the XFN address hierarchy to represent the hierarchical structure of the device. This allows the parameters of the devices to be addressed such that their values can be remotely obtained and set.

Figure 106 shows how an XFN stack is associated with the tunnelling component of the tunnel node. The tunnelling component of the tunnel node is responsible for the reception and transmission of Ethernet frames and isochronous packets, and for frame fragmentation and packing. This component has parameters that allow the transmission and reception isochronous channels, as well as the transmission isochronous packet sizes to be obtained and set. The XFN stack builds up a tree structure representing the tunnel node's structure, and the leaves of this tree structure (which represent the parameters) point to user defined callbacks within the tunnelling component. These callbacks are responsible for enabling the tunnelling within the tunnelling component of the tunnel nodes.

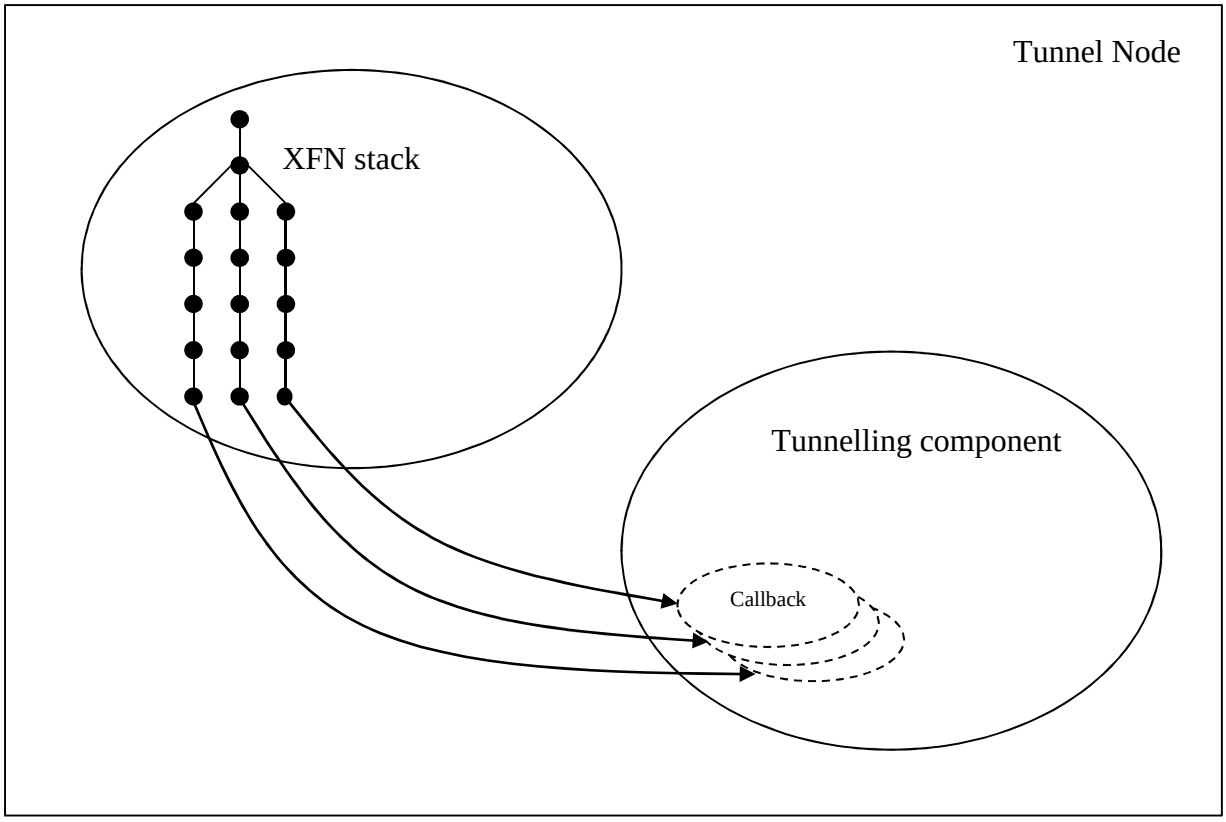

Figure 106: A tunnel node with an XFN stack

Figure 107 shows a portion of the XFN address hierarchy that is built up to represent a transmission isochronous stream of a tunnel node. In XFN, IEEE 1394 and Ethernet AVB streams are known as *multicores*. As this stream is transmitted from a tunnel node, the parameters are located within

stream section type of the *signal output section block*. All of the stream output multicore parameters are located within the *multicore parameter block*. Each *multicore parameter block* is indexed with a unique *parameter block index* value. Within each unique *parameter block index* are listed the parameters associated with each multicore:

- The *isochronous channel number* parameter is used to represent the isochronous channel that the stream will be, or is being, transmitted on. The value of this parameter may be obtained and set by a remote device.
- The *start* parameter is a Boolean parameter that is used to start and stop the transmission of the stream. The value of this parameter may be obtained and set by a remote device. Reading a value of one from this parameter indicates that the stream is being transmitted, and writing a value of one starts stream transmission (if stream transmission is not already started). Reading a value of zero indicates that the stream is not being transmitted, and writing a value of zero to this parameter stops stream transmission (if the stream is being transmitted).
- The *max payload size* parameter represents the maximum payload size that a transmitted isochronous packet may have. The value of this parameter may be obtained and set by a remote application. The value of this parameter affects a tunnel node's ability to either pack multiple Ethernet frames into single isochronous packets, or to fragment single Ethernet packets into multiple isochronous packets.

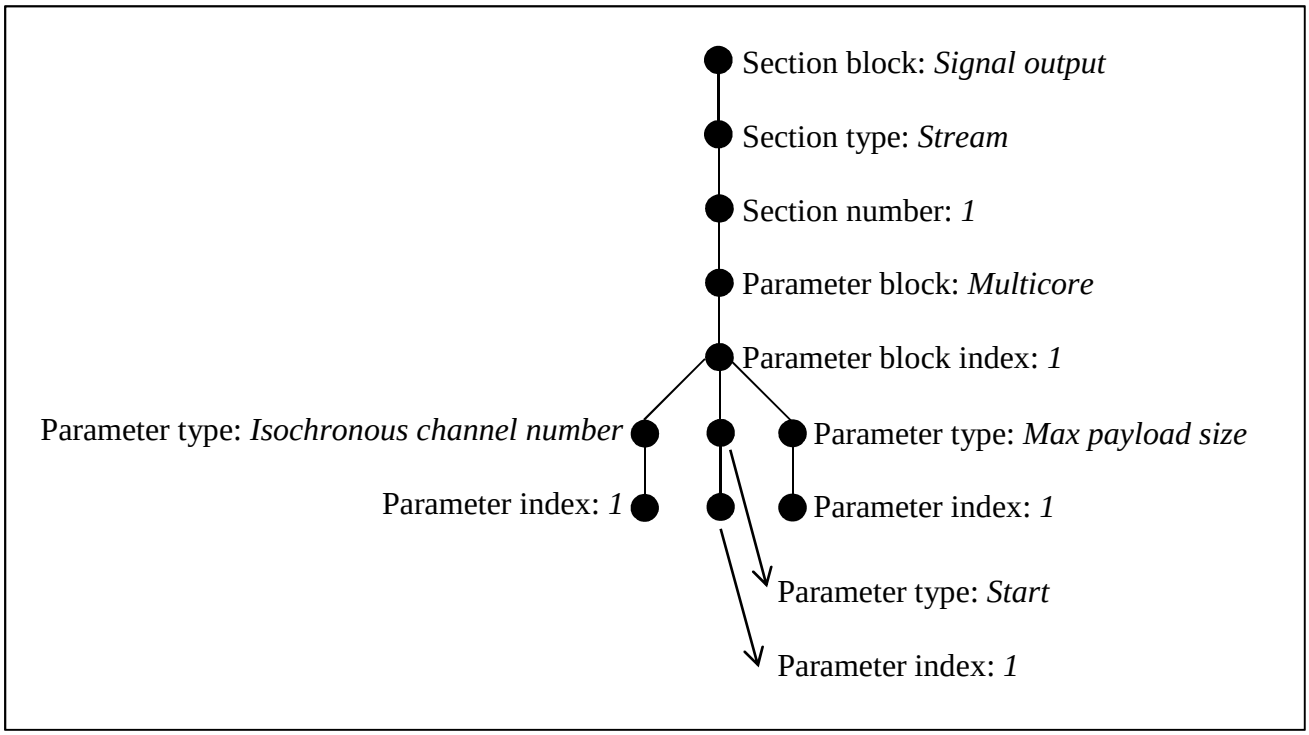

Figure 107: Portion of the XFN address hierarchy for an output isochronous stream

Figure 108 shows a portion of the XFN address hierarchy that is built up to represent a reception multicore of a tunnel node. This structure is almost identical to the structure shown in Figure 107 (for an output multicore) except that input multicores are represented under the *signal input section block*. An input multicore of a tunnel node has the following parameters:

- The *isochronous channel number* parameter is used to represent the isochronous channel number of the stream that the tunnel node is receiving, or should receive. It is possible to remotely obtain and set the value of this parameter.
- The *start* parameter is used to start and stop the reception of the isochronous stream. It is a Boolean parameter whose value may be obtained and set remotely. Reading a value of one from this parameter indicates that the tunnel node is configured to receive an isochronous stream, and writing a value of one to this parameter configures the tunnel node to start the reception of a stream (if it is not already receiving a stream). Reading a value of zero from this parameter indicates that the tunnel node is not configured to receive a stream, and writing a value of zero to this parameter configures it stop the reception of the stream (if it is already receiving a stream).

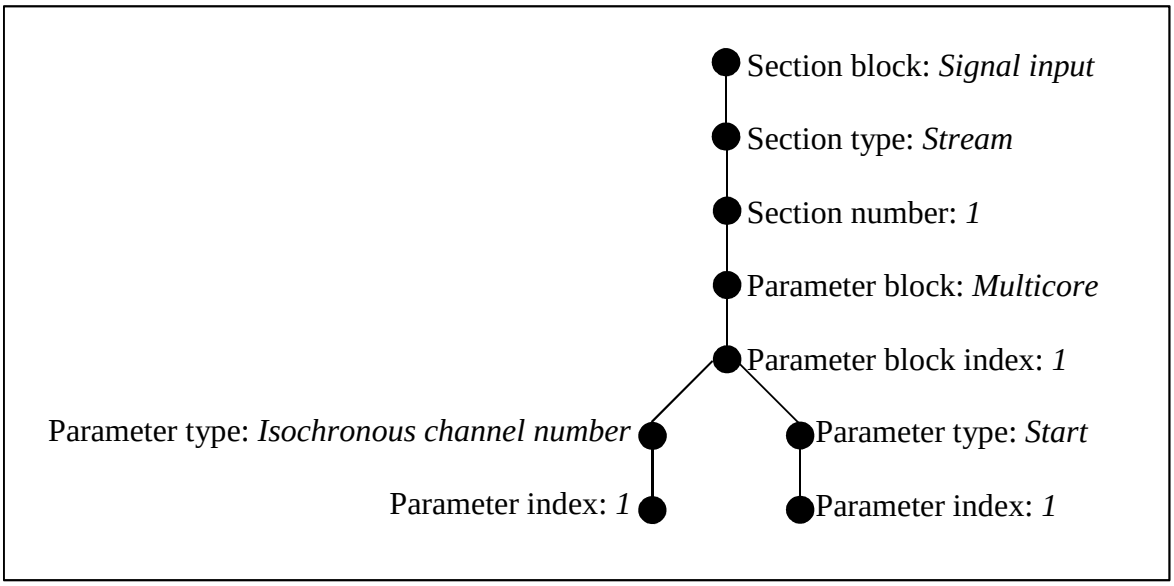

Figure 108: Portion of the XFN address hierarchy for an input isochronous stream

8.4.4 Tunnel Node Parameter Control with the Connection Manager

A desktop application (known as the *Connection Manager*) was developed to allow the tunnel nodes (and their parameters) to be remotely represented and adjusted. The Connection Manager application uses an XFN stack in order to communicate with the XFN stack of a tunnel node. The Connection Manager was developed using JUCE [93]. JUCE is a C++ cross-platform class library that allows for the development of desktop applications, amongst other things.

Each tunnel node has parameters that represent the IP addresses bound to its network interfaces, as shown in Figure 109. The built up XFN address hierarchy contains a *configuration section block* under which are listed the stream and non-stream interfaces of a device. Under each one of the interface types is an *IP parameter block* which is used to represent each interface's IP related parameters. An IP address is represented with the *IP address parameter type*.

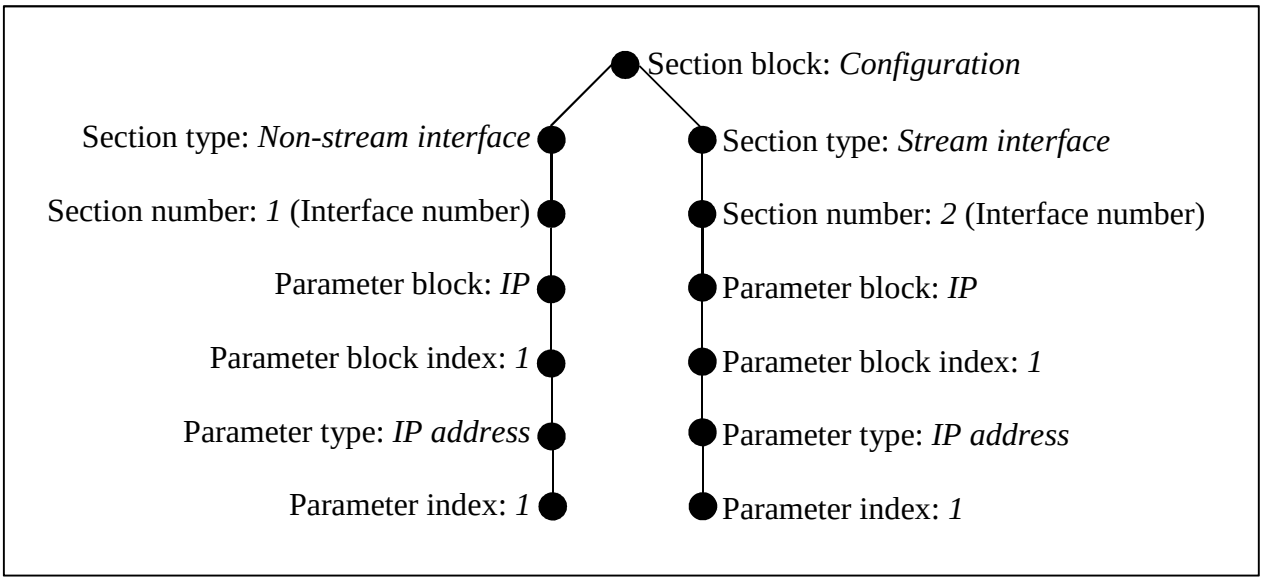

Figure 109: Portion of the XFN address hierarchy for representing IP addresses

When an XFN device would like to discover other XFN devices on a network, it sends out a broadcast packet containing an XFN *get value* command for all of the values of the *IP address parameter types*. The address hierarchies in the messages may contain a wildcard at each of the *section type* and *section number* levels to indicate that all of the IP addresses of all of the interfaces should have their IP addresses returned to the requester. Each device responds with its IP address. Once the requesting device knows the IP address of any other XFN device on a network, it is able to communicate with that device directly using its IP address.

When the Connection Manager application starts up, it discovers all of the XFN devices that exist on the network. As shown in Figure 110, the main display of the application represents the discovered devices along the axes of a grid. On this grid, the devices along the left are viewed as source devices (devices that produce streams). The devices along the top of the grid are viewed as destination devices (devices that consume streams). Each device that is discovered is able to both produce streams, and consume streams, thus each device is shown on the left hand side and at the top of the grid. The buttons between the devices allow for connections between the various devices to be established. In this figure, the Connection Manager is displaying two tunnel nodes (a typical network will usually consist of many different types of devices).

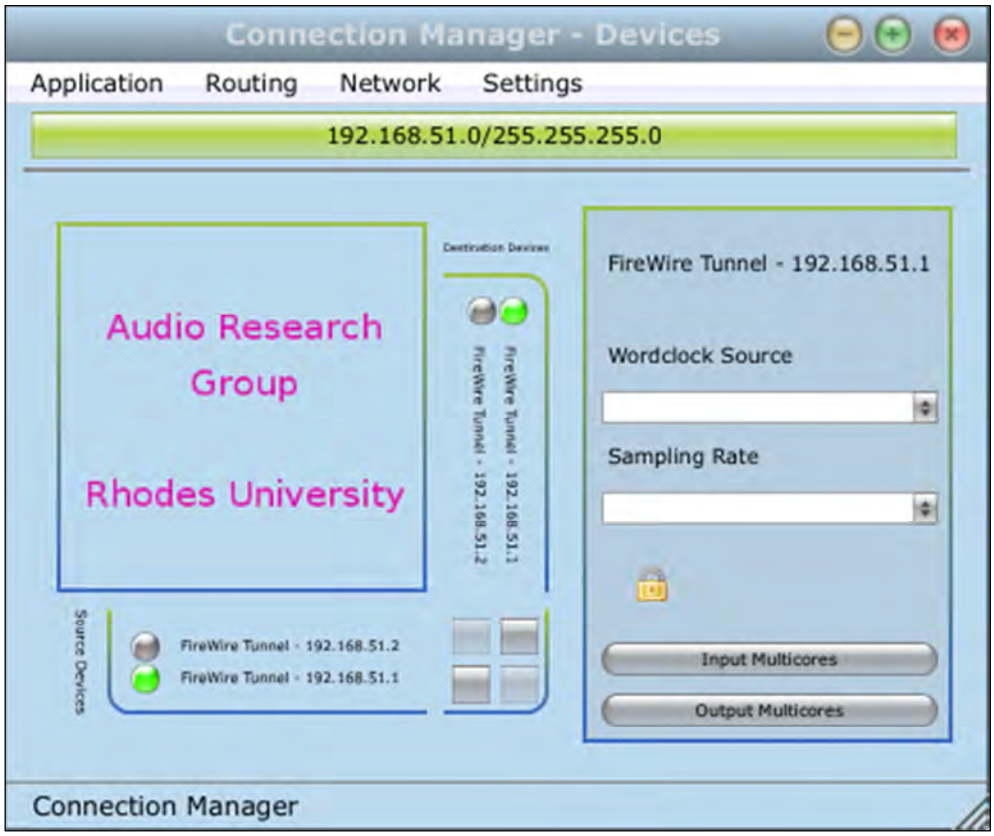

Figure 110: The Connection Manager's main interface

Once a device has been discovered on a network, it is possible to enumerate the device to discover its capabilities. In terms of the tunnel nodes, the Connection Manager is interested in knowing about its available multicores, and the properties of those multicores. For each *multicore parameter block* (of each *signal input* and *signal output section block*), the Connection Manager is able to determine the number of multicores that exist by obtaining the number of *parameter block index* level values that exist under the *multicore parameter block* level. In terms of the tunnel nodes, there exists only one input multicore, and one output multicore (a typical audio device may contain many multicores). Once the Connection Manager has obtained the number of input and output multicores, it is possible to query each of the parameters that exist for each multicore.

Selecting one of the cross points on the grid will display the various multicores associated with the devices, in the form of a grid (as shown in Figure 111). The multicore along the left hand side of the grid is viewed as a source multicore (this multicore is the originator of isochronous packets). The multicore along the top of the grid is viewed as a destination multicore (this multicore is the consumer of isochronous packets). Figure 111 is displaying the output multicore of the selected source tunnel node, and the input multicore of the selected destination tunnel node. Typically, a

device has a much larger number of multicores (allowing a single device to stream multiple streams of data). Hence, there is a desirability to display these multicores in the form of a grid.

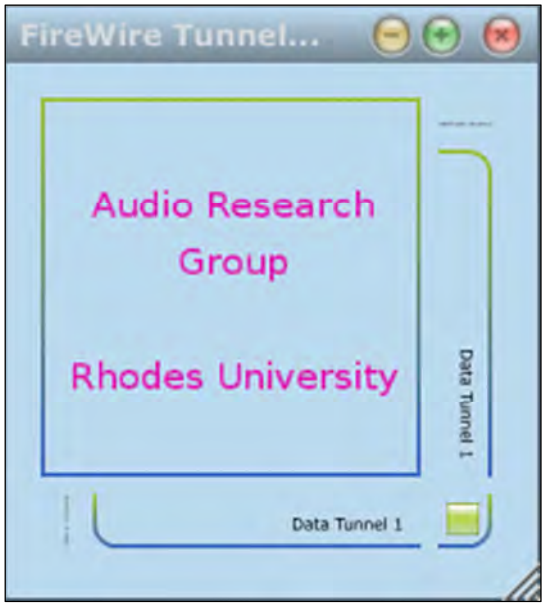

Figure 111: The Connection Manager's multicores display

With this grid display, it is possible to ensure that the source tunnel node's output multicore streams its isochronous packets to the destination tunnel node's input multicore. When the grid button (shown in Figure 111) is selected, the XFN stack of the Connection Manager sends XFN messages to the tunnel nodes to instruct them to set up their internal parameters to allow for streaming to take place. The following sequence of events takes place to ensure that this happens:

- The Connection Manager obtains the isochronous channel number of the source multicore from the source tunnel node. This is the channel number that the stream will be streamed on. It issues the tunnel node with an XFN *get value* request for the *isochronous channel number* parameter of the output multicore.
- The Connection Manager informs the destination tunnel node which channel to receive isochronous packets on. This is the channel number that was obtained from the transmitting tunnel node. The Connection Manager issues the destination tunnel node with an XFN *set value* request for the *isochronous channel number* parameter of the input multicore.
- The Connection Manager instructs the destination tunnel node to prepare itself to receive the stream. It issues the destination tunnel node with an XFN *set value* request for the *start* parameter (with a value of one) for the input multicore.

- The Connection Manager instructs the source tunnel node to start the stream transmission. It issues the source tunnel node with an XFN *set value* request for the *start* parameter (with a value of one) for the output multicore.

Selecting the multicores in Figure 111 displays the various parameters associated with them. Shown in Figure 112 is the window that is displayed when a source multicore is selected. These are the parameters that are depicted in Figure 107. The Connection Manager performs XFN *get value* requests on the *isochronous channel number*, *maximum payload size*, and *start* parameters to display their values. When any of these parameters are adjusted via the window, the new values are communicated to the relevant device via XFN *set value* request messages.

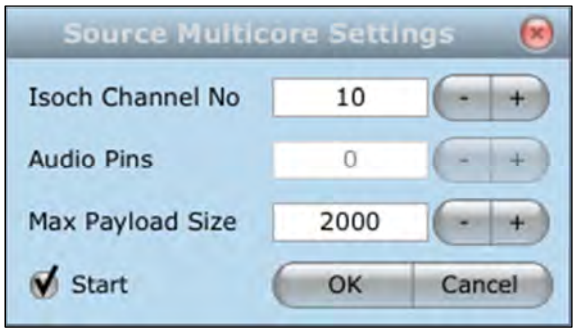

Figure 112: The Connection Manager's source multicore settings window

8.5 Conclusion

The development of the tunnel node application and the control thereof by the XFN protocol laid the groundwork for the interoperability of, and control over, IEEE 1394 and Ethernet AVB audio devices. The tunnel node application allows Ethernet frames to be transmitted in a deterministic fashion over IEEE 1394 networks. This was born from a need to allow a video encoder (which transmits Ethernet frames) to transit its frames to a video decoder (which receives Ethernet frames) via pre-existing IEEE 1394 networks.

XFN parameters were developed to allow for control over the tunnel nodes' parameters. The tunnel node application demonstrated the successful use of the XFN protocol for discovering remote tunnel nodes, establishing isochronous stream connections across IEEE 1394 networks, as well as allowing for other parameters of the devices to be viewed and adjusted.

Chapter 9 Networked Audio Devices

In this study, a number of pre-existing audio devices were used and further devices were developed. The pre-existing devices include prototype IEEE 1394 endpoint audio devices and IEEE 1394 bridges that were developed by *Universal Media Access Networks* (UMAN) [36]. Ethernet AVB endpoint devices, and IEEE 1394/Ethernet AVB audio gateway devices were developed. These devices were integral to the development of connection management capabilities. They provided IEEE 1394 audio reception and transmission capabilities, and Ethernet AVB audio reception and transmission capabilities. This chapter gives an overview of each of these devices, and the functionality they provide.

9.1 UMAN Evaluation Boards and Amplifier Nodes

UMAN evaluation boards and amplifier nodes are endpoint audio devices that contain a number of audio connectors which expect audio in a range of formats. The evaluation boards each contain an IEEE 1394 interface (expecting audio formatted according to IEC 61883-6 formatting rules), analogue inputs and outputs, ADAT interfaces, and AES-3 interfaces. The amplifier nodes each contain an IEEE 1394 network interface (expecting audio formatted according to IEC 61883-6 formatting rules), and analogue inputs and outputs. Figure 113 shows the front panel of a UMAN evaluation board which contains a number of analogue input and output connectors.

Figure 113: UMAN evaluation board (front)

Figure 114 shows the back panel of a UMAN evaluation board. The back panel of the device contains an Ethernet management interface, two IEEE 1394 interfaces, AES-3 interfaces, wordclock interfaces, ADAT interfaces, and MIDI [94] interfaces.

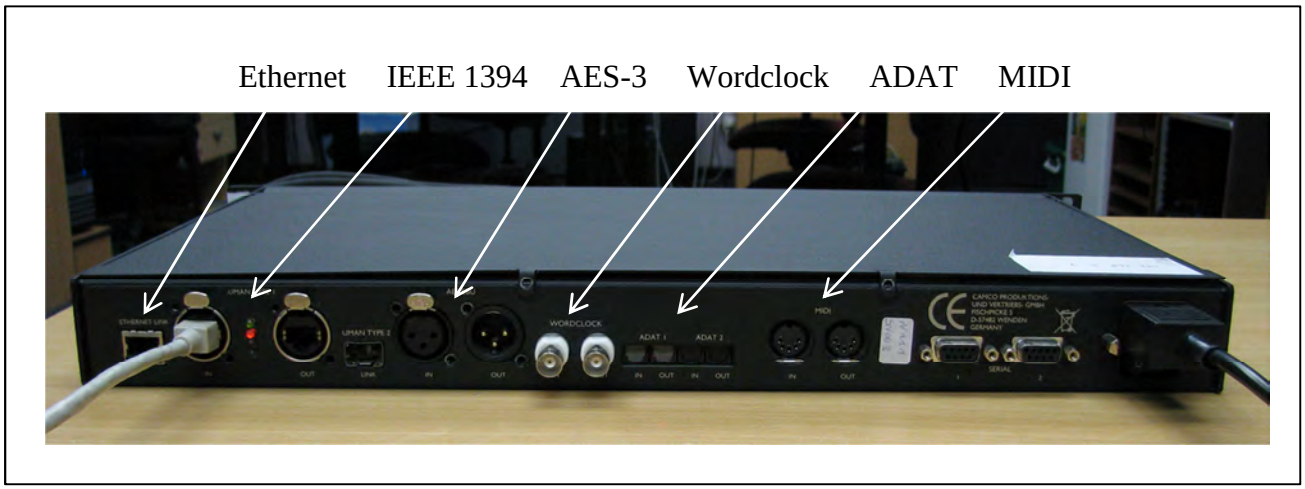

Figure 114: UMAN evaluation board (back)

It is possible to configure each of these devices such that audio is patched from the various inputs of the device to the various outputs of the device. For example, audio arriving at the IEEE 1394 interface of the device may be patched through to the ADAT and analogue output interfaces of the device. Figure 115 is a high-level diagram showing the functionality of a UMAN evaluation board with its various interfaces and patching capabilities. The devices provide mechanisms that allow their internal parameters (such as the patching of audio from one interface to another) to be adjusted. The amplifier nodes are similar to the evaluation boards, except that they provide less functionality in terms of the audio formats that they are able to process.

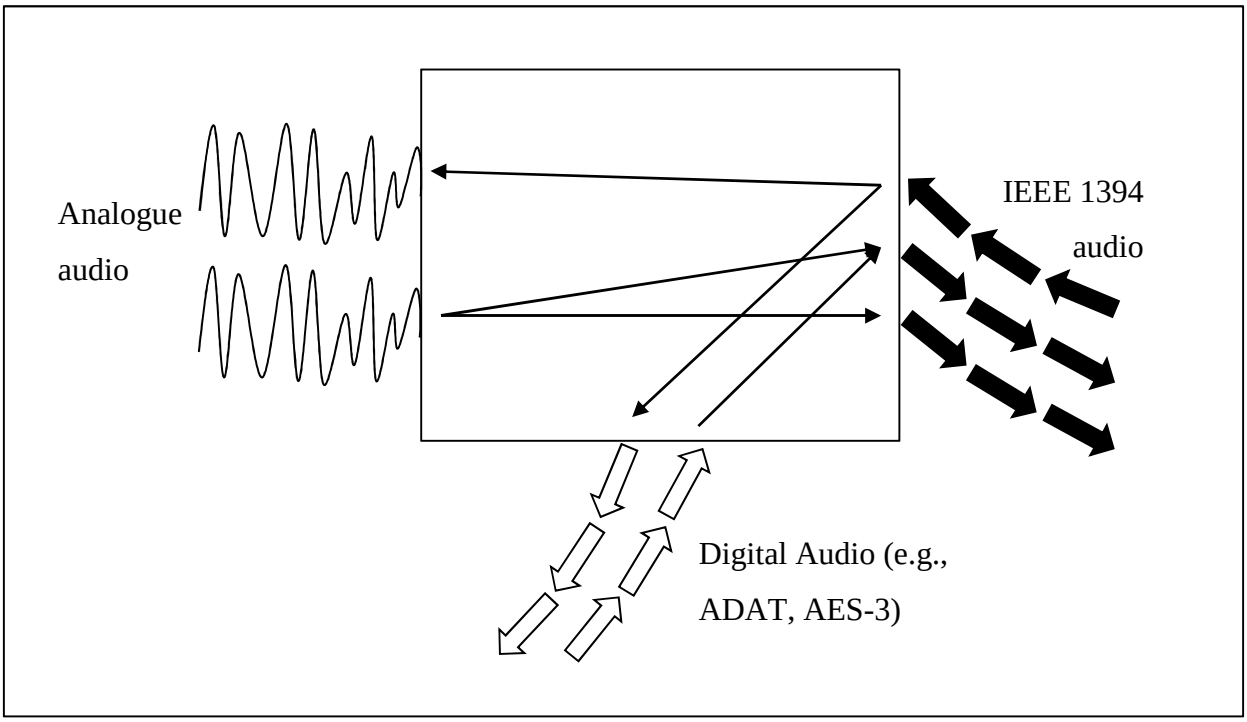

Figure 115: Conceptual routing within a UMAN evaluation board

Each of the UMAN endpoint devices contains an XFN stack. The internal address hierarchy of the stack is built up to reflect the natural hierarchy of the device. Via the XFN stack, it is possible to adjust the various parameters of the device. For example, the stack allows for the routing of audio signals arriving at the inputs of the device to the outputs of the device, and allows for connections to be established between the device and other IEEE 1394 devices.

9.2 UMAN IEEE 1394 Bridges/Routers

UMAN bridge/router devices are used to bridge IEEE 1394 buses together, and are used to route IP packets between IP subnets. Figure 116 shows a UMAN bridge/router connecting two IEEE 1394 buses together. This device also contains an Ethernet port to allow for management of the device.

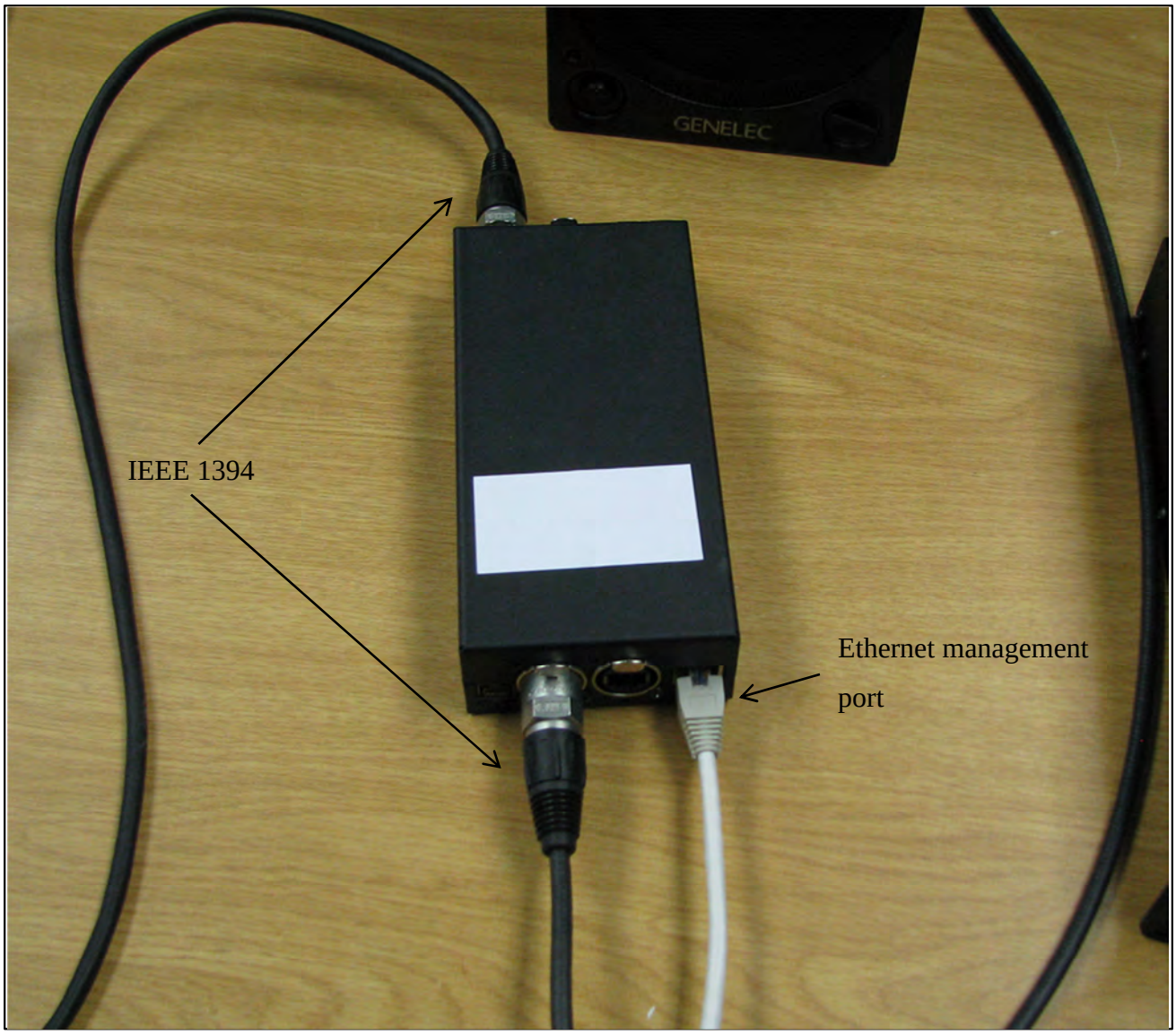

Figure 116: A UMAN IEEE 1394 bridge/router

The XFN protocol works such that an IEEE 1394 bus is mapped onto an IP subnet, as shown in Figure 117. When a stream connection is established across one of the UMAN IEEE 1394 bridges, two separate connections are established: one from the transmitting device to the bridge, and a second connection from the bridge to the receiving device. The XFN specification works such that connections are established between devices on an IP subnet. Thus, all of the devices that are able to make direct connections with each other exist on a unique IP subnet.

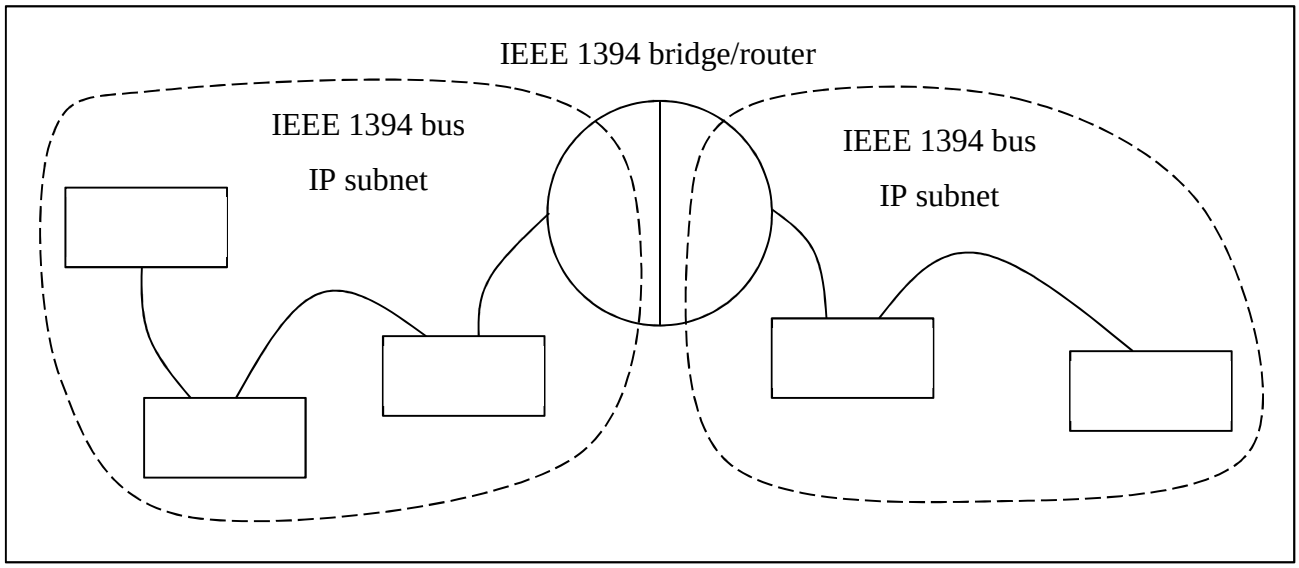

Figure 117: IEEE 1394 bus to IP subnet mapping

Figure 118 shows a UMAN IEEE 1394 bridge/router joining two IEEE 1394 buses together. Each bus contains IEEE 1394 evaluation devices.

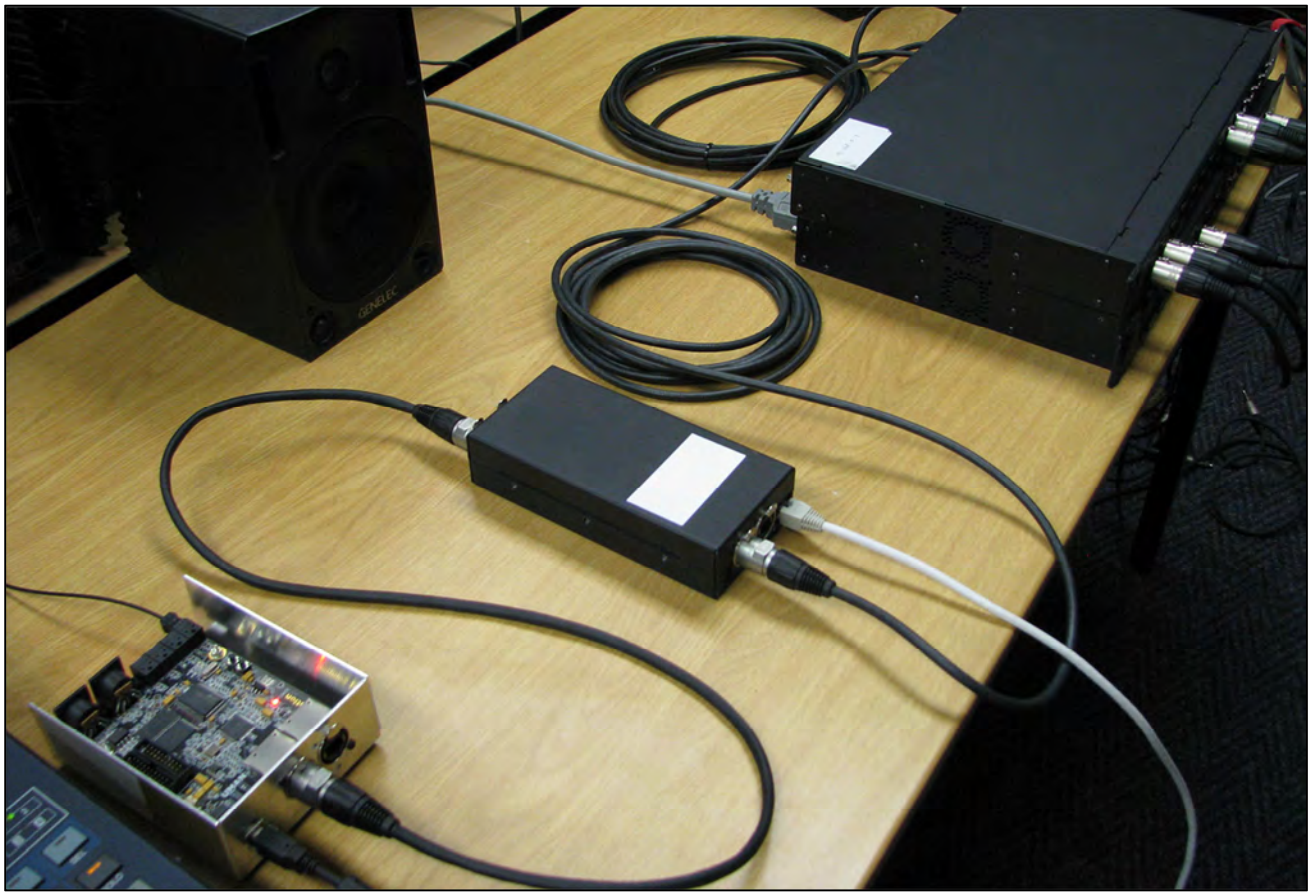

Figure 118: Two IEEE 1394 buses joined with an IEEE 1394 bridge

Each portal of the IEEE 1394 bridge is able to accept a number of incoming IEEE 1394 isochronous streams, and is able to transmit a number of IEEE 1394 isochronous streams. The bridge provides capabilities to bridge these streams from one portal to another. Figure 119 shows an example of a two portal IEEE 1394 bridge. Each side is both transmitting and receiving IEEE 1394 isochronous streams. Streams received on one portal are bridged across to the other portal for transmission.

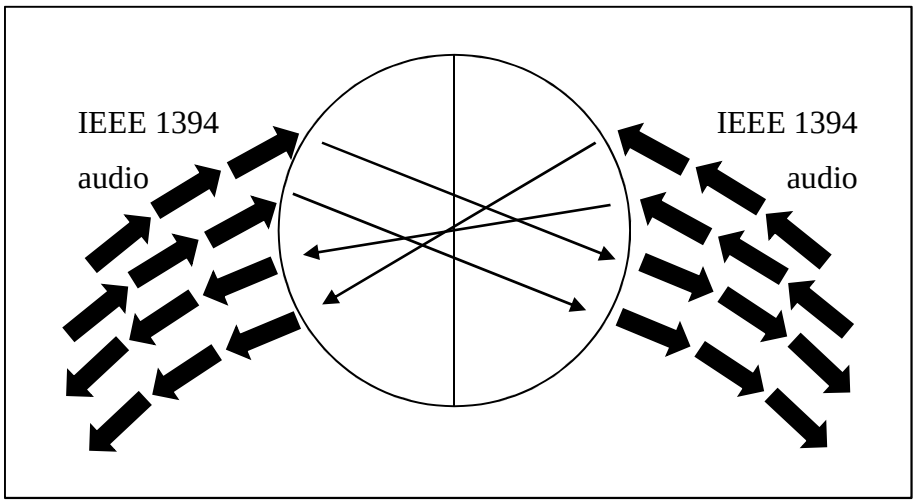

Figure 119: Conceptual stream routing within an IEEE 1394 bridge

9.3 Ethernet AVB Endpoint Devices

As part of this study, a Linux [95] based software system was developed to act as an Ethernet AVB endpoint device. This device is both a producer and a consumer of analogue audio and of AVTP Ethernet AVB audio streams (see Section 6.1.2 “Ethernet AVB Audio Frame Formats”).

The device is able to receive analogue audio via an analogue sound card. The device can be configured to capture this audio, format it appropriately and encapsulate it within AVTP packets. These packets can then be sourced on an Ethernet AVB network. The device is also able to receive AVTP packets from an Ethernet AVB network. The audio data contained within these packets is extracted, formatted appropriately and sourced out of the device’s analogue outputs.

The Ethernet AVB endpoint device contains an XFN stack. This stack builds up an internal tree structure that reflects the natural hierarchy of the device. Via this XFN stack, the parameters of the device may be adjusted, allowing for connection management to take place between this device and

other compatible devices, and internally within the device. The conceptual layout of the device is shown in Figure 120.

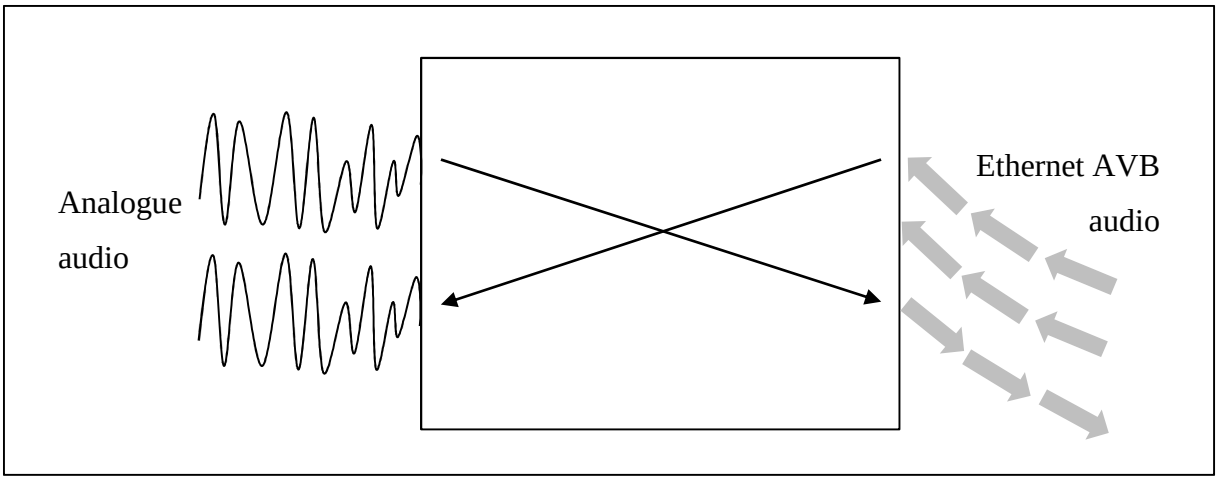

Figure 120: Conceptual routing within an Ethernet AVB endpoint device

9.4 IEEE 1394 / Ethernet AVB Audio Gateway Devices

A second Linux-based software system has been developed to act as an audio gateway between IEEE 1394 and Ethernet AVB networks. The audio gateway device is able to receive IEEE 1394 packets containing digital audio (formatted according to IEC 61883-6 formatting rules) via an IEEE 1394 network card. This captured audio is then appropriately formatted and encapsulated within AVTP frames which are sourced on an Ethernet AVB network. The audio gateway device is also able to receive AVTP packets (containing audio data) arriving at the Ethernet interface of the device. It is able to extract audio data from the packets, format it appropriately, encapsulate it within IEEE 1394 packets, and source this audio on an IEEE 1394 network.

The IEEE 1394/Ethernet AVB audio gateway device also contains an XFN stack. This stack builds up an internal tree structure that represents the natural hierarchical structure of this device. Via this XFN stack, the parameters of the device may be adjusted allowing for connection management to take place between this device and other devices, and for connection management to take place internally within the device. The conceptual layout of the device is shown in Figure 121.

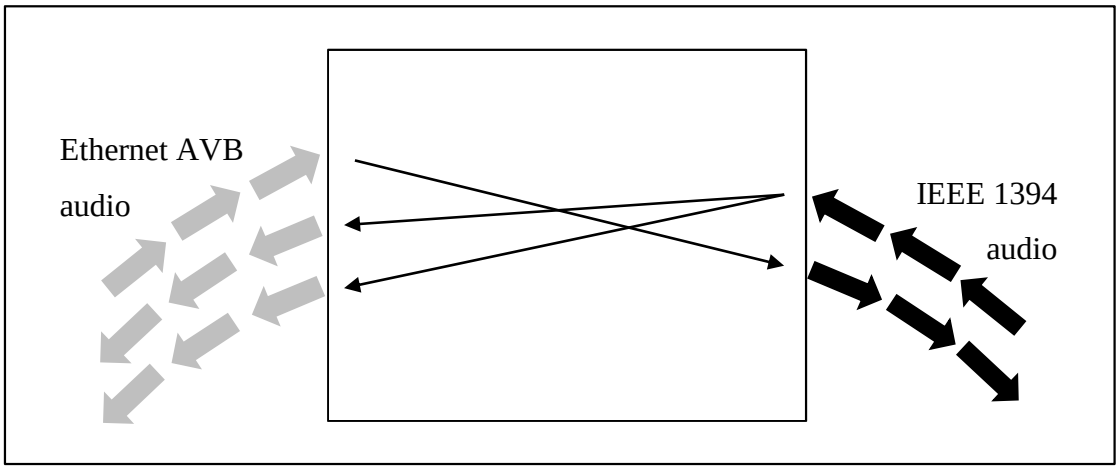

Figure 121: Conceptual routing within an IEEE 1394/Ethernet AVB audio gateway device

9.5 AVB Device Architecture

The Ethernet AVB devices have been developed as a set of interacting components. Figure 122 shows the architecture of the Ethernet AVB devices. The Ethernet AVB end-point devices do not contain the 1394 component, and the IEEE 1394/Ethernet AVB audio gateway devices do not contain the analogue component.

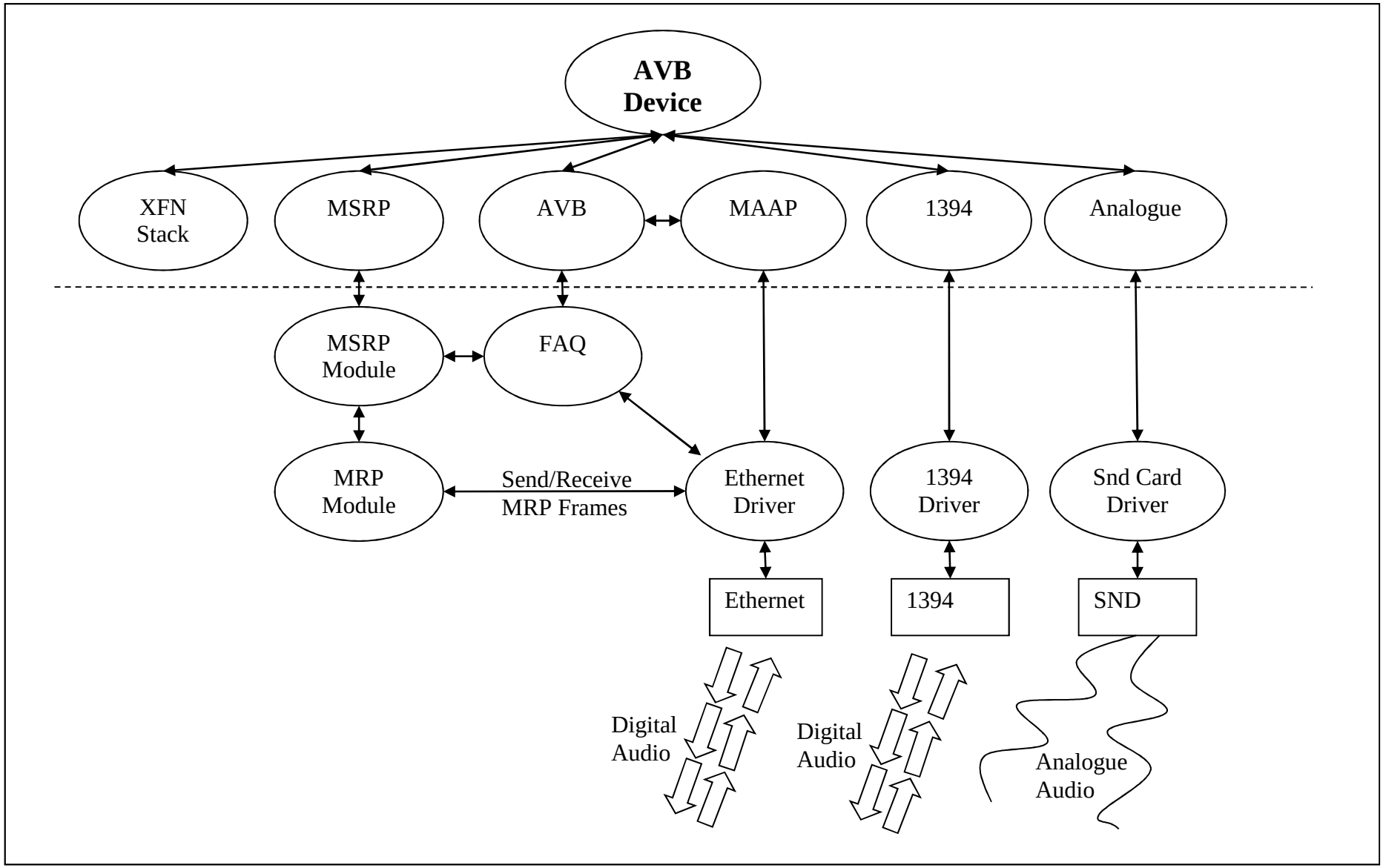

Figure 122: Ethernet AVB device architecture

This section provides a brief overview of these components. These components are discussed in more detail in subsequent chapters.

- AVB Device: The AVB Device component coordinates the activities of the system as a whole. It links the components together and allows for communication to take place between the components. It ensures orderly initialisation and shutdown of the components of the system.
- XFN Stack: The XFN Stack component is an implementation of the XFN protocol (see Section 7.4 “XFN”). It is responsible for all network communication with XFN stacks on other devices, for interpreting XFN messages from remote devices, for creating appropriate XFN messages for transmission to remote devices, and for notifying the AVB Device component of any requests that it should respond to.
- MRP Module, MSRP Module and MSRP: The MRP Module component is a kernel module implementation of MRP (see Section 3.2.1 “Multiple Registration Protocol”). The MSRP Module component is a kernel module implementation of MSRP (see Section 3.2.4 “Multiple Stream Reservation Protocol”). The MSRP component allows for user-space applications to interact with the MSRP kernel module.
- AVB: The AVB component is an implementation of the AVTP protocol (see Section 6.1.2 “Ethernet AVB”). It is responsible for accepting audio samples in various formats and for packetizing them in AVTP frames for transmission onto an AVB network. It is also responsible for accepting AVTP frames and extracting the audio samples so that they may be passed to other components.
- FAQ: The FAQ component is a placeholder for the implementation of the AVB *forwarding and queuing* rules (see Section 4.2.2 “Forwarding and Queuing”).
- MAAP: The MAAP component is an implementation of MAAP (see Section 6.3 “AVTP Address Allocation”).
- Ethernet Driver: The Ethernet Driver component is an Ethernet driver through which Ethernet frames are sent and received.
- 1394 and 1394 Driver: The 1394 component is responsible for the reception and transmission of audio over an IEEE 1394 bus. It is responsible for accepting audio samples in various formats and packetizing them for transmission, and it is responsible for receiving IEEE 1394 packets and extracting the audio samples so that they may be passed to other components.
- Analogue and Snd Card Driver: The analogue component is responsible for sending and receiving audio samples to and from an analogue sound card via a sound card driver (Snd Card

Driver). It is also responsible for accepting audio samples in various formats for transmission, and for receiving audio samples from a sound card and for passing these to other components of the system.

9.6 Conclusion

A number of audio devices were used or developed as part of this study. This chapter provided a brief overview of the functionality of UMAN evaluation devices. It also provided an overview of the functionality and architecture of the AVB capable devices that were developed as part of this study. The following chapters will discuss these devices in detail, showing their architecture and the mechanisms used to achieve audio streaming and control.

Chapter 10 Ethernet AVB Devices

As part of this study, proof of concept software was developed for computers running the Linux operating system, enabling them to act as Ethernet AVB endpoint devices, and as IEEE 1394/Ethernet AVB audio gateway devices. Chapter 9 gave a brief introduction to the functionality of these devices and to their components. This chapter provides a detailed discussion of these devices, their components, and the operation of these components in order to achieve the first goal of this work: provide the ability to allow audio data to be transferred from IEEE 1394 to Ethernet AVB networks, and vice versa.

10.1 Audio Components

The IEEE 1394 and Ethernet interfaces of the Ethernet AVB endpoint devices and the IEEE 1394/Ethernet AVB audio gateway devices can be viewed as audio interfaces, since they are able to transmit and receive audio streams. As shown in Figure 123, each of these interfaces has the same conceptual functionality: the ability to receive and transmit audio streams.

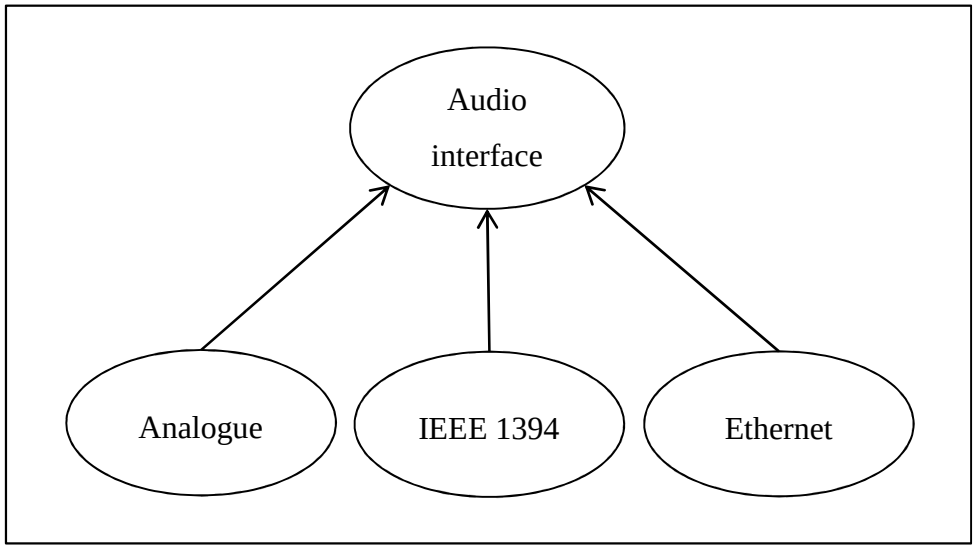

Figure 123: Audio interfaces

With this in mind, the Ethernet AVB endpoint devices and the IEEE 1394/Ethernet AVB audio gateway devices transfer audio data from one audio interface to another. An endpoint device transfers audio data from the analogue interface of the device to its Ethernet interface, and vice versa.

An audio gateway device transfers audio data from the IEEE 1394 interface of the device to its Ethernet interface, and vice versa.

Each one of these audio interfaces comprises one or more audio inputs and one or more audio outputs. An audio input is used to receive an audio stream from a source that is external to the system, and an audio output is used to send an audio stream to a sink that is external to the system. Within a device, an audio input of an audio interface is able to send its received audio stream to one or more audio outputs of an audio interface. These associations, or connections, are configurable.

Each audio interface of an endpoint or an audio gateway device is modelled in software by a component. Each input of an audio component keeps track of the output(s) to which it should send its received audio signal to. Figure 124 shows an IEEE 1394/Ethernet AVB audio gateway device with an AVB component representing the Ethernet interface of the device, and a 1394 component representing the IEEE 1394 interface of the device. Each of these components has a number of inputs and outputs to receive and transmit audio streams. The figure shows how the inputs of one component may be associated with the output(s) of another component.

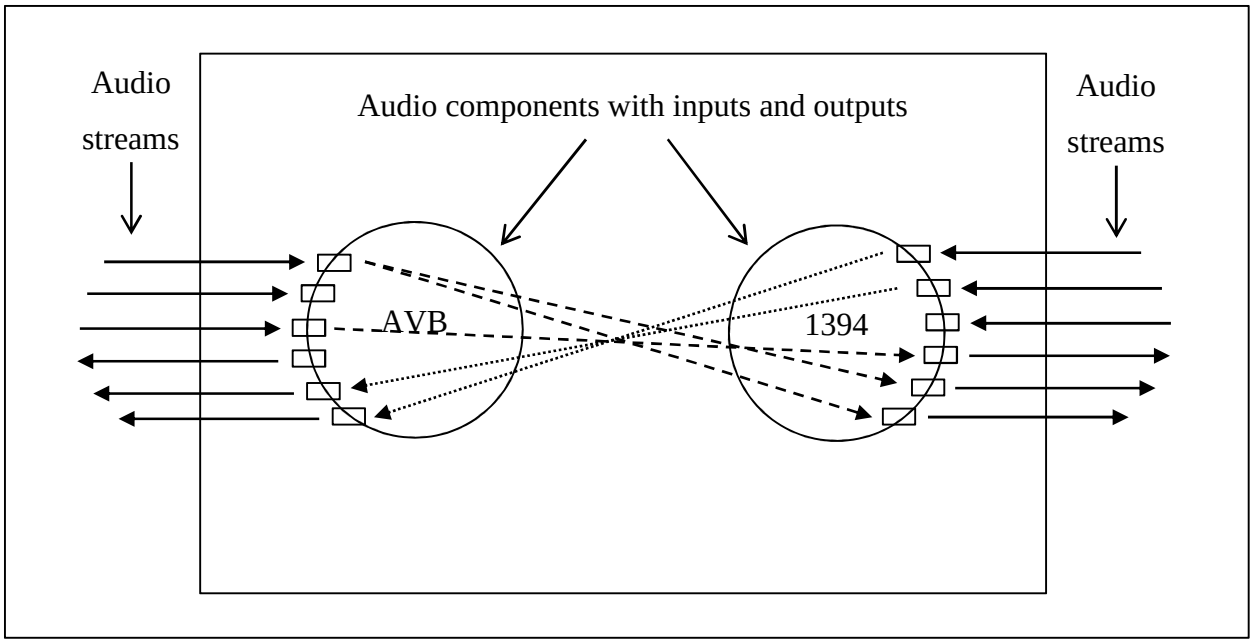

Figure 124: An example audio gateway device with audio components

Audio data that arrives at an input of a component is immediately sent to each of the outputs that it is associated with, if any. Each one of the audio components accepts audio samples packaged in a specific format. The audio data is packaged into a format that is appropriate for each specific output

and is placed into a queue for the output. Each output drains its queue and transmits the audio data out of the interface that it is representing.

10.1.1 Analogue Component

An Ethernet AVB endpoint device contains a software analogue component that is responsible for reception and transmission of analogue audio. This component interacts with the system's sound card driver via the *Advanced Linux Sound Architecture* (ALSA) system [96]. ALSA provides the Linux operating system with audio functionality. It supports multiple audio interfaces and provides a user-space library to allow for audio program development.

In the computer systems used for this study, an analogue component of an Ethernet AVB endpoint device has one analogue input (which represents the system's sound card *line in* interface) and one analogue output (which represents the system's sound card *speaker out* interface). These interfaces are stereo interfaces. Thus, the analogue input and the analogue output can each be viewed as a multicore that contains two channels of audio. A multi-channel sound card would have an analogue component with more than two inputs and outputs.

An input of the analogue component reads audio samples from ALSA (which originate from the analogue input of the sound card). The left and right samples are interleaved together in a *frame*. In the computers used during this study, the frames are presented to the application in groups of 32 continuous frames at a time. Figure 125 shows an example of the structure of the frames that are presented to the Ethernet AVB endpoint application by ALSA. For this system, each sample is formatted as a signed 16-bit little-endian number. Once these samples have been received by the Ethernet AVB endpoint device, they are transferred to the output(s) associated with the input, transformed into an appropriate format for the particular output, and then queued for transmission.

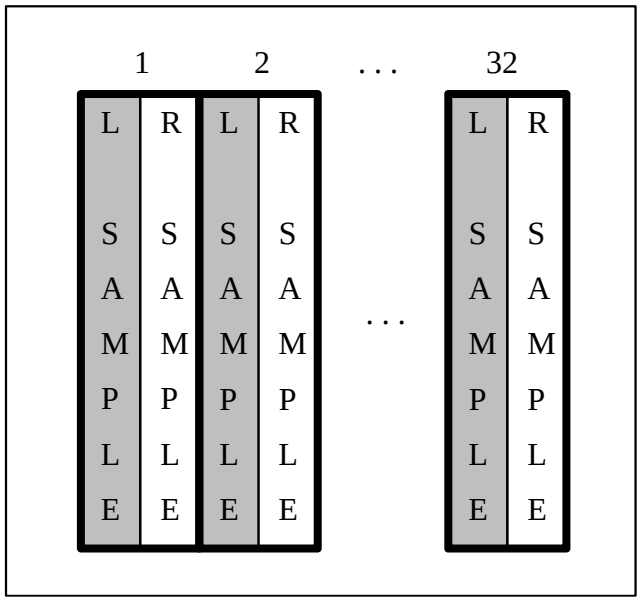

Figure 125: Example ALSA frames

An output of the analogue component is used to transmit audio samples to ALSA (and hence to the analogue output of the sound card). ALSA expects the samples to be interleaved and to be presented in a buffer of frames (each containing two samples for a left and a right channel), as shown in Figure 125. Each sample has to be formatted as a signed 16-bit little-endian number.

10.1.2 AVB Component

The Ethernet AVB endpoint devices and the IEEE 1394/Ethernet AVB audio gateway devices each contain a software AVB component. This component is responsible for the reception and transmission of AVTP audio streams across an Ethernet network (Section 6.1.2 “Ethernet AVB Audio Frame Formats” details AVTP). The AVB component is configured to transmit SR class A streams at a rate of 8000 packets per second. The number of sequences per stream is configurable.

The inputs and outputs of the AVB component are configured in software. Each input is used to receive a stream of AVTP frames (identified with a stream ID) from an Ethernet AVB network, and each output is used to transmit a stream of AVTP frames (identified with a stream ID) onto an Ethernet AVB network. Each input has a stream ID associated with it. This stream ID is configurable and is used to identify the stream frames that it is to receive from the Ethernet AVB network. Once an input has received stream frames, the audio samples contained within those frames are transferred

to the output(s) associated with the input, transformed to an appropriate format for the particular output, and queued awaiting transmission.

An output of the AVB component is used to transmit an AVTP audio stream onto an Ethernet AVB network. Associated with each one of these outputs is a stream ID that is generated when the system is initialised. The first 6 octets of the stream ID are made up of the MAC address that is associated with the Ethernet interface of the system. The last two octets are generated consecutively for each AVB output, starting with the value one. Each stream ID (that is associated with an AVB output) does not change during the lifetime of the program.

Also associated with each output of the AVB component is a unique multicast MAC address. This is the MAC address to which any stream frames transmitted by the output will be addressed. AVTP has a reserved range of multicast MAC address which it may use for stream transmission. Table 26 on page 160 shows this range of addresses. When the system is initialised, MAAP is run such that a range of multicast MAC addresses is allocated for use by the system. Once the protocol has obtained a unique range of MAC addresses, each output of the AVB component is assigned a unique multicast MAC address. Section 6.3 “AVTP Address Allocation” provides an overview of MAAP, and Section 10.4 “MAAP Component” shows its implementation in the AVB devices.

10.1.3 1394 Component

Each IEEE 1394/Ethernet AVB audio gateway device contains a software 1394 component. This component is responsible for the transmission and reception of audio streams to and from an IEEE 1394 bus. The formatting and transmission of the audio streams is detailed in Section 6.1.1 “IEEE 1394 Audio Packet Formats”. The inputs and outputs of the 1394 component are configured in software. An input is used to receive a stream of isochronous packets containing AM824 audio data, and an output is used to transmit a stream of isochronous packets containing AM824 audio data.

Each 1394 input has an isochronous channel number associated with it. This number represents the isochronous channel that it will receive isochronous packets on. The isochronous channel number is configurable. Audio samples that are received at the input (via the isochronous packets) are transferred to the outputs that the input may be associated with. At the output, the audio samples are converted to a format that is suitable for the particular output, and queued for transmission.

Each 1394 output has an isochronous channel number associated with it. When the system is initialised, unique isochronous channel numbers are allocated to each 1394 output. The availability of isochronous channel numbers is verified via the *channels available* register of the *isochronous resource manager* (IRM) node on the IEEE 1394 bus.

The transmission of audio data in CIP packets onto IEEE 1394 buses may take place in one of two modes: blocking, or non-blocking. When blocking mode is used, a transmitter waits until it receives a number of events equal to the *SYT interval* (which is eight when sampling at 48 KHz) before it transmits a packet. In non-blocking mode, a transmitter will transmit the number of events that it received since the last isochronous cycle. For transmission onto an IEEE 1394 network, the IEEE 1394 audio devices used in this study made use of the blocking transmission mode.

10.2 Audio Formatting

The packaging of audio data received on an audio interface of either an Ethernet AVB endpoint device or an IEEE 1394/Ethernet AVB audio gateway device is not compatible with the other interface of the same device. When transferring audio from one interface to another, the audio data has to be repackaged into an appropriate format.

10.2.1 Ethernet AVB Endpoint Device

When audio data is transferred from the analogue input of an Ethernet AVB endpoint device to an Ethernet AVB output, the 16-bit audio samples received via the analogue input are packaged into AVTP frames using the raw AM824 format. In this study, the analogue input of the sound card was set to sample at 48 kHz. As the analogue input samples audio at 48 kHz, and the AVB output transmits 8000 packets per second, six *events* are packaged into each AVTP packet, as shown in Figure 126. Each *event* represents the simultaneous sampling of the right and left channels of the analogue input of the device. The samples of the right and left channels each form a sequence (channel) of audio.

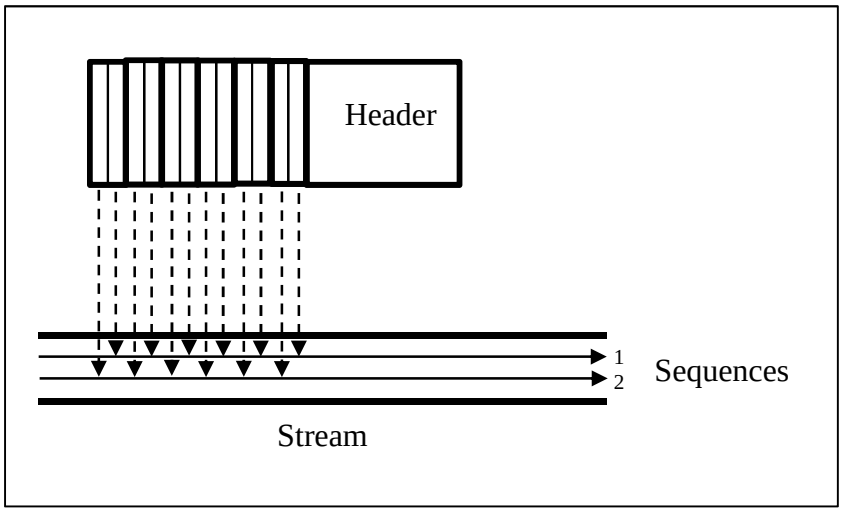

Figure 126: Packaging of audio in an AVTP frame

Figure 127 shows the packaging of six frames of audio received via ALSA into an AVTP frame. Each label of each AM824 sample is given the value 0x42 defined to represent a raw audio 16-bit sample (see Section 6.1.1.3 “IEC 61883-6”). The actual audio sample is placed into the 24-bit data section with the last octet set to zero. The figure shows the complete packaging of the audio data in an AVTP stream common frame with a CIP header. The fields of the AVTP and CIP headers are filled as specified in Section 6.1.2 “Ethernet AVB Audio Frame Formats”. The *stream ID* field is set to the value of the stream ID associated with the AVB component’s output. The *tv* (timestamp valid) field is set to zero indicating that the timestamp is not valid. Timestamps were not included in the AVTP frames due to the unavailability of a compatible combination of an IEEE 802.1 AS capable network interface, a suitable driver for the network interface, and gPTP software. Also, the unpredictable processing nature of user-space programs running on a general purpose operating system on a general purpose computer result in a timing uncertainty that is higher than that specified by IEEE 1722 (which is 125µs for class A streams (see Section 6.2.2.2 “Presentation Time Measurement Points”)).

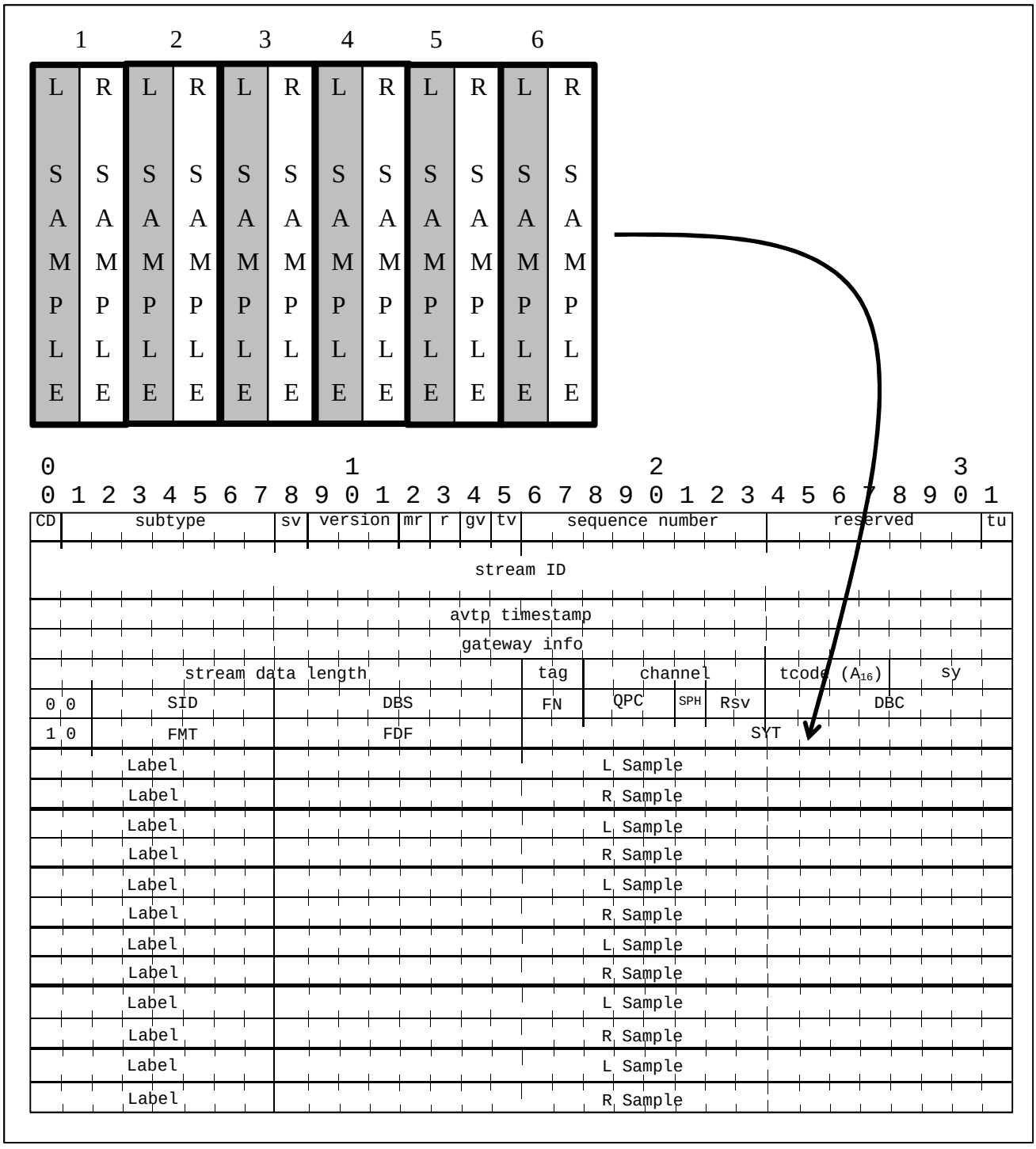

Figure 127: AVTP common stream header with CIP header and payload

When audio is transferred from an input of the Ethernet interface to an output of the analogue interface, the audio samples packaged into the received AVTP frames need to be extracted and packaged appropriately for ALSA to process. As the analogue output of the Ethernet AVB endpoint device only accepts two channels of audio, only the first two sequences of audio are extracted from

incoming AVTP frames (if more than two samples exist per event). These samples are then passed to ALSA, formatted as shown in Figure 125 on page 234.

10.2.2 IEEE 1394/Ethernet AVB Audio Gateway Device

When audio samples received via the IEEE 1394 interface of an IEEE 1394/Ethernet AVB audio gateway device are transferred to its Ethernet interface, the audio data is extracted from the received CIP packets and placed into AVTP frames. Each AVTP stream can be configured with the number of sequences it should contain. If an AVTP stream is configured to contain *n* sequences, where *n* is less than the number of sequences received on the IEEE 1394 interface, then the first *n* sequences are extracted from the received CIP packets, and the rest are discarded, as shown in Figure 128. If *n* is more than the number of sequences in the received CIP packets, then all of the sequences are extracted from the isochronous packets and packaged into AVTP frames. The rest of the sequences in the AVTP frames are zeroed.

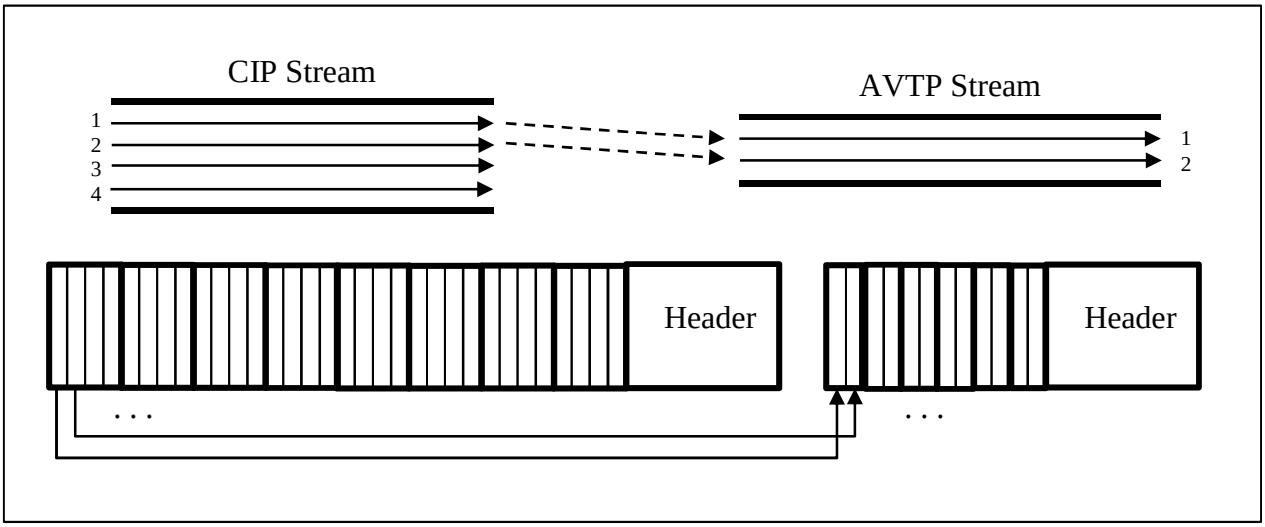

Figure 128: Stream sequence mapping

Assume that the packet shown in Figure 129 is received on the IEEE 1394 interface of an IEEE 1394/Ethernet AVB audio gateway device. Also assume that this is the first packet of a stream to arrive at the audio gateway. The packet contains eight events, as the transmitting IEEE 1394 device is configured to transmit audio in blocking mode. Each event in the packet contains four samples (hence, the packet represents four sequences of audio in a stream).

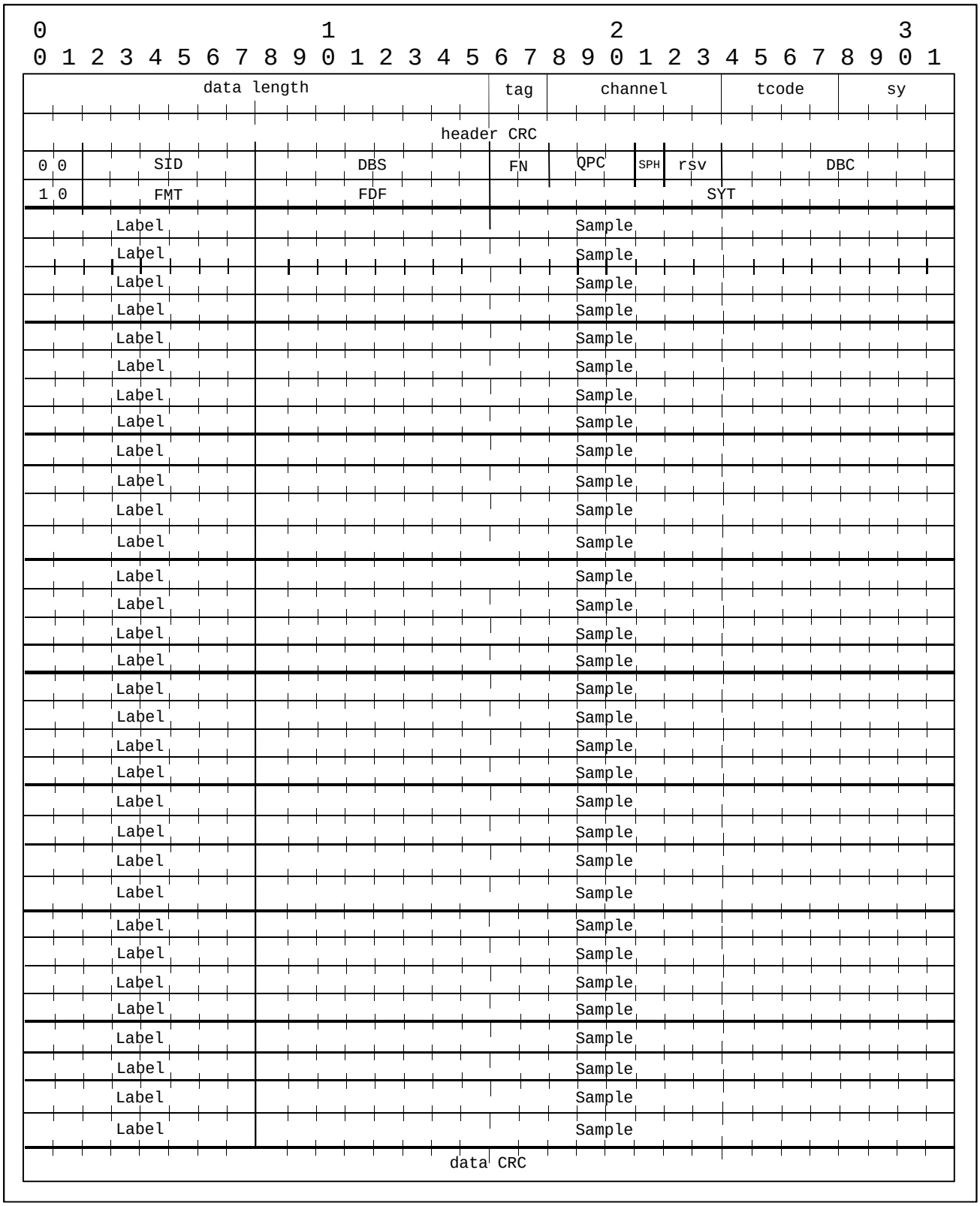

Figure 129: A received CIP packet

Assume that the IEEE 1394 input that received the packet shown in Figure 129 is associated with an output of the AVB component, and this output is configured to transmit two sequences of audio. In

this instance, the audio transmitted by the AVB component will be formatted as shown in Figure 130. The first two sequences of audio of the first six events will be packaged into an AVTP frame, and this frame will be queued for transmission. The first two sequences of the last two events will be packaged into the following AVTP frame. This frame will wait for the next IEEE 1394 CIP packet to arrive, where the first two sequences will be extracted from the first four events and placed into the remainder of the second AVTP frame. The second AVTP frame will then be queued for transmission.

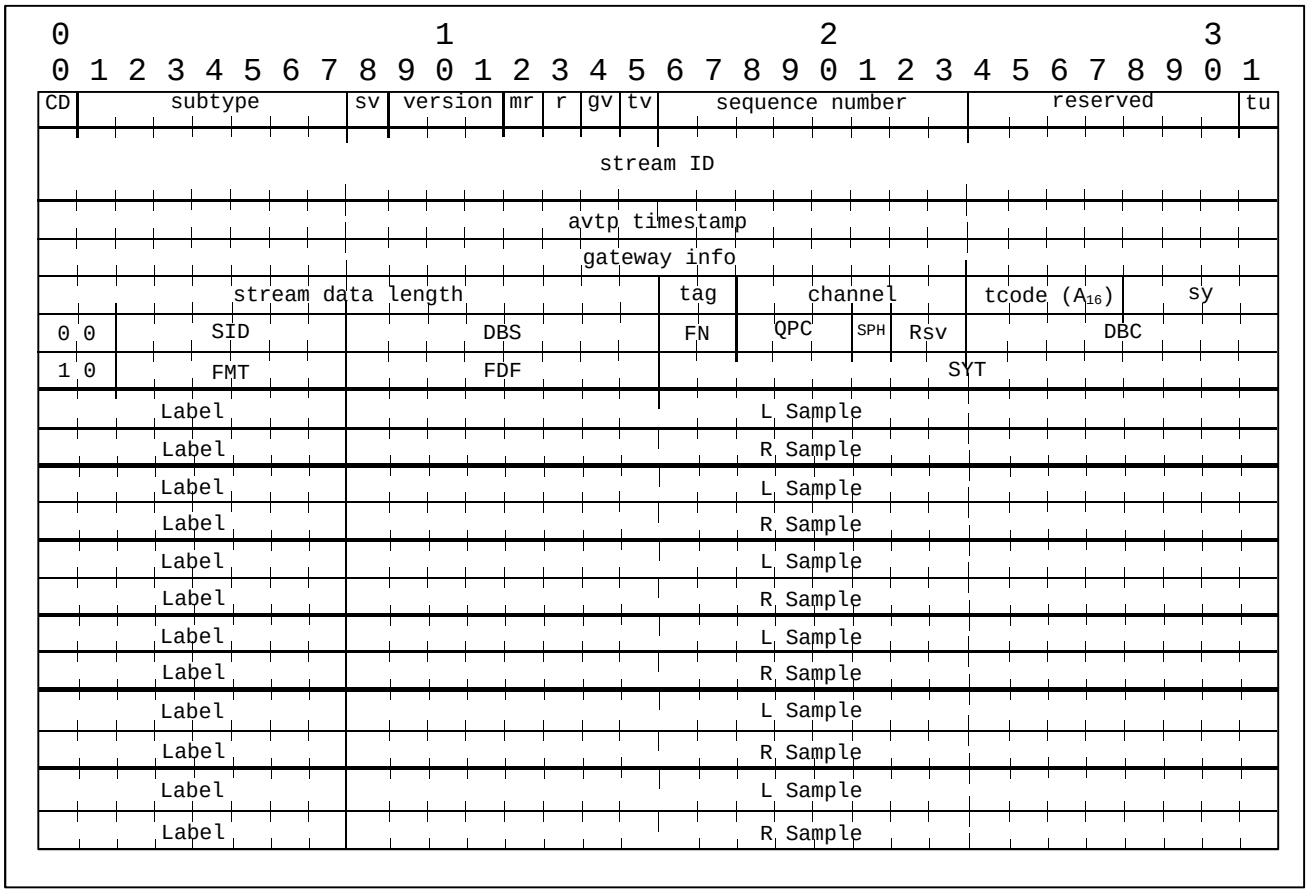

Figure 130: A transmitted AVTP frame

The same is true in the opposite case. Audio samples received in AVTP frames via the Ethernet interface of an audio gateway device are extracted and placed into CIP packets for transmission onto the IEEE 1394 bus. The number of sequences transmitted in each IEEE 1394 stream is configurable. Therefore, if an AVTP stream contains *n* sequences, and *n* is greater than the number of sequences supported by the CIP stream, say *m*, then the first *m* sequences are extracted from the AVTP frames and packaged into CIP packets. If *n* is less than *m*, all of the sequences are extracted from the AVTP frames and placed into CIP packets, and the remaining sequences are zeroed. As the IEEE 1394

outputs transmit audio in blocking mode, a packet is only queued for transmission once eight events have been received from the AVB input and packaged into the IEEE 1394 CIP packet.

10.3 Timing and Synchronisation

An IEEE 1394 bridge is responsible for bridging two IEEE 1394 buses together. Each portal of an IEEE 1394 bridge has an independent 24.576 MHz clock. These clocks typically do not run at exactly the same rate and drift apart. Thus IEEE 1394 bridges regenerate timestamp values within CIP packets such that these timestamps are relative to the cycle time of the portal that they are being transmitted on. See Section 6.2.1.1 “Cross IEEE 1394 Bridge Timestamp Regeneration”. Similar to an IEEE 1394 bridge, an IEEE 1394/Ethernet AVB audio gateway device has two separate incompatible clocks for each network type. Thus, it needs to implement similar cross network timing and synchronisation mechanisms as IEEE 1394 bridges do.

10.3.1 Timestamp Regeneration

When audio data is transferred across an IEEE 1394/Ethernet AVB audio gateway, it is essential that the intended absolute presentation time of the transmitter be maintained. Timestamp values need to be regenerated such that they are relevant on the network they are being transmitted on.

CIP packets received on the IEEE 1394 interface of an audio gateway device contain a 16-bit *SYT* field that contains the time (relative to the cycle time of the bus from which it was received) that the time-stamped event in the packet is to be presented at the receiver. The format of this field is the same as the lower 16-bits of the *cycle time* register, as shown in Figure 131. The *SYT* field is able to represent 2^4 (16) 125 µs isochronous cycles, which is equal to two milliseconds (125 µs × 16). The cycle offset field is able to represent up to 3071 ticks of the 24.576 MHz clock, which is equal to 124.96 µs (rounded to two decimal places), after which it rolls over to represent an isochronous cycle of 125µs. Therefore, the *SYT* field is able to hold a timestamp value that is up to 2.12496 ms (rounded to five decimal places) in the future.

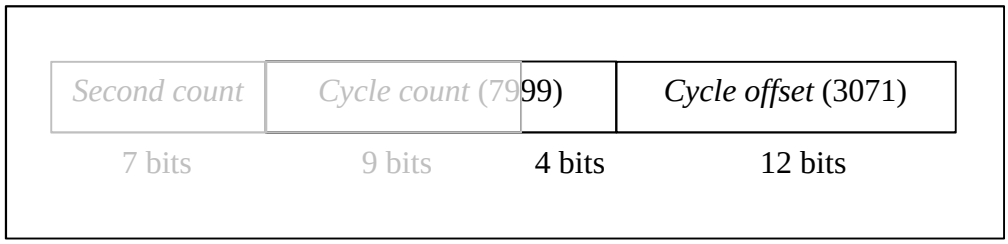

Figure 131: *SYT* field format

AVTP frames received on the Ethernet interface of an AVB device contain a 32-bit *avtp timestamp* field that represents the gPTP time (in nanoseconds) that the time-stamped event is to be presented at the receiver. The *avtp timestamp* field is able to represent 2^{32} ns which is equal to more than four seconds. Thus, the *avtp timestamp* field may hold a timestamp value that is more than four seconds into the future.

10.3.1.1 Timestamp Regeneration Example

Figure 132 shows a hypothetical gateway device that is used to join two networks together. Each side of the device contains an independent clock. The clock on side A (*clock A*) of the gateway has a nominal frequency of 12.5 MHz, and the clock on side B (*clock B*) of the gateway has a nominal frequency of 25 MHz. A packet arrives at side A of the gateway containing a presentation timestamp of 16 ticks relative to clock A. This timestamp needs to be transferred across the gateway such that when it arrives at its destination (on the network attached to side B of the gateway), it should be presented when clock A reaches a tick count of 16.

The difference between the two clocks, *delta*, needs to be calculated by subtracting clock A's count (expressed in block B's time base) from clock B's count:

$$delta = clock\ count_{B} - clock\ count_{\text{A expressed in B's clock time base}}$$

When the gateway calculates the difference, clock A's count is 12 ticks relative to clock A, and clock B's count is 36 ticks relative to clock B. In order to convert one clock's tick count to a tick count that is expressed relative to the other clock, the frequency difference between the two clocks needs to be accurately measured (for this example, assume that both clocks have exactly their nominal frequency). As clock B runs twice as fast as clock A, clock A's count would have a value of 24 ticks if it were running at the same rate as clock B. Thus the difference between these two clocks, in clock B's time base, is 12 ticks (36 ticks (B) – 24 ticks (A)).

As the timestamp is going to be transferred from network A ($timestamp_A$) to network B ($timestamp_B$), it has to be expressed in clock B's time base. The $timestamp_A$ has been generated relative to clock A's clock and thus needs to be expressed in clock B's time base. As clock B is running twice as fast as clock A, the $timestamp_A$'s value would be 32 ticks if it had been generated based on clock B's frequency. Once the timestamp has been converted, the difference between the two clocks, *delta*, is added to the converted timestamp to obtain the timestamp value that is to be transmitted:

$$timestamp_B = timestamp_{A\ \text{expressed in B's clock time base}} + delta_{\text{expressed in B's clock time base}}$$

Thus, in this example, the transmitted timestamp will have a value of 44 ticks (32 ticks + 12 ticks) which will maintain the original transmitter's intended absolute presentation time; when clock A reaches 16 ticks, clock B will reach 44 ticks.

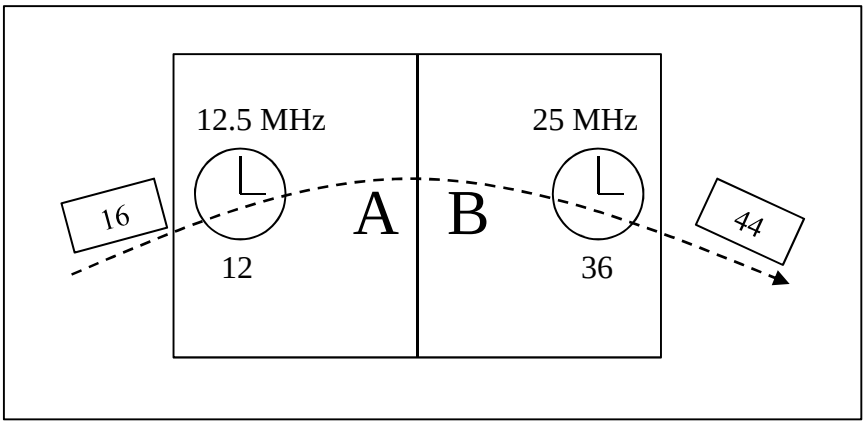

Figure 132: Hypothetical timestamp regeneration

10.3.1.2 CIP and AVTP Timestamp Regeneration

Timestamp values contained in CIP packets received via the IEEE 1394 interface of an audio gateway device need to be transformed into timestamps to be transmitted in AVTP frames via the Ethernet interface. Also, timestamp values contained in AVTP frames received via the Ethernet interface of an audio gateway need to be transformed into timestamps to be transmitted in CIP packets via the IEEE 1394 interface. Thus, received *SYT* timestamps need to be transformed from a format that contains a *cycle count* and a *cycle offset* field (that is expressed relative to the cycle time on the IEEE 1394 bus) to a nanosecond value (that is expressed relative to the gPTP time on the Ethernet AVB network). A received *avtp timestamp* field needs to be transformed from a nanosecond

value (that is expressed relative to the gPTP time on the Ethernet AVB network) to a timestamp value that is expressed in a *cycle count* and *cycle offset* field (expressed relative to the IEEE 1394 bus's cycle time).

The transformation of timestamp values should happen in a similar fashion to the method used by IEEE 1394 bridges, as described in Section 6.2.1.1 "Cross IEEE 1394 Bridge Timestamp Regeneration". An IEEE 1394/Ethernet AVB audio gateway device should be able to calculate the time difference between the clocks of each network. To do this, it must simultaneously read the current clock times of each network. Once these values have been retrieved, the times should be expressed in the same unit and the time difference between the two should be calculated.

The gPTP clock time is stored as seconds and nanoseconds. It is possible to convert the current IEEE 1394 clock time to gPTP time with the following formulae:

- *gPTP seconds* = *bus time*
- *gPTP nanoseconds* = int ((*cycle count* × 125 µs × 10^3) + (*cycle offset* × (125 µs × 10^3 / 3072)))

It is possible to convert the current gPTP clock time to *bus time*, *cycle count*, and *cycle offset* with the following formulae (where % represent a modulo operation):

- *bus time* = *gPTP seconds* % 2^{32}
- *cycle count* = int (*gPTP nanoseconds* / 125 µs × 10^3)
- *cycle offset* = int ((*gPTP nanoseconds* / (125 µs × 10^3 / 3072)) % 3072)

Thus, the time difference between the two clocks can be calculated in nanoseconds and IEEE 1394 clock time by ensuring that both clock times are expressed in the same units.

It is possible to convert a timestamp conveyed in a CIP packet to a value that is expressed in nanoseconds with the following formula:

- int ((*cycle count* × 125 µs × 10^3) + (*cycle offset* × (125 µs × 10^3 / 3072)))

It is possible to convert a timestamp expressed in nanoseconds conveyed in an AVTP frame to a value that is expressed in IEEE 1394 *cycle counts* and *cycle offsets* with the following formulae:

- *cycle count* = int ((*avtp timestamp* / 125 µs × 10^3) % 8000)
- *cycle offset* = int ((*avtp timestamp* / (125 µs × 10^3 / 3072)) % 3072)

With the above information, a received timestamp in a CIP packet can be transmitted in an AVTP frame by converting the *SYT* value to a nanosecond value, and adding the nanosecond difference between the two clocks. A received timestamp in an AVTP packet can be transmitted in a CIP packet by converting the *avtp timestamp* value to an *SYT* value and by adding the time difference between the two clocks (expressed in IEEE 1394 clock time) to it.

10.3.1.2.1 CIP and AVTP Timestamp Regeneration Example

Assume that an IEEE 1394/Ethernet AVB gateway device at a particular time has the times as shown in Table 40.

IEEE 1394		**gPTP**	
Bus time	12 s	*gPTP seconds*	34 s
Cycle count	100 cycles	*gPTP nanoseconds*	15,187,500 ns
Cycle offset	768 ticks		

Table 40: Clock times

The above IEEE 1394 time can be converted to gPTP time, and the above gPTP time can be converted to IEEE 1394 time, as shown in Table 41.

IEEE 1394 → gPTP	gPTP → IEEE 1394
gPTP seconds = *bus time* = 12 s	*Bus time* = *gPTP seconds* % 2^{32} = 34 % 2^{23} = 34 s
gPTP nanoseconds = int ((*cycle count* × 125 × 10^3) + (*cycle offset* × (125 × 10^3 / 3072))) = int ((100 × 125 × 10^3) + (768 × (125 × 10^3 / 3072))) = 12,531,250 ns	*Cycle count* = int (*gPTP nanoseconds* / 125 × 10^3) = int (15,187,500 / 125 × 10^3) = int (121.5) = 121 cycles
	Cycle offset = int ((*gPTP nanoseconds* / (125 × 10^3 / 3072)) % 3072) = int ((15,187,500 / (125 × 10^3 / 3072)) % 3072) = 1536 ticks

Table 41: Converted clock times

Once both networks times have been converted to a common base, the difference between the two clocks can be calculated, as shown in Table 42.

Time difference: gPTP	Time difference: IEEE 1394
= (34 s 15,187,500 ns) – (12s 12,531,250 ns) = 22s 2,656,250 ns	= (34 s 121 cycles 1536 ticks) – (12 s 100 cycles 768 ticks) = 22 s 21 cycles 768 ticks

Table 42: Clock time differences

Assume that a CIP packet arrives at the IEEE 1394 interface of the audio gateway device (shortly after the readings in Table 40 were taken) with a *cycle count* of 12 cycles and a *cycle offset* of 2304 ticks. This means that presentation time of the time-stamped event is 12 s, 108 cycles and 2304 ticks. This is calculated as follows:

- *bus time*

= node's *bus time*

= 12 s

- *cycle count*

  = (node's *cycle count* & 0x1FF0) | SYT *cycle count*

  = (100 & 0x1FF0) | 12

  = 96 | 12

  = 108 cycles

- *cycle offset*

  = SYT *cycle offset*

  = 2304 ticks

The *cycle count* calculation replaces the lower four bits of the node's *cycle count* with the four bits of the *cycle count* field of the CIP packet's SYT field. The time value would be converted to a gPTP time of 12 s 13,593,750 ns. This value is calculated as follows:

- *gPTP seconds = bus time*

  = 12

- *gPTP nanoseconds* = int ((*cycle count* × 125 µs × 10^3) + (*cycle offset* × (125 µs × 10^3 / 3072)))

  = int ((108 × 125 µs × 10^3) + (2304 × (125 µs × 10^3 / 3072)))

  = int (13,500,000 + 93750)

  = 13,593,750 ns

The difference between the two clocks (22s 2,656,250 ns) is added to this to yield a presentation time of 34 s 16,250,000 ns. This time value is then converted to a nanosecond *avtp timestamp* (using the formula from Table 22 on page 149). The formula for this conversion is:

- *avtp timestamp*

  = (*gPTP seconds* × 10^9 + *gPTP nanoseconds*) % 2^{32}

  = (34 × 10^9 + 16,250,000) % 2^{32}

  = 3,951,478,928 ns

If an AVTP frame arrives at the Ethernet interface of the audio gateway device (shortly after the readings in Table 40 were taken) with an *avtp timestamp* value of 3,951,478,928 ns, this would mean that the presentation time of the time-stamped event is 34 s 16,250,000 ns. This is calculated as

follows (assume that at the time that the frame arrived, the gPTP nanoseconds advanced to 15,200,500 ns):

= ((*gPTP seconds* × 10^9 + *gPTP nanoseconds*) & 0xFFFFFFFFFFFF00000000) | *avtp timestamp*

= ((34 x 10^9 + 15,200,500) & 0xFFFFFFFFFFFF00000000) | 3,951,478,928

= 34,016,250,000

= 34 s 16,250,000 ns

This calculation replaces the lower 32-bits of the time with the timestamp value conveyed in the AVTP frame. The value converted to IEEE 1394 time is 34 s, 130 cycles, 0 ticks. This time value is calculated as follows:

- *bus time* = *gPTP seconds* % 2^{32}
  = 34 % 2^{32}
  = 34 s
- *cycle count* = int (*gPTP nanoseconds* / 125 µs × 10^3)
  = int (16,250,000 / 125 × 10^3)
  = int (130)
  = 130 cycles
- *cycle offset* = int ((*gPTP nanoseconds* / (125 µs × 10^3 / 3072)) % 3072)
  = int ((16,250,000 / (125 µs × 10^3 / 3072)) % 3072)
  = int (399,360 % 3072)
  = int (0)
  = 0 ticks

The difference between the two clocks (22 s, 21 cycles, 768 ticks) is subtracted from this value to yield a value of 12 s, 108 cycles, 2304 ticks. Thus, the value of the SYT field would be 12 cycles, 2304 ticks (the lower four bits of 108 is equal to 12).

10.4 MAAP Component

As part of this study, a reusable MAAP component was developed. This component is an implementation of MAAP (see Section 6.3 “AVTP Address Allocation”) and is responsible for the allocation of multicast MAC addresses for AVTP frame transmission. Each AVB capable device contains a MAAP component. The MAAP component defines a `MAAP` struct and a set of functions

that act on it, and a `MAAPAddressRange` struct and a set of functions that acts on it. The `MAAP` struct represents the MAAP protocol and is responsible for keeping track of the protocol's state. The `MAAPAddressRange` struct represents a range of dynamically allocatable MAAP MAC addresses.

10.4.1 `MAAP` struct

The AVB devices making use of MAAP create a `MAAP` struct, and initialise it with the `MAAP_init` function. The signature of the `MAAP_init` function is shown in Listing 2. The `maap` argument is a pointer to the `MAAP` struct to be initialised. The `ethIFName` argument is the name of the Ethernet interface on which MAAP will transmit and receive frames. This could be "eth0", for example. The `MAAP_AddressRangeNoLongerAvailableCallback` argument is a pointer to a user supplied callback function. When an address range that has been acquired by MAAP is no longer available for use (for example, if two networks are joined together, and another device announces that is it using a range of MAC addresses that MAAP has acquired), the application is notified of this via this callback. The callback function has a single argument, which is a pointer to a `MAAPAddressRange` struct. This struct represent the range of addresses that are no longer available.

```
int MAAP_init (
    MAAP * maap,
    char * ethIFName,
    MAAP_AddressRangeNoLongerAvailableCallback
        notAvailableCallback);
```

Listing 2: `MAAP_init` function

Once a `MAAP` struct has been successfully initialised, it can be used to acquire ranges of dynamically allocatable multicast MAC addresses. The MAAP component defines two functions for achieving this. These functions are shown in Listing 3. The `MAAP_generateRandomMACAddresses` function returns a randomly generated consecutive range of MAC addresses. The `maap` argument is a pointer to the `MAAP` struct previously initialised. The `numConsecutiveAddresses` argument is the number of addresses that the application is requesting. When this function is called, a random range of addresses are generated that do not conflict with any other addresses previously acquired by

this instance of MAAP, or any other address ranges recently announced by other instances of MAAP. MAAP then transmits *MAAP probe* messages to the network to probe the availability of the address range. If after three *MAAP probe* transmissions no *MAAP defend* messages are received for the address range, the address range is passed back to the caller via a `MAAPAddressRange` struct.

```
MAAPAddressRange * MAAP_generateRandomMACAddresses (
    MAAP * maap,
    uint16 numConsecutiveAddresses);

MAAPAddressRange * MAAP_tryGetMACAddresses (
    MAAP * maap,
    uint8 startAddress [6],
    uint16 numConsecutiveAddresses);
```

Listing 3: MAAP methods for acquiring MAC addresses

The `MAAP_tryGetMACAddresses` function tries to acquire the range of MAC addresses supplied by the caller. The `startAddress` argument is the first address of the range that the caller would like to acquire, and the `numConsecutiveAddresses` argument is the number of consecutive addresses that the caller would like to acquire. When this function is called, MAAP starts the probe procedure, and if no other instance of MAAP tries to defend the use of the address range, the address range is acquired. If another instance of MAAP defends the use of the address range, MAAP proceeds as if `MAAP_generateRandomMACAddresses` was called and returns a randomly generated address range.

10.4.2 `MAAPAddressRange` struct

The `MAAPAddressRange` struct represents a range of dynamically allocatable MAAP MAC addresses and is returned by the functions shown in Listing 3. The members of the `MAAPAddressRange` struct should be considered opaque and all interaction with the struct should be via the defined functions. The MAAP component defines the `MAAPAddressRange_getAddress` method that allows for the retrieval of a single MAC address out of a MAC address range represented by a `MAAPAddressRange` struct. This function is

shown in Listing 4. The `maapAddressRange` argument is the `MAAPAddressRange` struct which represents the range of addresses from which to get the address. The `index` argument is the index of the address to obtain, with 0 being the first address. The `address` argument is a user supplied buffer into which the address is placed. The `addressSize` argument is the size of the `address` buffer. The size of the buffer has to be at least 6 octets in size to ensure that a 48-bit MAC address can be copied into it.

```
int MAAPAddressRange_getAddresss (
    MAAPAddressRange * maapAddressRange,
    int index,
    uint8 address [6],
    int addressSize);
```

Listing 4: `MAAPAddressRange_getAddress` function

10.4.3 MAAP Utilisation

When an AVB device is initialised, the MAAP component is initialised and the application requests a range of dynamically allocatable MAAP MAC addresses. The number of requested MAC addresses is dependent on the number of AVTP stream outputs that the AVB component implements. Once MAAP returns a range of MAC addresses, a MAC address is allocated to each of the AVB component's outputs. The allocated MAC address is the address that the output will use to transmit its stream frames on (i.e., it is the address set in each Ethernet frame's *destination address* field).

10.5 MRP Component

Each AVB device contains a *multiple registration protocol* (MRP), *multiple VLAN registration protocol* (MVRP), *multiple MAC registration protocol* (MMRP), and *multiple stream reservation protocol* (MSRP) component. These components were implemented as Linux kernel modules. The MRP kernel module implements the MRP protocol (see Section 3.2.1 "Multiple Registration Protocol") for an end station. It is responsible for keeping track of attributes, and their declaration

and registration states. It is also responsible for communicating attribute declaration and registration states to a bridged LAN to make other MRP kernel modules aware of attribute declarations.

The MRP kernel module defines an `mrp` struct that represents the MRP protocol and keeps track of the protocols state. All of the members of the struct should be considered opaque. All interaction with the `mrp` struct occurs via a set of defined functions. An MRP application (e.g., MSRP) creates one of these `mrp` structs and initialises it with a call to the `mrp_init` function. The signature of the `mrp_init` function is shown in Listing 5.

```
int mrp_init (
    struct mrp * mrp,
    const u8 application_address [6],
    u16 ether_type,
    const char * port_name,
    struct mrp_ops * mrp_ops,
    int protocol_version,
    int use_attribute_list_length);
```

Listing 5: `mrp_init` function

The `mrp` argument is a pointer to the created `mrp` struct that is to be initialised. The `application_address` argument is the reserved MAC address that the application uses to transmit and receive Ethernet frames. Each MRP application defines its own unique MAC address (from a set of addresses reserved for such use) used for communication between instances of the MRP application. Currently, three MRP applications have been defined (MMRP, MVRP, and MSRP), and each one of these has been assigned a unique MAC address. These addresses have been defined in the MRP kernel module as shown in Listing 6. The MRP kernel module registers the supplied MAC addresses with the kernel in order to instruct the kernel to pass it any frames destined to that address.

```
const u8 mrp_application_address_mmrp [6] =
    {0x01, 0x80, 0xc2, 0x00, 0x00, 0x20};

const u8 mrp_application_address_mvrp [6] =
    {0x01, 0x80, 0xc2, 0x00, 0x00, 0x21};

const u8 mrp_application_address_msrp [6] =
    {0x01, 0x80, 0xc2, 0x00, 0x00, 0x22};
```

Listing 6: MRP application addresses

The `ether_type` argument is the EtherType that the MRP application uses for its frames. Each MRP application defines its own unique EtherType, as shown in Listing 7.

```
#define MRP_ETHERTYPE_MMRP 0x88F6
#define MRP_ETHERTYPE_MVRP 0x88F5
#define MRP_ETHERTYPE_MSRP 0x22EA
```

Listing 7: MRP application EtherTypes

The `port_name` argument is the name of the port that the MRP application will transmit and receive its frames on. For example, this could be the value “eth0”. The `mrp_ops` argument is a pointer to an `mrp_ops` struct that contains pointers to various user defined callback functions. These callbacks are discussed in Section 10.5.2 “MRP Callbacks”. The `protocol_version` argument is the version of the MRP application. Each MRP application defines the version number to use. The protocol version is carried within MRPDUs and allows for future extensions to the protocols. The `use_attribute_list_length` specifies whether the MRP application makes use of the *attribute list length* field inside each message of an MRPDU. The *attribute list length* field specifies the number of octets that the list of attributes within a message of an MRPDU uses. Currently, MMRP and MVRP do not make use of this field, whereas MSRP does.

Once an MRP application is done with an `mrp` struct, it calls `mrp_clean_up` passing it a pointer to the created `mrp` struct. The `mrp_clean_up` function’s signature is shown in Listing 8. This function is responsible for releasing any resources that were acquired by the MRP kernel module.

```
void mrp_clean_up (struct mrp * mrp);
```

Listing 8: `mrp_clean_up` function

10.5.1 Attribute Registration

An MRP application is able to request MAD to declare attributes, and withdraw attribute declarations. MRP defines the *MAD join request* and the *MAD leave request* service primitives which are represented in the MRP kernel module by the `mrp_mad_join_request_notify` function (shown in Listing 9) and `mrp_mad_leave_request_notify` function (shown in Listing 10).

When an MRP application would like to declare an attribute, it makes a call to `mrp_mad_join_request_notify`. The signature of this function is shown in Listing 9. The `mrp` argument is a pointer to the `mrp` struct that was previously initialised. The `attribute_type` argument is the numeric value that is used to uniquely identify the attribute that the application would like to declare. Each MRP application defines unique numeric values for the attributes that it defines. MSRP, for example, defines unique numeric values for *talker advertise*, *talker failed*, and *listener* attributes. The `attribute_value` argument represents an instance of the attribute identified by the `attribute_type` argument. The `four_packed_type` argument is only used by MSRP for *listener* attributes to specify their type (i.e., either *listener ready*, *listener ready failed*, or *listener failed)*. MSRP defines unique numeric values for each of the listener attribute types. The `new` argument is a Boolean argument whose value is used to indicate an explicitly signalled new attribute declaration. When an MRP application makes an attribute declaration, it may explicitly signal to other MRP applications that the declaration is new.

```
int mrp_mad_join_request_notify (
    struct mrp * mrp,
    int attribute_type,
    void * attribute_value,
    int four_packed_type,
    int new);
```

Listing 9: `mrp_mad_join_request_notify` function

When an MRP application would like to withdraw an attribute declaration, it makes a call to `mrp_mad_leave_request_notify`. The signature of this function is shown in Listing 10. The arguments have the same definitions as those defined for the `mrp_mad_join_request_notify` function call.

```
int mrp_mad_leave_request_notify (
    struct mrp * mrp,
    int attribute_type,
    void * attribute_value);
```

Listing 10: `mrp_mad_leave_request_notify` function

10.5.2 MRP Callbacks

The MRP kernel module defines a number of callback functions that have to be implemented by an MRP application. These callback functions either notify the MRP application of various events, or request application specific information. When an `mrp` struct is initialised, it expects a pointer to an `mrp_ops` struct that contains a number of function pointers defined by the MRP application. A portion of the `mrp_ops` struct is shown in Listing 11.

```
struct mrp_ops
{
    mrp_mad_join_indication  mad_join_indication;
    mrp_mad_leave_indication mad_leave_indication;
    ...
};
```

Listing 11: `mrp_ops` struct

Listing 11 only shows the two most important members of the `mrp_ops` struct. The definitions of these members are shown in Listing 12 and Listing 13. MRP defines the *MAD join indication* and the *MAD leave indication* service primitives to notify an MRP application of changes in attribute registration. These service primitives are represented by two callbacks. The *MAD join indication* service primitive is represented by the `mrp_mad_join_indication` callback (shown in Listing 12), and the *MAD leave indication* service primitive is represented by the `mrp_mad_leave_indication` callback (shown in Listing 13).

When an attribute is registered by an MRP application's MAD component, the MAD component notifies the participant of this registration via the `mrp_mad_join_indication` callback. The `mrp` argument is a pointer to the MRP application that the attribute registration is related to. The `attribute_type` argument is a numeric value indicating the attribute type. For example, MSRP defines a *talker advertise* attribute and it defines a unique numeric value to represent that attribute. The `attribute_value` argument is the actual value of the attribute instance. For example, this could be an MSRP application's *talker advertise* attribute with all of its fields. It is the responsibility of an MRP application to extract the fields out of the data buffer to which `attribute_value` points. The `four_packed_type` argument is a numeric value used by *listener* attributes to indicate their type (either *listener ready*, *listener ready failed*, or *listener failed*). The `new` argument is a Boolean argument whose value is used to indicate an explicitly signalled new declaration. When an MRP application makes an attribute declaration, it may explicitly signal that the declaration is new. When the attribute is registered with other participants on the LAN, they are notified of this via this `new` argument.

```
typedef int (*mrp_mad_join_indication) (
    struct mrp * mrp,
    int attribute_type,
    void * attribute_value,
    int four_packed_type,
    int new);
```

Listing 12: `mrp_mad_join_indication` callback function

When an attribute is deregistered by an MRP application's MAD component, it notifies the participant of this deregistration via the `mrp_mad_leave_indication` callback. The argument definitions of this callback are the same as those defined for the `mrp_mad_join_indication`.

```
typedef int (*mrp_mad_leave_indication) (
    struct mrp * mrp,
    int attribute_type,
    void * attribute_value,
    int four_packed_type);
```

Listing 13: `mrp_mad_leave_indication` callback function

10.5.3 MRP Kernel Module Usage

Figure 133 shows an example of how the MRP kernel module is used when declaring an attribute (in the simplest case). An MRP application has an attribute that it would like to declare and it invokes the `mrp_mad_join_request_notify` function for the attribute (indicated by the number 1). The MRP kernel module adds the attribute to a list of attributes that it is keeping track of (if the attribute is not already there), as indicated by the number 2 (in the MRP kernel module implementation, the attributes are kept in a linked list). A newly added attribute has its applicant and registrar state machines initialised by signalling them with a *being* event, and then the attribute's applicant state machine is signalled with a *join* event, as discussed in 3.2.1.3.2 "Applicant State Machine". The attribute declaration is communicated to the LAN segment that the MRP application is associated with and it is registered on other participants (indicated by the number 3). The participant adds the attribute to a list of attributes that it is keeping track of (if it not already in the list). A newly added attribute has its applicant and registrar state machines initialised by signalling

them with a *being* event, and then its registrar state machine is signalled with a *receive join empty* event, as discussed in Section 3.2.1.3.3 “Registrar State Machine”. Once the attribute has been registered, the attribute registration is communicated to the MRP application via the `mrp_mad_join_indication` callback function. The MRP application then decides on what to do with the attribute.

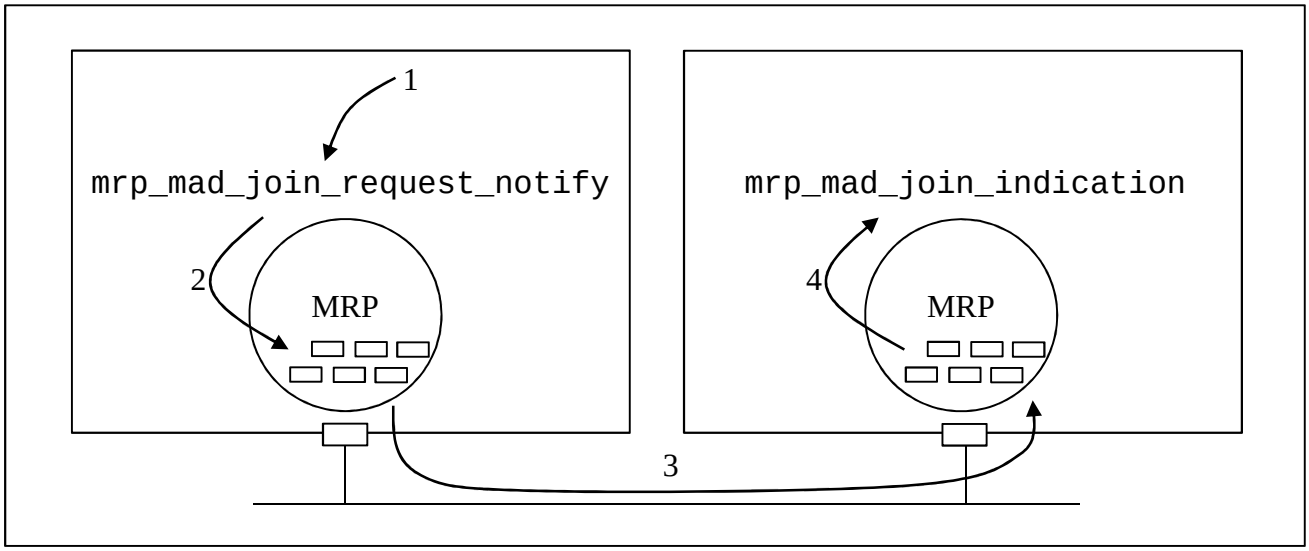

Figure 133: MRP kernel module attribute declaration

Figure 134 shows an example of how the MRP kernel module is used when withdrawing an attribute declaration (in the simplest case). When an MRP application would like to withdraw a declared attribute, it invokes the `mrp_mad_leave_request_notify` function for the attribute (indicated by the number 1). The MRP kernel module iterates through the attributes that it is keeping track of to locate the attribute (indicated by the number 2). Once the attribute has been located, the attribute’s applicant state machine is signalled with a *leave* event, as discussed in Section 3.2.1.3.2 “Applicant State Machine”. The withdrawal of the attribute declaration is communicated to the LAN segment (as indicated with the number 3). Any station on the LAN segment that has previously registered the attribute has the attribute’s registrar state machine signalled with a *receive leave* event which will cause the attribute to be deregistered (assuming that no other station on the LAN segment has declared the attribute). The attribute deregistration is indicated to the MRP application via the `mrp_mad_leave_indication` callback function (indicate by the number 4). The MRP application decides on how to behave when it receives notification of the attribute deregistration.

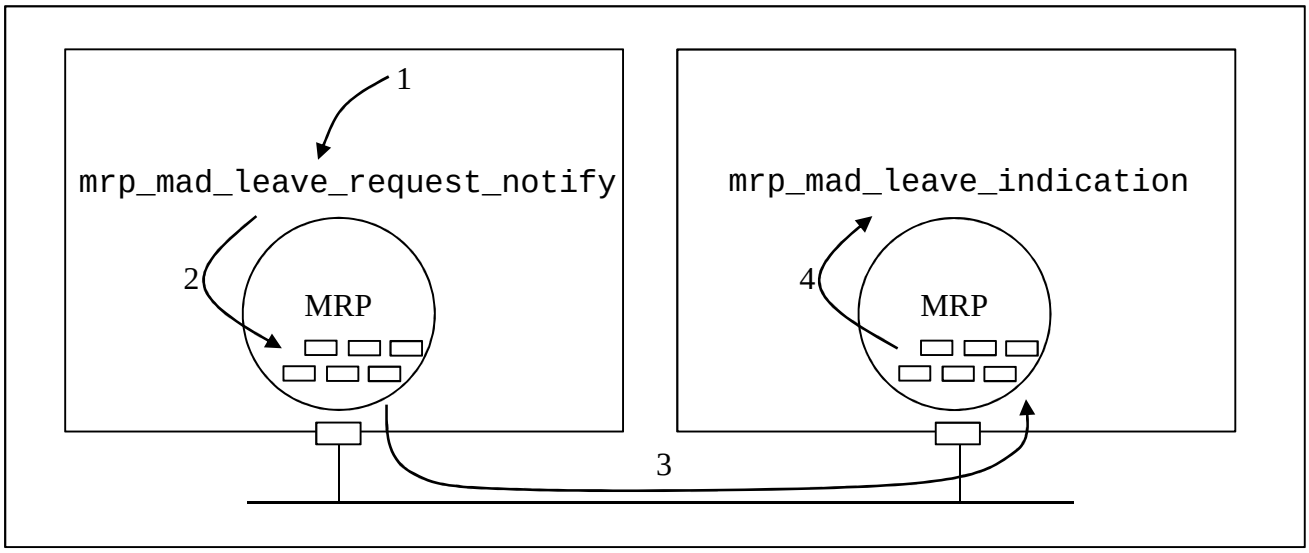

Figure 134: MRP kernel module attribute declaration withdrawal

10.6 MMRP and MVRP Kernel Modules

As part of this study, kernel modules were developed to represent MMRP, MVRP, and MSRP applications for end stations. This section discusses the MMRP and MVRP kernel modules, as they are very similar. Section 10.7 “MSRP Kernel Module” discusses the MSRP kernel module.

The MMRP and MVRP kernel modules make use of the functionality provided by the MRP kernel module. These modules are responsible for defining attribute types, the semantics associated with their attribute registrations, and for defining an interface to allow for other modules to interact with them.

MMRP is represented with an `mmrp` struct, and MVRP is represented with an `mvrp` struct. Each struct’s only member is an `mrp` struct. When each of these modules is loaded into the kernel, its `mrp` struct is initialised (they each run an independent copy of MRP). These structs are shown in Listing 14 and Listing 15.

```
struct mmrp
{
    struct mrp mrp;
};
```

Listing 14: `mmrp` struct

```
struct mvrp
{
    struct mrp mrp;
};
```

Listing 15: `mvrp` struct

The MMRP specification defines service primitives that allow for the registration of MAC addresses. These service primitives are represented in the MMRP kernel module with the functions shown in Listing 16.

```
int mmrp_register_mac_address (u8 mac_address [6]);

int mmrp_deregister_mac_address (u8 mac_address [6]);
```

Listing 16: MMRP functions

The `mmrp_register_mac_address` function represents MMRP's *register MAC address* service primitive. It allows an application to request the reception of frames with a certain destination MAC address. Figure 135 shows two stations on a LAN segment each having an MMRP kernel module. Each one of these MMRP kernel modules makes use of the functionality of an MRP kernel module. An application calls the `mmrp_register_mac_address` function and supplies it with the multicast MAC address. This function calls `mrp_mad_join_request_notify` in order to declare a MAC attribute. Other stations on the LAN segment register the attribute. The MMRP module is notified of the MAC attribute registration via the `mrp_mad_join_indication` callback function. When this callback is called within a bridge, the bridge will be configured to forward frames with the given MAC address out of the port that registered the attribute. An end station that would like to transmit frames with the MAC address can use the registration information

to decide whether or not any station is interested in receiving the frames (i.e., a lack of a MAC attribute registration for the MAC address indicates to the end station that there is no interest in frames destined to the MAC address).

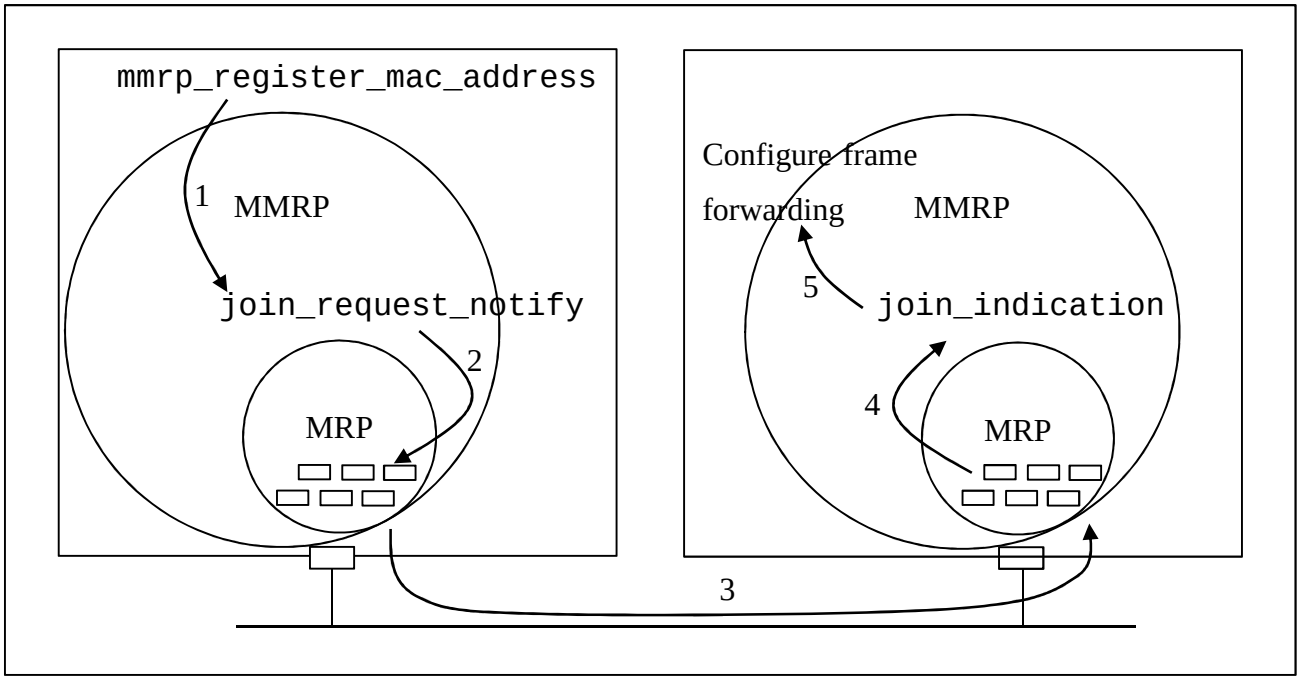

Figure 135: MMRP kernel module usage

The `mmrp_deregister_mac_address` function represents MMRP's *deregister MAC address* service primitive. When called, it withdraws the MAC attribute declaration for the supplied MAC address by calling `mrp_mad_leave_request_notify` for the MAC attribute. Any bridges that deregister the MAC attribute are configured to filter the frames with the given MAC address (that were previously forwarded towards the station that made the attribute declaration). When the attribute is deregistered from an end station, it can use this information to stop the transmission of frames destined to the MAC address as it knows that no stations are interested in receiving the frames.

The MVRP kernel module has two functions that are used to represent the service primitives of MVRP. These functions are shown in Listing 17. Each one of the functions has a single parameter that is used to represent a VLAN ID. The `mvrp_register_vlan_member` function is used by a station to become a member of a VLAN, and the `mvrp_deregister_vlan_member` is used by a station to deregister its VLAN membership.

```
int mvrp_register_vlan_member (u16 vid);

int mvrp_deregister_vlan_member (u16 vid);
```

Listing 17: MVRP functions

When the `mvrp_register_vlan_member` function is called, MVRP declares a VLAN attribute with a value that is equal to the requested VLAN ID. The declaration of the attribute is performed via the `mrp_mad_join_request_notify` function call. This ensures that the attribute declaration is communicated to the bridged LAN, which may result in the attribute being registered (if it is not already registered) on the ports of the bridges and other end stations of the bridged LAN. When the attribute is registered, the bridges and end stations are configured such that they will forward frames that are part of the VLAN towards the station that made the attribute declaration.

When the `mvrp_deregister_vlan_member` function is called, it withdraws the VLAN attribute declaration. This attribute declaration withdrawal happens via the `mrp_mad_leave_request_notify` function. The attribute declaration withdrawal is communicated to the bridged LAN, which may result in the attribute being deregistered from the ports of the bridges and end stations of the bridged LAN. The attribute will not be deregistered from all ports if other stations on the network have declared the same attribute. When the attribute is deregistered, the stations are configured to not forward frames that are part of the VLAN out of the port that deregistered the attribute.

10.7 MSRP Kernel Module

A kernel module was developed to represent an MSRP application for an end station. MSRP was implemented as a kernel module as it is responsible for configuring low-level network parameters (such as those of the credit-shaper algorithm (see Section 4.2.2 “Forwarding and Queuing”)). The kernel module has been implemented independently of any application that makes use of it. This promotes reusability of the module allowing multiple applications and future applications to make use of it.

The operation of MSRP is defined in Section 3.2.4 “Multiple Stream Reservation Protocol”. The MSRP kernel module makes use of the functionality of the MRP kernel module, as well as the functionality of the MVRP kernel module, and optionally the MMRP kernel module if *talker pruning* is enabled (*talker pruning* is used to limit the scope of *talker* attribute propagation to just those listeners interested in the stream on offer (see Section 3.2.4.2 “Talkers Advertising Streams”)). The MSRP kernel module defines an `msrp` struct, as shown in Listing 18. This listing shows the most important members of the `msrp` struct: an `mrp` struct and a pointer to an `msrp_ops` struct. When the kernel module is loaded into the kernel, the `msrp` struct is initialised, which includes the initialisation of the `mrp` struct.

```
struct msrp
{
    struct mrp mrp;
    struct msrp_ops * msrp_ops;
    ...
};
```

Listing 18: `msrp` struct

The `msrp_ops` struct contains pointers to user defined callback functions. An application that would like to make use of the MSRP kernel module defines the callback functions, and registers them with the MSRP kernel module. These callback functions allow the MSRP kernel module to notify the user of the module of any relevant events. These callback functions are discussed in Section 10.7.2 “MSRP Callback Functions”.

10.7.1 MSRP Functions

The MSRP kernel module has a number of functions that allow for talkers to register and deregister streams, and for listeners to register and deregister attachment to streams that are available from talkers. This section provides an overview of the functionality provided by the MSRP kernel module. Section 10.7.4 “Using the MSRP User-space Functions” discusses how the MSRP kernel module is used from user-space by the AVB devices that were developed during this study.

10.7.1.1 Registering Streams

When a talker has a stream to offer, it has to register this stream with an AVB network by declaring a *talker* attribute for the stream. The MSRP kernel module allows for talkers to register streams with the `msrp_register_stream_request` function. This function represents the MSRP *register stream request* service primitive. The `msrp_register_stream_request` function is shown in Listing 19.

```
int msrp_register_stream_request (
    u8 stream_id [8],
    int declaration_type,
    struct msrp_data_frame_parameters *
        msrp_data_frame_parameters,
    struct msrp_tspec * msrp_tspec,
    struct msrp_priority_and_rank * msrp_priority_and_rank,
    u32 accumulated_latency);
```

Listing 19: `msrp_register_stream_request` function

This function has the following arguments:

- `stream_id`: The value of the `stream_id` argument is the 64-bit stream ID used to uniquely identify the stream being advertised by the talker. It is up to the talker to generate the stream ID.
- `declaration_type`: The value of the `declaration_type` argument is a numeric value indicating the type of talker attribute that the talker should declare. The value of this argument indicates either an attribute declaration of *talker advertise,* or of *talker failed*. If the talker has sufficient resources to support the stream, the declaration type is set to *talker advertise*, otherwise it is set to *talker failed*.
- `msrp_data_frame_parameters`: The `msrp_data_frame_parameters` argument is a pointer to an `msrp_data_frame_parameters` struct. Listing 20 shows the members of this struct. The `destination_address` field's value contains the destination MAC address of the transmitted stream. The `vlan_identifier` field's value contains the VLAN ID of the VLAN that the stream will be transmitted on. In its default configuration, AVB uses VLAN 2.

```
struct msrp_data_frame_parameters
{
    uint8 destination_address [6];
    uint16 vlan_identifier;
};
```

Listing 20: `msrp_data_frame_parameters` struct

- `msrp_tspec`: The `msrp_tspec` argument is a pointer to an `msrp_tspec` struct. The members of this struct are shown in Listing 21. The `max_frame_size` field's value represents the maximum size that any frame that is part of the stream may be. The `max_interval_frames` field's value represents the maximum number of frames that will be transmitted in one *class measurement interval*. For a class A stream, the *class measurement interval* is 125µs, and for a class B stream, the *class measurement interval* is 250µs. The `max_frame_size` and the `max_interval_frames` values are used by bridges to calculate the stream's bandwidth requirements and to reserve resources for the stream.

```
struct msrp_tspec
{
    uint16 max_frame_size;
    uint16 max_interval_frames;
};
```

Listing 21: `msrp_tspec` struct

- `msrp_priority_and_rank`: The `msrp_priority_and_rank` argument is a pointer to an `msrp_priority_and_rank` struct. Listing 22 shows this struct (the listing shows Little-Endian bit fields and would need to be reversed when using Big-Endian bit fields). The `data_frame_priority` field's value represents the priority value that the stream will be tagged with. By default, class A streams are transmitted using priority three, and class B streams are transmitted using priority two. The `rank` field's value is used to determine a stream's importance. A lower numeric value has a higher importance that a higher numeric value. This could be used, for example, to allow the placement of an emergency phone call on a network that

is already being used to its capacity. A lower rank stream will be dropped in favour of the higher rank stream should resources be constrained.

```
struct msrp_priority_and_rank
{
     uint8 reserved : 4;
     uint8 rank : 1;
     uint8 data_frame_priority : 3;
};
```

Listing 22: `msrp_priority_and_rank` struct

- `accumulated_latency`: The value of this field represents the worst case latency that the stream will encounter from the talker to the listener. When a talker calls `msrp_register_stream_request`, it initially sets this argument's value to an amount that specifies the amount of latency that a stream's frames will encounter before being passed to the MAC service. MSRP then adds the maximum per-port per-traffic class latency that the frame may experience through the underlying MAC service to this argument.

When a talker makes a call to `msrp_register_stream_request`, MSRP makes a call to the MRP module's `mrp_mad_join_request_notify` function to request that a *talker* attribute be declared (the type of *talker* attribute is dependent on the value of the `declaration_type` argument). The fields of the *talker* attribute (as shown in Section 3.2.4.5.1 "*Talker* Attributes") are filled with the values passed into the `msrp_register_stream_request` function.

10.7.1.2 Deregistering Streams

When a talker no longer has, or wants to, offer one of its previously advertised streams, it calls the `msrp_deregister_stream_request` function. This function represents MSRP's *deregister stream request* service primitive, and its signature is shown in Listing 23. The caller of this function supplies the stream's stream ID as an argument to the function. Calling this function results in a call being made to `mrp_mad_leave_request_notify` for the *talker* attribute that has the supplied stream ID.

```
int msrp_deregister_stream_request (u8 stream_id [8]);
```

Listing 23: `msrp_deregister_stream_request` function

10.7.1.3 Receiving a Stream

When a listener would like to receive a particular stream, it makes a call to the MSRP kernel module's `msrp_register_attach_request` function. This function represents MSRP's *register attach request* service primitive, and its signature is shown in Listing 24.

```
int msrp_register_attach_request (
    u8 stream_id [8],
    int declaration_type);
```

Listing 24: `msrp_register_attach_request` function

The `stream_id` argument should be supplied with a value that is equal to the stream ID of the stream that it would like to receive (i.e., one of the streams that were advertised by a talker on an AVB network). The `declaration_type` argument should be set to the type of *listener* attribute that should be declared (a numeric value representing a *listener* type of either *ready*, *ready failed*, or *asking failed*). The value of the `declaration_type` argument is dependent on the type of *talker* attribute registered for the stream. A listener searches through its list of attributes for the *talker* attribute with the stream ID of the stream it would like to receive.

- If the *talker* attribute is not found, or a *talker failed* attribute is found, the `declaration_type` argument is set to *asking failed*.
- If the *talker* attribute is found and its type is *talker advertise*, the `declaration_type` argument is set to *ready*.

A call is made to `mrp_mad_join_request_notify` to request MAD to declare a *listener* attribute of the type specified by the `declaration_type` argument. The declared *listener* attribute's stream ID field is set to the value of the `stream_id` argument.

10.7.1.4 Stopping Stream Reception

If a listener is receiving a stream and it would like to stop this reception, it makes a call to the MSRP kernel module's `msrp_deregister_attach_request` function. This function represents MSRP's *deregister attach request* service primitive, and its signature is shown in Listing 25. An application supplies this function with the stream ID of the stream that it no longer would like to receive. This function then calls the `mrp_mad_leave_request_notify` function to withdraw the *listener* attribute that has the stream ID indicated by the `stream_id` argument.

```
int msrp_deregister_attach_request (
    u8 stream_id [8]);
```

Listing 25: `msrp_deregister_attach_request` function

10.7.2 MSRP Callback Functions

The MSRP kernel module defines a number of callback functions that a talker and/or listener should supply to it such that the module can notify it of certain events taking place. An `msrp_ops` struct has been defined that contains pointers to the callback functions. The members of this struct are shown in Listing 26. The callback functions' signatures are shown in Listing 28 through to Listing 31.

```
struct msrp_ops
{
    msrp_register_attach_indication
        register_attach_indication;
    msrp_deregister_attach_indication
        deregister_attach_indication;
    msrp_register_stream_indication
        register_stream_indication;
    msrp_deregister_stream_indication
        deregister_stream_indication;
};
```

Listing 26: `msrp_ops` struct

10.7.2.1 Callback Registration

The user of the MSRP kernel module should register an instance of an `msrp_ops` struct with the MSRP kernel module to supply the module with pointers to application defined callback functions. The MSRP kernel module defines the `msrp_register` function that allows an application to register the callback functions with the module. Listing 27 shows the signature of this function.

```
void msrp_register (struct msrp_ops * msrp_ops);
```

Listing 27: `msrp_register` function

10.7.2.2 Stream Registration Notification

Listing 28 shows the `msrp_register_stream_indication` callback function. This function is used to represent MSRP's *register stream indication* service primitive. When a talker advertises a stream on a bridged LAN, each listener is notified of the stream's presence via the `msrp_register_stream_indication` callback function. When a *talker* attribute is registered on a listener, the MRP kernel module notifies the MSRP kernel module of this attribute registration via the `mrp_mad_join_indication` callback function. The MSRP kernel module then in turn notifies the listener of the stream's presence via the `msrp_register_stream_indication` callback function. The definitions of the function's

arguments are the same as those defined for the `msrp_register_stream_request` function shown in Section 10.7.1.1 “Registering Streams”.

```
typedef int (*msrp_register_stream_indication) (
    u8 stream_id [8],
    int declaration_type,
    struct msrp_data_frame_parameters *
        msrp_data_frame_parameters,
    struct msrp_tspec * msrp_tspec,
    struct msrp_priority_and_rank * msrp_priority_and_rank,
    u32 accumulated_latency);
```

Listing 28: `msrp_register_stream_indication` function

10.7.2.3 Stream Deregistration Notification

Listing 29 shows the signature of the `msrp_deregister_stream_indication` callback function. This function represents MSRP’s *deregister stream indication* service primitive. When a stream is no longer being offered by a talker, it withdraws the *talker* attribute for the stream. When the *talker* attribute is deregistered from a listener, the listener’s MRP kernel module notifies its MSRP kernel module of this via the `mrp_mad_leave_indication` callback function. This in turn notifies the listener via the `msrp_deregister_stream_indication` callback function. The `stream_id` argument’s value is the stream ID of the stream that is no longer offered by the talker.

```
typedef int (*msrp_deregister_stream_indication) (
    u8 stream_id [8]);
```

Listing 29: `msrp_deregister_stream_indication` function

10.7.2.4 Stream Reception Notification

Listing 30 shows the signature of the `msrp_register_attach_indication` callback function. This function is used to represent MSRP’s *register attach indication* service primitive.

When a listener on a network requests attachment to a stream (by declaring a *listener* attribute), the *listener* attribute is registered on the talker's port. The MRP kernel module notifies the MSRP kernel module of the attribute registration via the `mrp_mad_join_indication` callback function. The talker that initially advertised the stream is then notified of this attachment request via the `msrp_register_attach_indication` callback function. The `stream_id` argument is the stream ID of the talker's stream that the listener is requesting attachment to. The `declaration_type` argument is a numeric value indicating the type of the *listener* attribute that was registered on the talker's port. The type of *listener* attribute is either *listener ready*, *listener ready failed*, or *listener asking failed*. When this callback function is called, and the `declaration_type` has a numeric value indicating that the *listener* attribute has a type of either *listener ready* or *listener ready failed*, the talker should start the transmission of the stream identified by the value of the `stream_id` parameter.

```
typedef int (*msrp_register_attach_indication) (
    u8 stream_id [8],
    int declaration_type);
```

Listing 30: `msrp_register_attach_indication` function

10.7.2.5 Stream Reception Deregistration Notification

Listing 31 shows the signature of the `msrp_deregister_attach_indication` callback function. This function is used to represent MSRP's *deregister attach indication* service primitive. When no more listeners are interested in receiving a particular stream (when all of the listeners that were receiving a particular stream have withdrawn their *listener* attribute for the stream), the *listener* attribute is deregistered from the talker. The MRP kernel module notifies the MSRP kernel module of this attribute deregistration via the `mrp_mad_leave_indication` callback function. The talker that is transmitting the stream is then notified that no listeners would like to receive the particular stream via the `msrp_deregister_attach_indication` callback function. The value of the `stream_id` argument is the stream ID of the stream that no more listeners are interested in receiving. When this callback function is called, the talker stops the transmission of the stream.

```
typedef int (*msrp_deregister_attach_indication) (
    u8 stream_id [8]);
```

Listing 31: `msrp_deregister_attach_indication` function

10.7.3 MSRP User-space Functions

The MSRP kernel module allows for user-space applications to interact with it via a set of defined user-space functions. These functions are very similar to those of the MSRP kernel module, and are briefly introduced here. The Ethernet AVB endpoint devices, and the IEEE 1394/Ethernet AVB audio gateway devices have been developed as user-space proof of concept applications. These applications interact with MSRP via the user-space functions. A user-space application that makes use of the functionality of MSRP creates an `MSRP` struct and passes it to the `MSRP_init` function for initialisation. The signature of the `MSRP_init` function is shown in Listing 32. The `MSRP_init` function also expects to receive pointers to application defined callback functions representing the indication service primitives defined by MSRP.

```
int MSRP_init (
    MSRP * msrp,
    MSRP_registerStreamIndication registerStreamIndication,
    MSRP_deregisterStreamIndication deregisterStreamIndication,
    MSRP_registerAttachIndication registerAttachIndication,
    MSRP_deregisterAttachIndication deregisterAttachIndication);
```

Listing 32: `MSRP_init` function

The signatures of the callback functions are shown in Listing 33. The arguments of these functions are the same as those defined for the MSRP kernel module (the argument names use a different naming convention but retain the same meanings).

```
typedef int (*MSRP_registerStreamIndication) (
    uint8 streamID [8],
    int declarationType,
    uint8 destinationAddress [6],
    uint16 vlanIdentifier,
    uint16 maxFrameSize,
    uint16 maxIntervalFrame,
    uint8 dataFramePriority,
    uint8 rank,
    uint32 accumulatedLatency);

typedef int (*MSRP_deregisterStreamIndication) (
    uint8 streamID[8]);

typedef int (*MSRP_registerAttachIndication) (
    uint8 streamID[8],
    int declarationType);

typedef int (*MSRP_deregisterAttachIndication) (
    uint8 streamID[8]);
```

Listing 33: MSRP callback functions

There is a set of user-space functions that allow user-space applications to register and deregister streams on AVB networks, and to register and deregister attachment to streams on offer by talkers on AVB networks. These functions are shown in Listing 34. These functions are similar to the functions of the kernel module. The first argument of each function is a pointer to the user-space `MSRP` struct to which the function call applies. All of the other arguments have the same definitions as those of the MSRP kernel module.

```
int MSRP_registerStreamRequest (
    MSRP * msrp,
    uint8 streamID [8],
    int declarationType,
    uint8 destinationAddress [6],
    uint16 vlanIdentifier,
    uint16 maxFrameSize,
    uint16 maxIntervalFrames,
    uint8 dataFramePriority,
    uint8 rank,
    uint32 accumulatedLatency);

int MSRP_deregisterStreamRequest (
    MSRP * msrp,
    uint8 streamID [8]);

int MSRP_registerAttachRequest (
    MSRP * msrp,
    uint8 streamID [8],
    int declarationType);

int MSRP_deregisterAttachRequest (
    MSRP * msrp,
    uint8 streamID [8]);
```

Listing 34: MSRP functions

Figure 136 shows how the `MSRP_registerStreamRequest` function would be used from user space. Calling this function results in the kernel module's `msrp_register_stream_request` function being called, which results in a *talker advertise* attribute being declared. When this attribute is registered on other stations, MSRP is notified via its `mrp_mad_join_indication` callback function. This results in kernel space and user space listener applications being notified of the presence of the stream via the `msrp_register_attach_indication` and `MSRP_registerAttachIndication` functions, respectively.

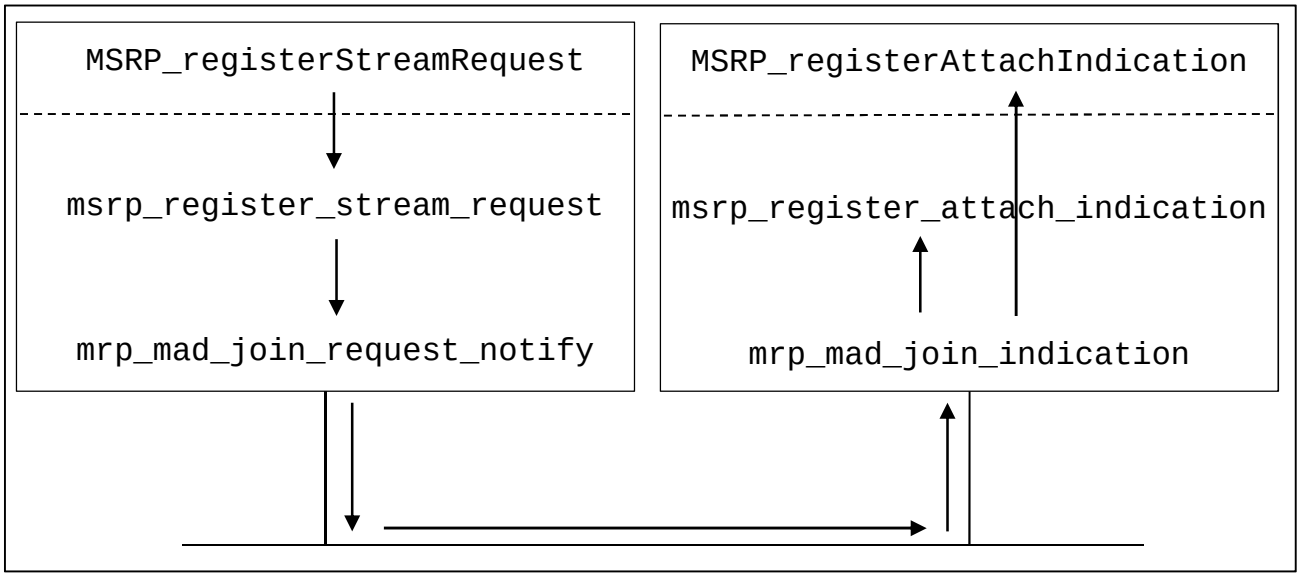

Figure 136: MSRP user-space usage

10.7.4 Using the MSRP User-space Functions

The Ethernet AVB devices that were developed as part of this study make use of the MSRP user-space functions. When one of the AVB devices has an AVTP stream to offer (represented by an output of the AVB component), it advertises this to an AVB network by calling the `MSRP_registerStreamRequest` function. This function has to be supplied with a number of arguments:

- `streamID`: At initialisation time, a unique stream ID is associated with each AVB output. When the AVTP stream (represented by the AVB output) is advertised to an AVB network, the stream ID associated with the output is supplied to the `streamID` argument.
- `declarationType`: When an AVB device initially advertises a stream to an AVB network, the `declarationType` argument is supplied with a value that represents a *talker advertise* declaration type.
- `destinationAddress`: When the AVB devices are initialised, each AVB output is assigned a unique multicast MAC address by the MAAP component. The `destinationAddress` argument is supplied with the value of this multicast MAC address.
- `vlanIdentifier`: By default, AVB streams are transmitted on VLAN 2, and this is the value that the AVB devices supply to this argument. It would be possible to allow this value to be user defined.

- `maxFrameSize`: Each output of the AVB component is aware of the number of sequences it transmits, and is aware of the number of events that it packs into each AVTP frame. It is also aware of the AVTP header size. With this information, each output is able to calculate the maximum size AVTP frame it will transmit. This value is supplied to the `maxFrameSize` argument.
- `maxIntervalFrames`: The AVB devices only implement class A streams (class A has a class measurement interval of 125 µs). The AVB devices transmit one frame per class measurement interval and thus set the value of this parameter to one.
- `dataFramePriority`: By default, class A streams are transmitted using priority three, and class B streams are transmitted using priority two (see Section 4.2.2.2 "Stream Reservation Traffic Classes"). The AVB devices provide this argument with the default value for class A streams. It would be possible to allow this value to be user defined.
- `rank`: For the AVB devices, each output has the `rank` argument set to one, which is the default value (see Section 3.2.4.5.1.4 "Priority and Rank").
- `accumulatedLatency`: No latency measurements were made for the AVB devices, and thus the value of this argument is set to zero.

When one of the AVB devices no longer has one of its streams to offer, it calls the `MSRP_deregisterStreamRequest` function and supplies it with the stream ID that identifies the stream. The stream ID value is maintained by the output (that represents the stream) of the AVB component.

When one of the AVB devices would like to receive a particular AVTP stream, it calls the `MSRP_registerAttachRequest` function which results in a *listener* attribute being declared for the requested stream. The `streamID` argument is supplied with the value of the stream ID of the talker's stream. This value has to be supplied to the application via a higher layer protocol. Chapter 11 "XFN Control and Representation" details how this is performed with the XFN protocol. The type of *listener* attribute that should be declared for the stream is specified via the `declarationType` argument. If a *talker advertise* has been received for the stream (as indicated by the `MSRP_registerStreamIndication`), the AVB devices supply this argument with a value that represents an attribute type of *listener ready*.

The Ethernet AVB devices implement the callback functions (representing MSRP's indication service primitives) shown in Listing 33. When the `MSRP_registerAttachIndication` callback function is called, it indicates to the device that at least one listener is interested in receiving one of its streams. The value of the `declarationType` argument specifies the type of *listener* attribute that was registered by the device for the stream. If the value indicates that the listener attribute is a *listener ready* or a *listener ready failed* attribute, then the device locates the output of the AVB component whose stream ID matches the value of the `streamID` argument. Once the output has been located, the transmission of the AVTP stream associated with output is started.

When the `MSRP_deregisterAttachIndication` callback function is called, it indicates to the device that the listener(s) previously interested in receiving the stream identified by the `streamID` argument are no longer interested in the reception, i.e., the *listener* attribute for the stream has been deregistered from the device. When this callback function is called, the device locates the output of the AVB component whose stream ID matches the value of the `streamID` argument. Once the output has been located, the transmission of the AVTP stream associated with the output is stopped.

10.8 Forwarding and Queuing (FAQ) Kernel Module

A kernel module has been implemented as a placeholder for the implementation of the forwarding and queuing rules specified in IEEE 802.1 Qav (see Section 4.2.2 "Forwarding and Queuing"). This module simply accepts Ethernet frames and transmits them out of the Ethernet interface of the Ethernet AVB devices. At the time of developing the Ethernet AVB devices, no suitable Ethernet network cards and drivers were available. The credit-shaper algorithm specified in IEEE 802.1 Qav requires that packets be transmitted at even intervals. Packets should not be transmitted in bursts, or clusters. A talker transmitting a class A stream transmits 8000 packets per second, and thus a packet transmission should occur once every 125 µs. This requirement places strict constraints on the behaviour of an implementation of the credit-shaper algorithm. Thus, dedicated hardware is required to meet the performance requirements of the credit-shaper algorithm.

10.9 AVB Device Interface

Each Ethernet AVB device is represented in software with an `AVBDevice` struct and a set of functions that acts on it. The `AVBDevice` struct represents an AVB device (it is able to represent either an endpoint device or an audio gateway device) and is responsible for coordinating the activities of the components of the device (see Figure 122 on page 228). Each Ethernet AVB audio device provides a function API interface that allows for control over its parameters and behaviour. This interface allows other modules (for example, a command and control module such as one representing the XFN protocol) to interact with the device and make adjustments to its parameters. This section provides an overview of the core functionality provided by this interface.

10.9.1 Internal Connections

Section 10.1 “Audio Components” discusses the general architecture that the AVB devices use to make connections between its audio inputs and audio outputs. Each audio device provides a function API interface that allows for associations between the inputs and outputs of the device to be obtained and for them to be created and destroyed.

The Ethernet AVB endpoint devices have a number of AVB stream inputs and analogue inputs, and a number of AVB stream outputs and analogue outputs. The IEEE 1394/Ethernet AVB audio gateway devices have a number of IEEE 1394 and AVB stream inputs, and a number of IEEE 1394 and AVB stream outputs. These devices were developed such that the details of these inputs and outputs (such as their type, and the formatting of the audio) are abstracted away from the interface. From the perspective of an external entity (such as an XFN entity), these are viewed as generic inputs and outputs.

Figure 137shows an example IEEE 1394/Ethernet AVB audio gateway device. This device has four audio inputs (two IEEE 1394 inputs, and two AVB inputs), and four audio outputs (two IEEE 1394 outputs, and two AVB outputs). Audio signals arriving at the inputs of the device are patched through to the outputs of the device. The input to output associations are shown in Table 43.

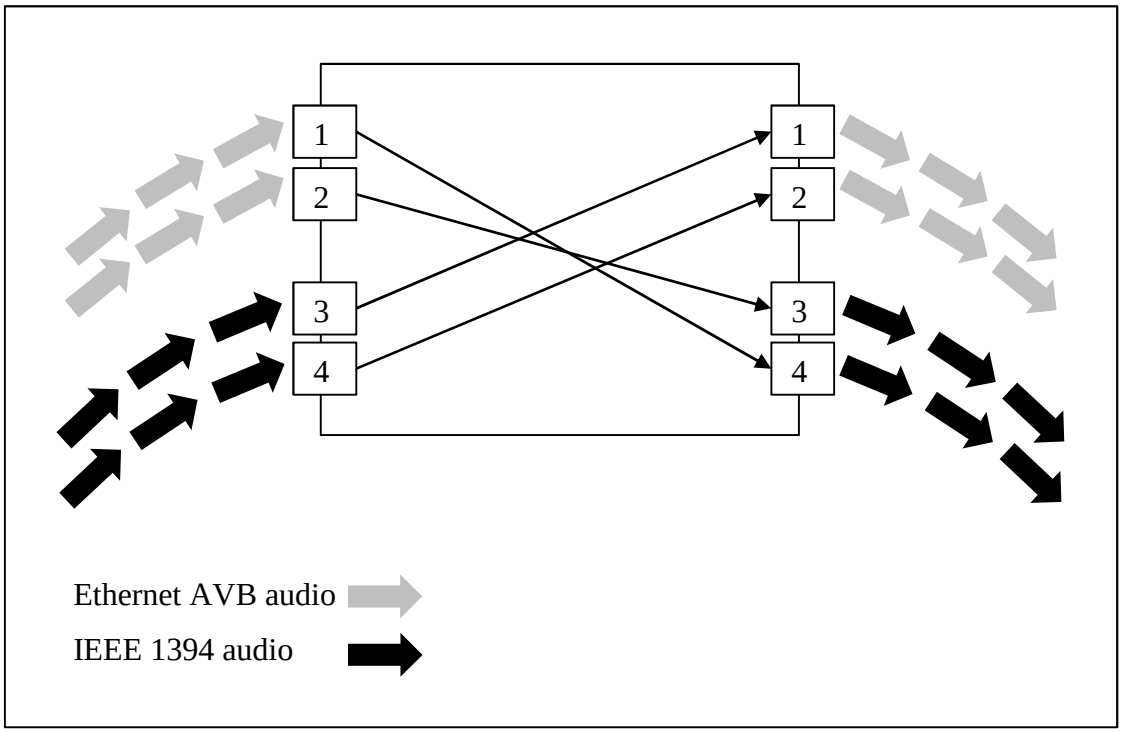

Figure 137: Generic inputs and outputs

Input	Output
1	4
2	3
3	1
4	2

Table 43: Input to output associations

Listing 35 shows the functions of an AVB device related to its audio inputs and outputs (both the Ethernet AVB endpoint devices and the IEEE 1394/Ethernet AVB audio gateway devices use the same function interface). The `AVBDevice_getNumGlobalInputs` function allows for the retrieval of the number of audio inputs that an AVB device has, and the `AVBDevice_getNumGlobalOutputs` function allows for the retrieval of the number of audio outputs that an AVB device has. If, for example, an Ethernet AVB endpoint device has an analogue input, and three AVTP stream inputs, then a call to `AVBDevice_getNumGlobalInputs` for the device would return the value four. The `avbDevice` argument is a pointer to an `AVBDevice` struct representing the Ethernet AVB device.

The `AVBDevice_getGlobalInputName` function allows for the retrieval of the name of a particular input of an AVB device, and the `AVBDevice_getGlobalOutputName` function allow for the retrieval of the name of a particular output of a device. The `inputIndex` argument should be supplied with the index of the input of interest. If, for example, an AVB device has four inputs, the `inputIndex` argument will accept values ranging from zero through to three, where zero is used to address the first input. Similarly, the `outputIndex` argument should be supplied with the index of the output of interest.

The `AVBDevice_getOutputInputPatchIndex` function allows for the retrieval of the index of the input that is patched through to a specific output (as specified by the `outputIndex` argument). The `AVBDevice_setOutputInputPatchIndex` function allows for patches to be created between inputs and outputs. The `outputIndex` argument should be supplied with the index of the output that is to be associated with the input. The `inputIndex` argument should be supplied with the index of the input which is to be associated with the output (as specified with the value of the `outputIndex` argument). In order to break a connection, a value of -1 is supplied to the `inputIndex` argument.

```
int AVBDevice_getNumGlobalInputs (
    AVBDevice * avbDevice);

int AVBDevice_getNumGlobalOutputs (
    AVBDevice * avbDevice);

char * AVBDevice_getGlobalInputName (
    AVBDevice * avbDevice,
    int inputIndex);

char * AVBDevice_getGlobalOutputName (
    AVBDevice * avbDevice,
    int outputIndex);

int AVBDevice_getOutputInputPatchIndex (
    AVBDevice * avbDevice,
    int outputIndex);

int AVBDevice_setOutputInputPatchIndex (
    AVBDevice * avbDevice,
    int outputIndex,
    int inputIndex);
```

Listing 35: Audio input and output functions

10.9.2 IEEE 1394 Interface

The IEEE 1394/Ethernet AVB audio gateway devices provide an interface that allows for isochronous stream connections to be established across IEEE 1394 buses. Isochronous streams are established when a transmitting IEEE 1394 device transmits an isochronous stream on a particular isochronous channel, and a receiving device is configured to receive the isochronous stream on that channel. Listing 36 shows the interface functions that were created for an audio gateway device.

```
int AVBDevice_get1394InputIsochChannelNumber (
     AVBDevice * avbDevice,
     int inputIndex);

int AVBDevice_get1394OutputIsochChannelNumber (
     AVBDevice * avbDevice,
     int outputIndex);

int AVBDevice_set1394InputIsochChannelNumber (
     AVBDevice * avbDevice,
     int inputIndex,
     int isochChannel);

int AVBDevice_set1394OutputIsochChannelNumber (
     AVBDevice * avbDevice,
     int outputIndex,
     int isochChannel);

int AVBDevice_start1394Output (
    AVBDevice * avbDevice,
    int outputIndex);

int AVBDevice_stop1394Output (
    AVBDevice * avbDevice,
    int outputIndex);

int AVBDevice_start1394Input (
    AVBDevice * avbDevice,
    int inputIndex);
```

Listing 36: Gateway IEEE 1394 functions

- `AVBDevice_get1394InputIsochChannelNumber`: This function allow for the retrieval of an IEEE 1394 input's isochronous channel number. The input isochronous channel number

represents the isochronous channel that the IEEE 1394 input should receive its stream on. The `inputIndex` argument should be supplied with the index of the IEEE 1394 input of the 1394 component (Figure 124 on page 232 shows an example 1394 component with inputs). The `AVBDevice_getNum1394Inputs` function allows for the retrieval of the number of IEEE 1394 stream inputs that an audio gateway device has. If the device has three IEEE 1394 stream inputs, the `inputIndex` argument would accept values from zero through to two where zero is used to address the first input.

- `AVBDevice_get1394OutputIsochChannelNumber`: This function allows for the retrieval of an IEEE 1394 output's isochronous channel number. The output isochronous channel number represents the isochronous channel number that is associated with the IEEE 1394 output stream. The `outputIndex` argument should be passed the index of the IEEE 1394 output of the 1394 component. The `AVBDevice_getNum1394Outputs` function allows for the retrieval of the number of IEEE 1394 stream outputs that an audio gateway has.
- `AVBDevice_set1394InputIsochChannelNumber`: This function allows the isochronous channel number of an IEEE 1394 input to be set. The `inputIndex` argument should be passed the index of the input of the 1394 component. The `isochChannel` argument should be set to the isochronous channel number that the input should receive its stream on.
- `AVBDevice_set1394OutputIsochChannelNumber`: This function allows the isochronous channel number of an IEEE 1394 output to be set. The `isochChannel` argument should be set to the isochronous channel number that the output's stream is transmitted on.
- The IEEE 1394/Ethernet AVB audio gateway devices also have a number of functions that allow isochronous streaming to be started and stopped. The `AVBDevice_start1394Output` and `AVBDevice_stop1394Output` functions start and stop the transmission of an isochronous stream for a particular IEEE 1394 output. The `AVBDevice_start1394Input` and `AVBDevice_stop1394Input` functions start and stop the reception of an isochronous stream for a particular IEEE 1394 input. Similarly, there are functions that allow for the current streaming state of the inputs and outputs to be obtained: `AVBDevice_is1394InputStarted` and `AVBDevice_is1394OutputStarted`.

10.9.3 AVB Interface

The AVB devices that were developed provide an interface that allows for the triggering of AVB stream advertisements and for triggering attachment to AVB streams. These functions for an AVB device are shown in Listing 37 (the functions are the same for the Ethernet AVB endpoint devices, and for the IEEE 1394/Ethernet AVB audio gateway devices).

```
int AVBDevice_registerAVTPStream (
     AVBDevice * avbDevice,
     int outputStreamIndex);

int AVBDevice_deregisterAVTPStream (
     AVBDevice * avbDevice,
     int outputStreamIndex);

int AVBDevice_registerAVTPStreamAttach (
     AVBDevice * avbDevice,
     int inputStreamIndex);

int AVBDevice_deregisterAVTPStreamAttach (
     AVBDevice * avbDevice,
     int inputStreamIndex);
```

Listing 37: Gateway stream registration functions

- The `AVBDevice_registerAVTPStream` function instructs the audio gateway to register an AVB stream (represented by an output of the AVB component) with the attached AVB network. The `outputStreamIndex` argument should be supplied with the index of the AVB output that is to be registered. Calling this function results in the output being located and the stream advertised to the AVB network via the MSRP component.
- The `AVBDevice_deregisterAVTPStream` function instructs the audio gateway to deregister an AVB stream from the attached AVB network. The `outputStreamIndex` argument should be supplied with the index of the AVB output that is to be deregistered. Calling this function results in the output being located, and the stream advertisement for the stream

being withdrawn. The request to withdraw the stream advertisement is handled by the MSRP component.

- The `AVBDevice_registerAVTPStreamAttach` function instructs the audio gateway to request attachment to a stream. The `inputStreamIndex` argument is the index of the input of the AVB component that is to receive the particular stream. Before calling `AVBDevice_registerAVTPStreamAttach`, the AVB input given as an argument needs to know the stream ID of the stream that it is to receive. The audio gateway defines the `AVBDevice_setInputStreamID` function (as shown in Listing 38) that allows for this to happen. Once the `AVBDevice_registerAVTPStreamAttach` function has been called, the input is located, and attachment to the stream is requested via the MSRP component.
- The `AVBDevice_deregisterAVTPStreamAttach` function instructs the audio gateway to detach from a stream that it is receiving. The `inputStreamIndex` argument is supplied with the index of the input of the AVB component that should stop receiving the stream. When this function is called, the input is located, and detachment from the stream is requested via the MSRP component.

```
int AVBDevice_setInputStreamID (
     AVBDevice * avbDevice,
     int inputIndex,
     uint8 streamID [8]);
```

Listing 38: `Gateway_setInputStreamID` function

10.10 Conclusion

This chapter provided a detailed overview of the implementation of the Ethernet AVB endpoint and IEEE 1394/Ethernet AVB audio gateway devices with a focus on how internal and external audio connections are established. Each device has a number of audio interfaces that are able to receive and transmit audio signals in various formats: an Ethernet AVB endpoint device allows for the transfer of audio between its analogue interface and its Ethernet interface (and vice versa), and the IEEE 1394/Ethernet AVB audio gateway device allows for the transfer of audio between its IEEE 1394 interface and its Ethernet interface (and vice versa). These devices provide a means to configure the internal audio routing between the audio interfaces via a set of functions.

The devices also allow for inter-device audio streaming to take place: the endpoint and audio gateway devices are able to transmit and receive audio signals over an Ethernet AVB network, and the audio gateway device is able to transmit and receive audio signals over an IEEE 1394 bus. These devices provide a means to establish stream connections across these networks and control over this connection management is provided via a set of functions.

The implementation of these devices, and the provision of the functions that allow for control over their parameters, allow for external modules to control the devices. Chapter 11 “XFN Control and Representation” details how control over these devices has been achieved with the XFN command and control protocol.

Chapter 11 XFN Control and Representation

Each of the audio devices used or developed during this study contains an XFN stack. This stack is an implementation of the XFN protocol (see Section 7.4 “XFN”). It allows for the creation of a seven-level XFN address hierarchy and allows for XFN network communication. Amongst other things, the stack provides a means for remote devices to obtain and set parameter values within a device. The core of the XFN stack was developed by *Universal Media Access Networks* (UMAN) [36] and was extended as part of this study.

One of the goals of the XFN protocol is to provide a common interface to a diverse range of devices. For each of the audio device types used in this study, an XFN address hierarchy has been defined. When each device is initialised, it builds up an XFN address hierarchy that represents the hierarchical structure of the device itself. Via the address hierarchy, it is possible to view the parameters that are available on a device, as well as obtain and set these parameter values. Through this mechanism, it is possible to configure audio streaming internally within a device, and also between devices.

This chapter discusses how the XFN protocol has been used to achieve the second goal of this work: provide a common view of the parameters of IEEE 1394 and Ethernet AVB audio devices to allow for remote control over these parameters. This remote control has been achieved via a graphical patchbay application.

11.1 XFN Stack Component

In the AVB audio devices that were developed, the reusable XFN Stack component (see Figure 122 on page 228) is represented with an `XFN` struct and a set of functions that operate on it. The XFN Stack component allows a device to build an XFN address hierarchy which enables remote controllers to obtain and set parameters of the device.

11.2 Graphical Representation of XFN Devices

Each of the audio devices introduced in Chapter 9 “Networked Audio Devices” is remotely controllable via the XFN protocol. Each device builds up an XFN address hierarchy that reflects the structure of the device with its parameters. These devices allow for their parameter values to be obtained and set remotely.

An application, known as the *Connection Manager*, was developed during this study to represent and control a number of XFN capable devices. This application makes use of the XFN command and control protocol to discover available XFN devices on a network, obtain their parameter values, and graphically represent them. Any changes to parameter values via the application’s *graphical user interface* (GUI) are communicated to the appropriate devices via XFN messages.

11.2.1 Connection Manager Architecture

Figure 138 shows the architecture of the Connection Manager. The Connection Manager consists of a GUI component that is used to represent XFN devices graphically, and also allows for user control over the parameters of the devices. The *problem domain component* (PDC) keeps track of devices and their parameters’ states. It accepts requests from the GUI component for parameter value changes, and also communicates to the GUI any parameter value changes, so that they may be represented graphically. The XFN stack component is responsible for all XFN network communication. It accepts requests from the PDC to obtain parameter values from remote devices and also set their parameter values. The XFN stack component communicates any responses back to the PDC.

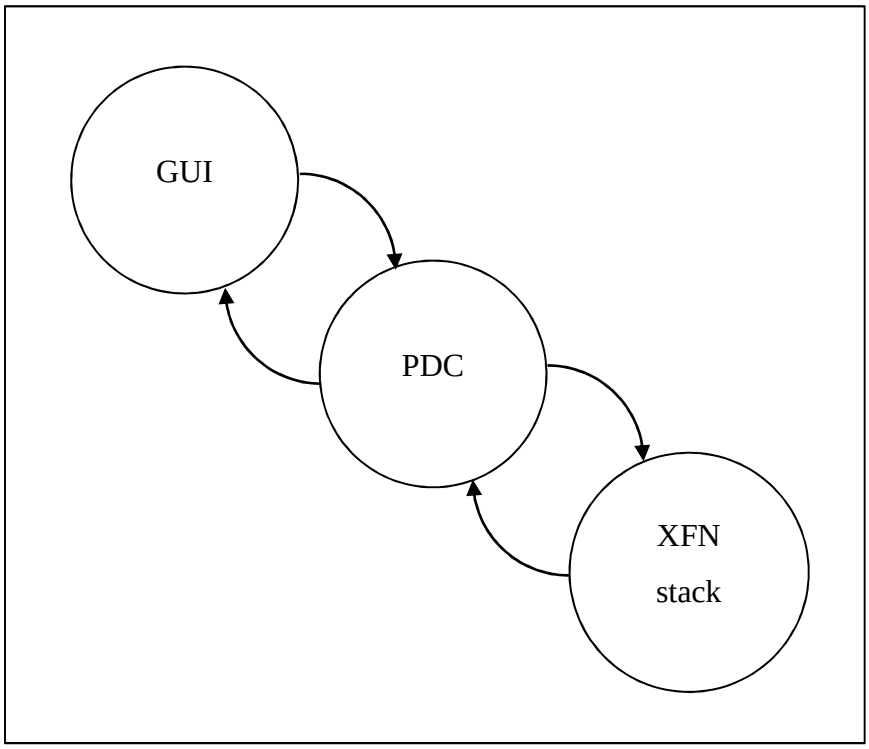

Figure 138: Connection Manager architecture

11.3 Building an XFN Address Hierarchy with the XFN Stack Component

The XFN Stack component of the AVB devices provides a set of functions that allow a device to build up an XFN address hierarchy to be used to address device parameters. The XFN protocol allows an XFN stack to have a number of nodes, each with its own address hierarchy. Each node would typically represent a unique device. These nodes are useful when a device is representing a number of other devices (a proxy device, for example). The functions used for building up an address hierarchy are shown in Listing 39.

```
int XFN_addApplicationNode ();

int XFN_addSectionBlockToLastAppNode (
    uint32 nodeID,
    char * levelAlias);

int XFN_addSectionTypeToLastSectionBlock (
    uint32 nodeID,
    char * levelAlias);

int XFN_addSectionNumberToLastSectionType (
    uint32 nodeID,
    char * levelAlias);

int XFN_addParameterBlockToLastSectionNumber (
    uint32 nodeID,
    char * levelAlias);

int XFN_addParameterBlockIndexToLastParameterBlock (
    uint32 nodeID,
    char * levelAlias);

int XFN_addParameterTypeToLastParameterBlockIndex (
    uint32 nodeID,
    char * levelAlias);

int XFN_addParameterIndexToLastParameterType (
    uint32 nodeID,
    char * levelAlias);

int XFN_addParameterToLastParameterIndex (
    uint8 valueFormat,
    XFNParameterCallback parameterCallback,
    void * applicationData);
```

Listing 39: XFN Stack component's address hierarchy building functions

When building up an address hierarchy with the XFN Stack component, an application first adds a top-level application node to the address hierarchy by calling the `XFN_addApplicationNode` function. The XFN Stack component keeps track of the application node that was last added to the stack. Figure 142 shows conceptually what the stack's address hierarchy looks like after this call. Once the node has been added, it is possible to start building up the address hierarchy for that node, starting from the top of the XFN address hierarchy structure.

Figure 139: An application node

- `XFN_addSectionBlockToLastAppNode`: This function allows for a section block to be added to the node that was last added to the stack. This function's `nodeID` argument is the value of the *section block* node of the address hierarchy, and the `levelAlias` argument is the node's alias. For example, an application could add an *input section block* to the address hierarchy which has a value of 0x01 (which is defined in the XFN specification), and is able to set the tree node's alias to anything it likes ("Inputs", for example). Aliases are used for meaningful display of the XFN address hierarchy. When a *section block* is added to the XFN Stack component, the stack keeps track of the last section block that was added. Figure 140 shows conceptually what the stack's address hierarchy looks like after adding a *section block*.

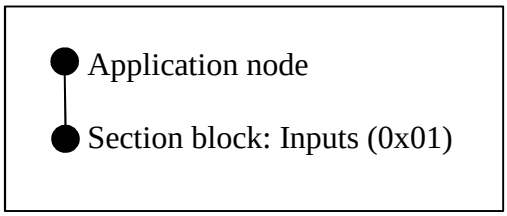

Figure 140: An application node with a *section block*

- `XFN_addSectionTypeToLastSectionBlock` : Once a *section block* has been added to the XFN address hierarchy, it is possible to add *section types* to it with this function. This function adds a *section type* to the *section block* that was last added to the XFN address hierarchy. The arguments to this function are similar to those of the `XFN_addSectionBlockToLastAppNode` function. The `nodeID` argument is the value of

the *section type* being added to the address hierarchy, and the `levelAlias` argument is an alias for the *section type* being added. For example, the XFN specification defines the value 0x04 to represent the *stream interface section type*. An application could give this an alias of “Stream interfaces”, for example. Figure 141 shows conceptually what the stack’s address hierarchy looks like after adding a *section type*.

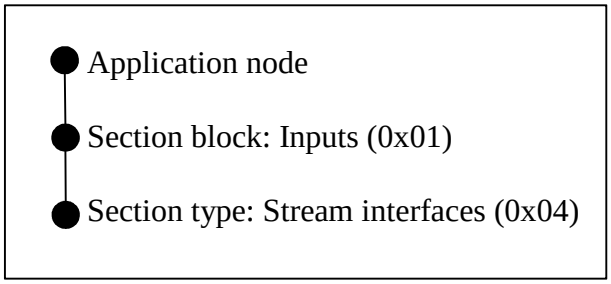

Figure 141: An application node with a *section block*

An application continues this process by adding a *section number*, *parameter block*, *parameter block index*, *parameter type*, and a *parameter index*. Once a *parameter index* has been added to the address hierarchy, it is possible to associate a parameter with it via the `XFN_addParameterToLastParameterIndex` function. The function expects the following arguments:

- The `valueFormat` argument of this function is a numeric value that is used to represent the format of the parameter’s value. The XFN specification defines values used to represent the various parameter value formats. Value formats are conveyed in XFN messages to allow for their correct interpretation on receipt. For example, the specification defines values to represent integers, floats, and data blocks of various sizes.
- The `parameterCallback` argument is a pointer to an application supplied callback function that is called when some action is performed on the parameter. For example, when a remote device tries to obtain or set the parameter’s value, this callback function is called to request the application to perform the action. The callback function specifies the specific action to perform. Figure 98 on page 197 shows the process of receiving an XFN message and the resulting parameter callback function call.
- The `applicationData` argument is a pointer to application data that is supplied to the callback function when it is called. An application could associate a single callback function with multiple parameters and use unique application data to identify the parameter that the invocation of the callback function is associated with.

11.4 Device Discovery

Device discovery is the process of discovering the available XFN capable devices that exist on a particular network. There is a need to be able to obtain the IP addresses of these devices such that communication with these devices can take place. As such, each XFN device builds up an address hierarchy that represents the IP addresses bound to the interfaces of the device. The address hierarchy that is built up to represent IP address parameters is shown in Figure 142:

- Each XFN capable device has a *configuration section block* under which any device configuration parameters may reside.
- Network interfaces are divided into interfaces through which audio and video data may be streamed, and those through which audio and video data may not be streamed (a management interface of a device may be classified as such, for example). Thus, a *stream interface section type* and a *non-stream interface section type* has been defined under which configuration parameters related to these interfaces are addressed.
- The *section number* level in the address hierarchy is used to address individual interfaces of each type.
- Each interface may have a number of IP related parameters associated with it (such as an IP address, subnet mask, and a default gateway). Thus, the *IP parameter block* has been defined to allow IP related parameters to be addressed.
- Each interface may have associated with it a number of IP addresses and the use of unique *parameter block indexes* are used to address each set of IP information.
- Under each unique *parameter block index* are listed the IP parameters that a device may have. These parameters include IP address, subnet mask, and default gateway parameters.
- The *parameter index* is not used for IP related parameters, and is set to one.

On an IEEE 1394 audio device, the IP addresses bound to the IEEE 1394 interfaces of the device are listed under the *stream interface section type*, and the IP address bound to the device's management interface is listed under the *non-stream interface section type*. Similarly, on the AVB devices developed during this study, the IP address bound to the Ethernet and IEEE 1394 interfaces of the devices are listed under the *stream interface section type*. The AVB devices do not have management

interfaces. These level hierarchies allow remote XFN capable devices to obtain the IP addresses bound to the interfaces of the XFN capable devices.

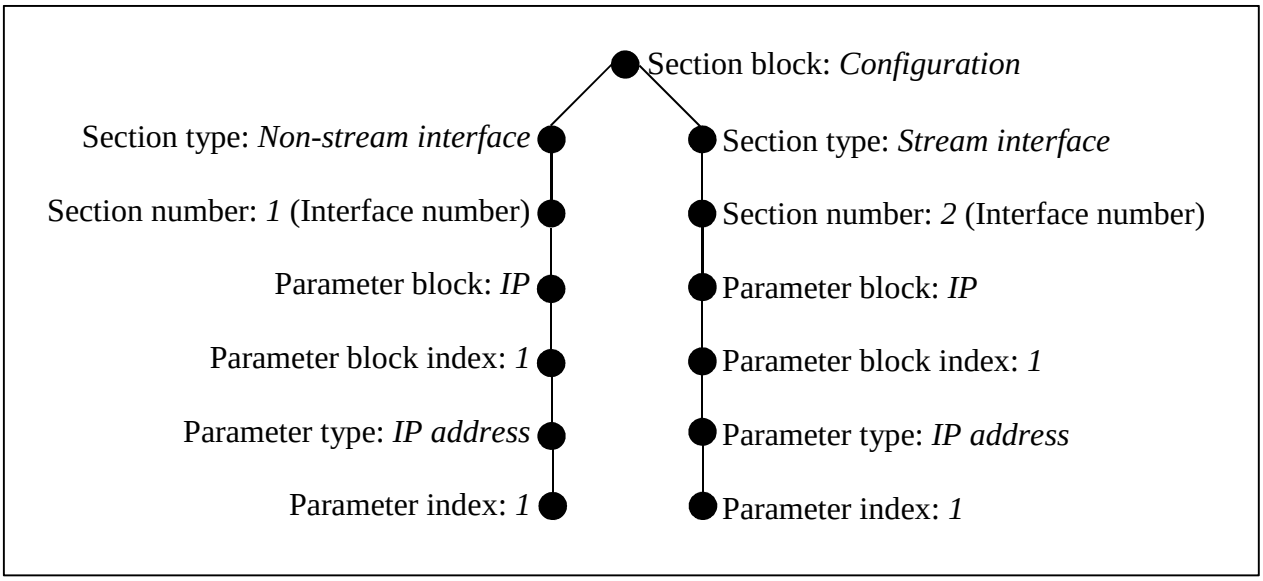

Figure 142: Portion of the XFN address hierarchy for representing IP addresses

Figure 143 shows how device discovery takes place with the XFN protocol. When an XFN capable device would like to discover other XFN capable devices on a network, it transmits a broadcast XFN *get value* message to the network for the IP address parameter values of each device (as shown at the top of the figure). Each device responds with a unicast message to the requester with the value(s) of its IP address parameter(s) (as shown at the bottom of the figure). Once an XFN device is aware of another XFN capable device, it can query it directly as it knows the device's IP address.

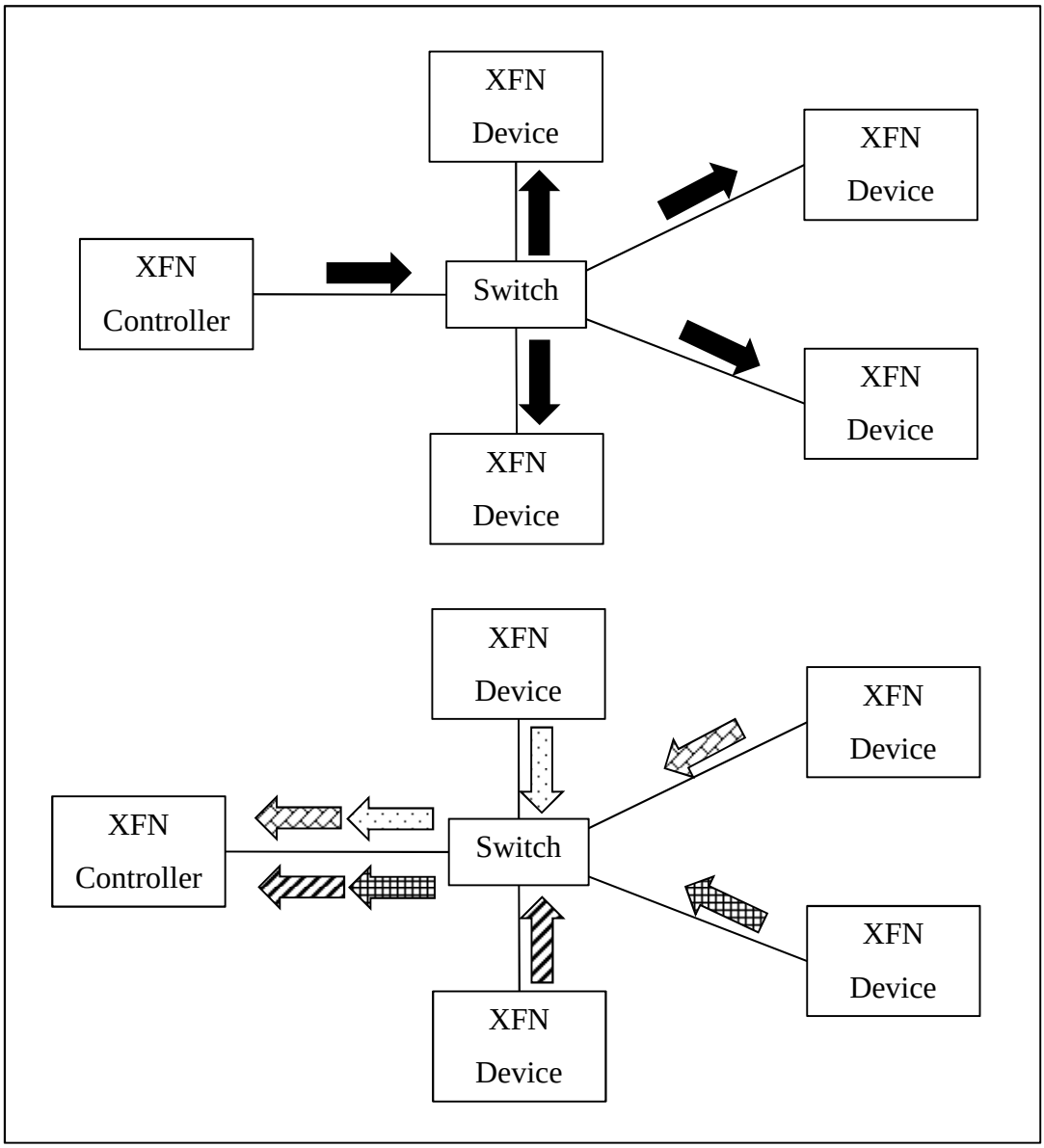

Figure 143: Device discovery

11.4.1 XFN Stack Component Device Discovery

The XFN Stack component of the Connection Manager (shown in Figure 138) is represented by an `XFN` object. This object inherits its core functionality from an implementation of the XFN protocol developed by UMAN and extends it. This object is responsible for all communication between the PDC of the Connection Manager and remote XFN devices. Once an `XFN` object has been created, it may be used to discover other XFN devices that exist on a network. The `XFN` object notifies the Connection Manager's PDC component of events via a listener interface. An `XFNListener` class has been defined that contains virtual functions that the PDC should implement. The PDC inherits

from the XFNListener class and overrides its functions. The PDC registers itself with the XFN object via the addXFNListener method of the XFN object (shown in Listing 40).

```
void addXFNListener (XFNListener * xfnListener);
```

Listing 40: addXFNListener function

The XFNListener class defines the ipDiscoverCallback virtual function (shown in Listing 41) that the PDC of the Connection Manager overrides. When the XFN object becomes aware of the presence of an XFN capable device, the Connection Manager is made aware of this device via this callback function. The ipAddress argument holds the value of the IP address of the discovered XFN capable device. The sessionID argument is a value supplied to the XFN object when it was requested to discover the available XFN devices on the network.

```
virtual void ipDiscoverCallback (
     const uint32 ipAddress,
     const int sessionID);
```

Listing 41: ipDiscoverCallback function

The XFN object defines the discoverIPAddresses function that allows the Connection Manager to discover the available XFN devices that are on a network. This function is shown in Listing 42. When this function is called, the XFN object sends out a broadcast packet (to the address specified by the broadcastAddress argument) containing an XFN *get value* command for all of the values of the *IP address parameter types*. Each device responds individually with its IP address(es). When this happens, the Connection Manager is notified via the ipDiscoverCallback function. The value of the sessionID argument is supplied to the ipDiscoverCallback function to allow requests and responses to be matched. Once the Connection Manager has been made aware of an XFN device, it is able to communicate with the device directly using its IP address.

```
const bool discoverIPAddresses (
     const uint32 broadcastAddress,
     const int sessionID);
```

Listing 42: `discoverIPAddresses` function

Figure 144 shows the sequence of events that takes place when the `XFN` object is called upon to discover the available XFN capable devices that exist on a network.

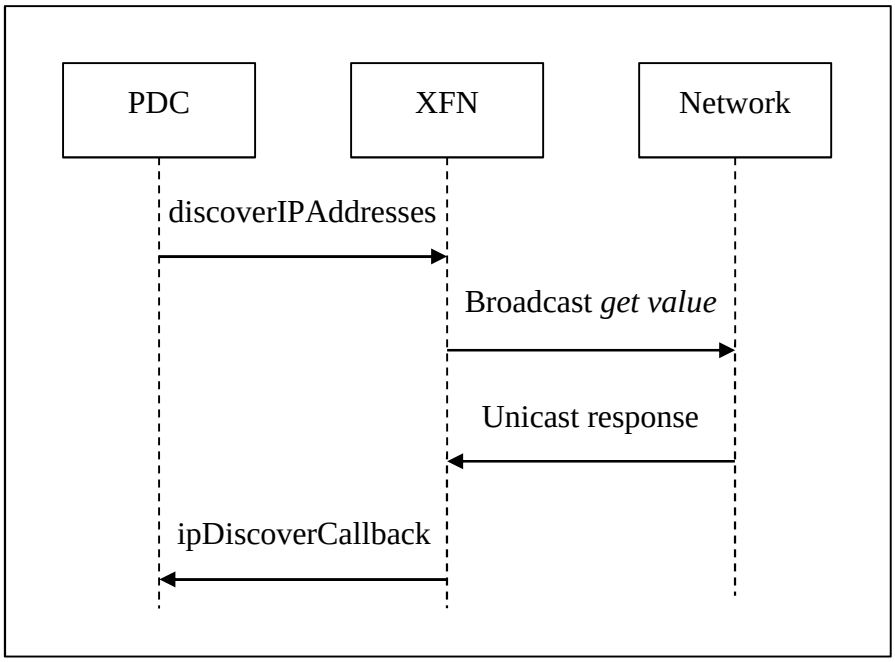

Figure 144: Device discovery sequence diagram

11.4.2 Graphical Representation of Discovered Devices

Figure 145 shows an example audio network composed of two IEEE 1394 buses joined together with an IEEE 1394 bridge, and one of the IEEE 1394 buses joined to an AVB network with an IEEE 1394/Ethernet AVB audio gateway device. The IEEE 1394 buses and the AVB network are each represented with a unique IP subnet. In this example, the IEEE 1394 buses are represented with the IP subnets 192.168.103.0/24 and 192.168.1.0/24, and the Ethernet AVB network is represented with the IP subnet 146.231.120.0/21. The XFN protocol allows a controlling device (for example, a control application running on a computer) to be located anywhere on the network. The figure shows a PC connected to one of the IEEE 1394 buses, and a PC connected to the Ethernet AVB network.

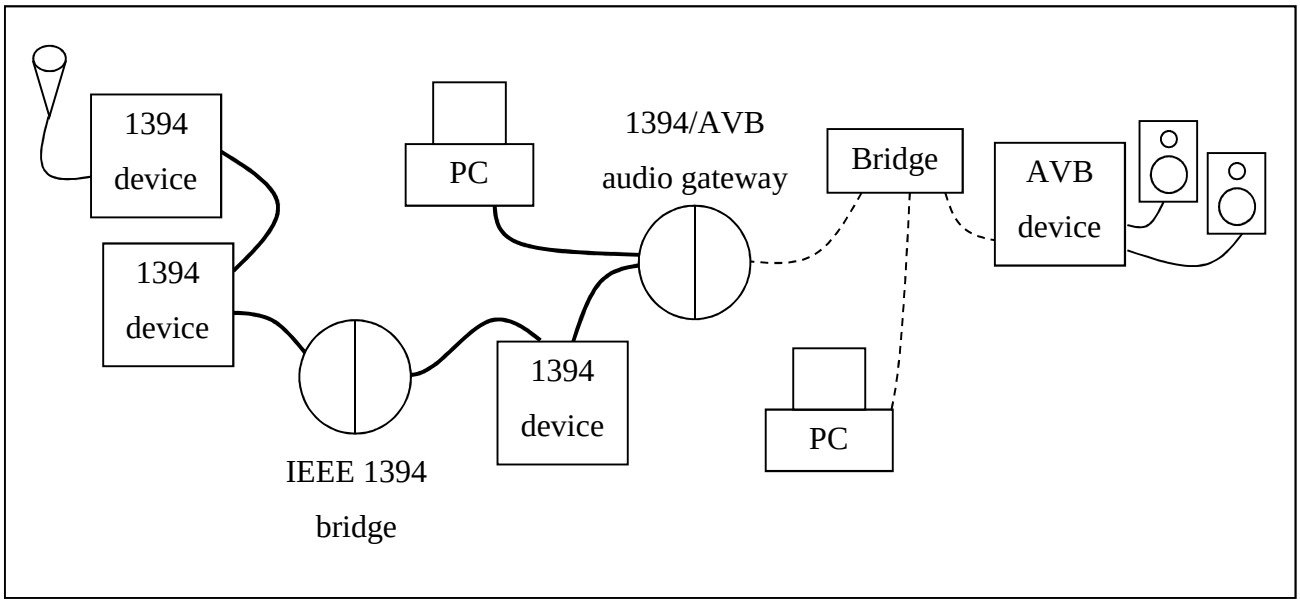

Figure 145: An example audio network

In Figure 146 is the main display of the Connection Manager demonstrating how the network shown in Figure 145 is graphically represented. When the Connection Manager is initialised, it discovers all of the XFN compatible devices that exist on the network. The discovery is performed by transmitting a broadcast XFN *get value* request for the values of the IP address parameters of all XFN devices. Once the Connection Manager has a device's IP address, it starts querying it. Initially, the Connection Manager finds out what subnet the device is on, and the name of the device. Once it has this information, it is able to graphically display it. XFN devices are organised into IP subnets and each IP subnet is represented as a tab along the top of the display. For each IP subnet, the devices are arranged along the axes of a matrix where the devices along the left hand side of the matrix are viewed as devices that produce audio streams, and the devices along the top of the matrix are viewed as devices that consume audio streams.

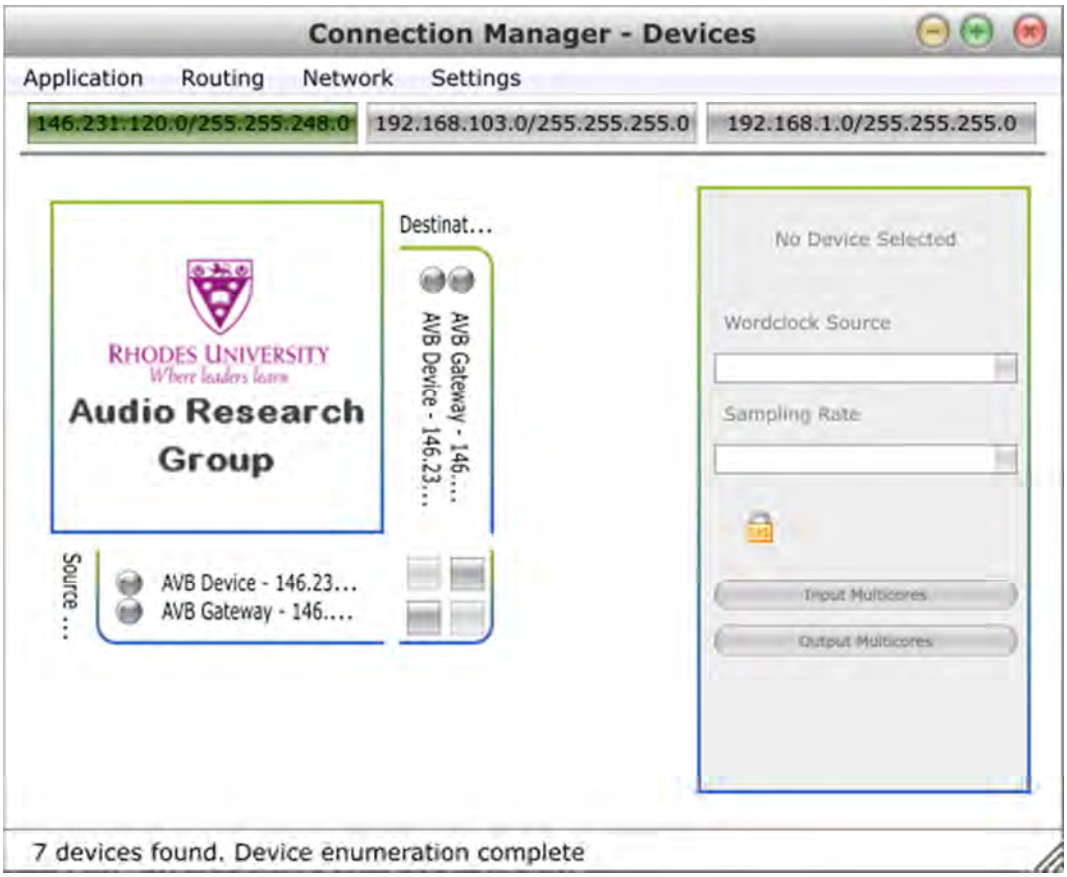

Figure 146: The Connection Manager networks and devices display

11.5 Internal Device Routing

The XFN specification defines the model for representing internal patching capabilities within a device whereby signal sources, signal destinations, and patch points are all modelled. Within a device, audio signals are patched from signal source points to signal destination points. Some signal destination points only accept a single source (e.g., an output of an audio mixer), and some signal destination points may accept multiple signal sources (e.g., a stereo bus of an audio mixer). Shown in Figure 147 is an example internal patching matrix of a device. This device has four signal source points (Input 01 to Input 04), and four signal destination points (Output 01, Output 02, Bus 01 and Bus 02). Shown in the figure are the patches that exist between the signal source points and signal destination points. The outputs are only able to accept one signal source at a time, and the buses are able to accept multiple sources at a time. Thus, the XFN protocol models every possible patch point that exists between the signal source and destination points within a device.

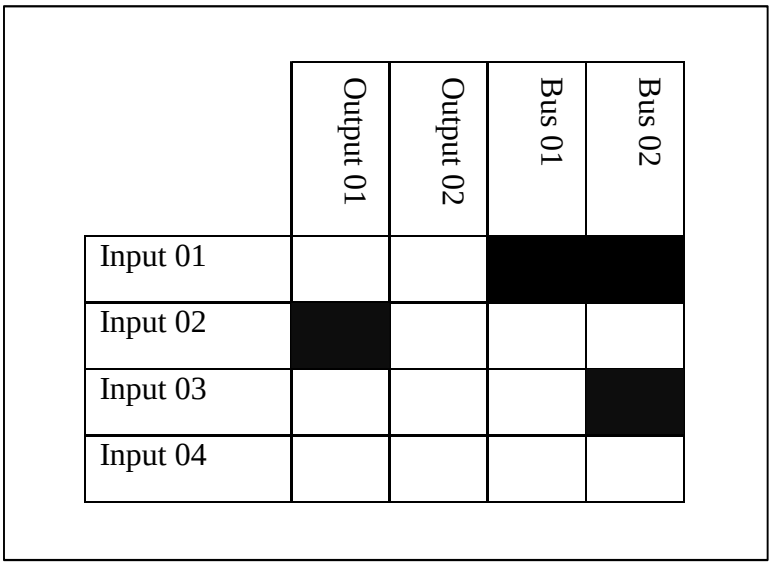

Figure 147: An example internal patching matrix

Figure 148 and Figure 149 show how the XFN specification defines how a patching matrix is modelled with the XFN protocol. Each input keeps track of the outputs to which it is patched, and each output keeps track of the inputs that are patched through to it.

Figure 148 shows how this is achieved for the inputs of a system with *crosspoint enable parameter types*:

- The audio inputs of a patching matrix are addressed under the *audio section type* of the *signal input section block.*
- Each input is indexed with a unique *section number* value (the figure shows two inputs).
- Each input contains an *output parameter block* which is used to represent the outputs of the patching matrix.
- Under each *output parameter block* is a unique *parameter block index* value for each output (the figure shows two outputs for each input).
- For each *parameter block index* there is a *crosspoint enable parameter type*.
- As there is only one *crosspoint enable parameter type* for each unique *parameter block index*, each *parameter index* has a value of one.

For the *signal input section block*, the *crosspoint enable parameter type* is a Boolean parameter used to specify whether or not the input (identified by the *section number*) is patched through to the output (identified by the *parameter block index*). A remote XFN device is able to query the *crosspoint enable* parameters to determine which inputs are patched through to which outputs. A remote XFN

capable device is also able to set the value of the *crosspoint enable* parameters in order to create patches between specific inputs and outputs.

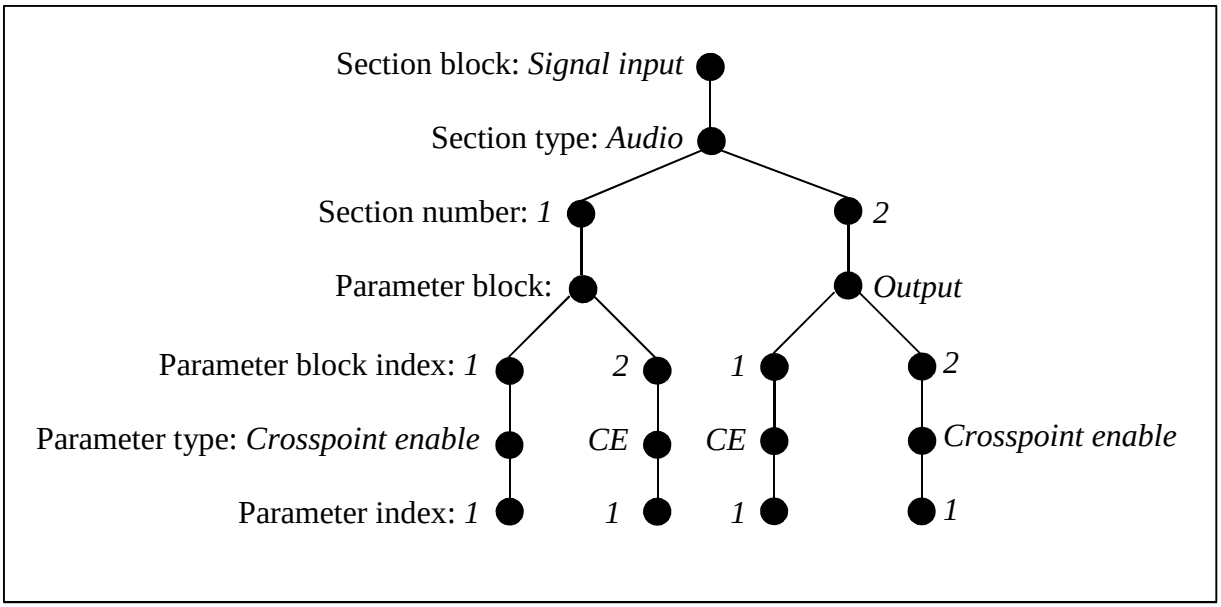

Figure 148: XFN input crosspoint modelling

Figure 149 shows how patching is achieved from the perspective of the outputs of a system with the *crosspoint enable parameter type*. The XFN address hierarchy for the outputs is similar to the XFN address hierarchy for the inputs (as shown in Figure 148), with a few exceptions:

- The outputs are listed under the *audio section type* of the *signal output section block*.
- For each output, there is an *input parameter block* under which all of the inputs are listed.
- Under each *input parameter block* is a unique *parameter block index* value for each input (the figure shows two inputs for each output).
- For the *signal output section block*, the *crosspoint enable parameter type* determines whether the output (identified by the *section number*) is receiving a signal from the input (identified by the *parameter block index*).

A remote XFN device is also able to query these *crosspoint enable* parameters to determine which inputs are patched through to which outputs. A remote XFN device is also able to set the value of these *crosspoint enable* parameters in order to create patches between inputs and outputs.

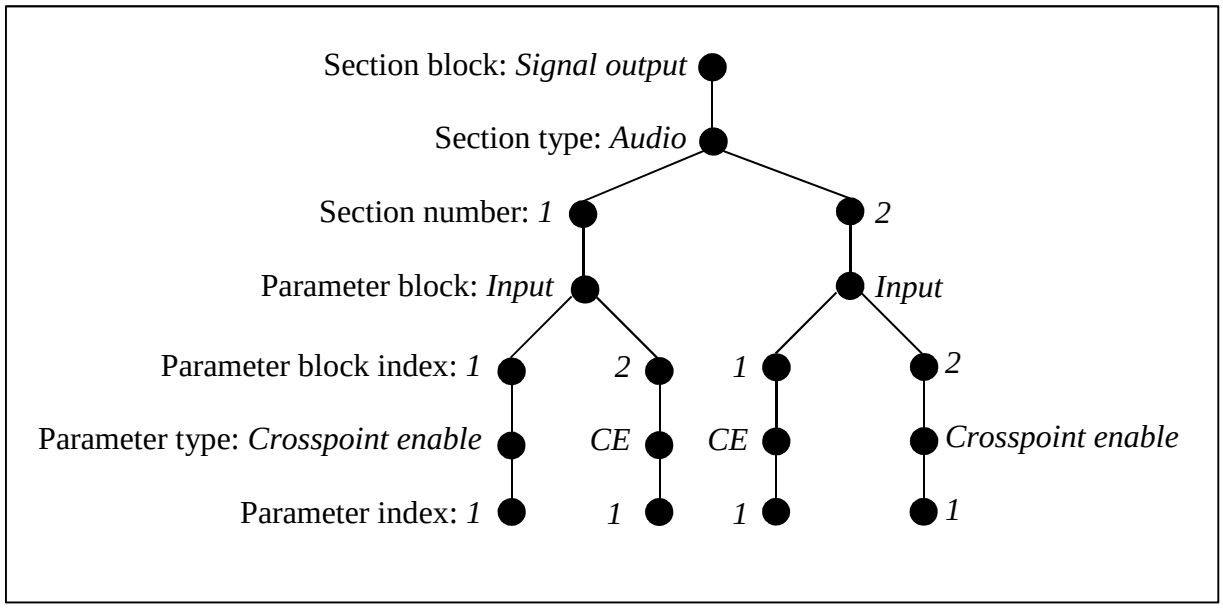

Figure 149: XFN output crosspoint modelling

The *crosspoint enable* parameters for the inputs and the outputs represent the same crosspoint control from different perspectives. The rationale for having *crosspoint enable* parameters for both the inputs and the outputs of an audio device is to allow for the tracing of audio signal paths through a device from any point within a network. With this approach, it is possible to trace an audio signal from its final destination through to its source, and it is possible to trace an audio signal from its source through to its final destination. It is therefore necessary for each output to be aware of the input(s) that it is receiving its signal(s) from, and for each input to be aware of the output(s) that it is sending its signal(s) to.

The IEEE 1394 and AVB devices have a number of audio inputs and outputs, and in these devices audio signals can be patched between these inputs and outputs. For example:

- IEEE 1394 audio devices: These devices have analogue, ADAT, and IEEE 1394 stream inputs and outputs.
- Ethernet AVB endpoint devices: These devices have an analogue input and output, and a number of Ethernet AVB stream inputs and outputs.
- IEEE 1394/Ethernet AVB gateway devices: These devices have a number of IEEE 1394 stream inputs and outputs, and a number of Ethernet AVB stream inputs and outputs.

The IEEE 1394 and Ethernet AVB devices build XFN address hierarchies (as shown in Figure 148 and Figure 149) for each of their inputs and outputs. This enables remote devices to create and destroy patches between the inputs and outputs of the devices.

In the Ethernet AVB capable devices, each one of the *crosspoint enable* parameters is associated with a callback function. When this function is called and a *get value* command is indicated to the function (as a result of a remote XFN device performing a *get value* request on the parameter), the `Gateway_getOutputInputPatchIndex` function of the device is called to determine if the addressed input is patched through to the addressed output. The device returns either a true or false indication to the requesting device. If the patch is enabled between the input and the output, the device responds with a true to the requesting device. If the patch is not enabled, the device responds with a false to the requesting device. When the parameter's callback function is called and a *set value* command is indicated to the callback function (as a result of a remote XFN device performing a *set value* request on the parameter), the `Gateway_setOutputInputPatchIndex` function is called in order to create or break a patch between the addressed input and output: a value of true indicates that a patch should be created, and a value of false indicates that a patch should be broken.

11.5.1 Connection Manager Representation of Internal Device Routing

Each XFN device that the Connection Manager discovers is enumerated to discover its capabilities and functionality. With the XFN protocol, it is possible to determine the number of child nodes that a particular node in an XFN address hierarchy has. Each XFN capable device is able to process a *get child node aliases* command. Associated with this command is the address of the XFN address hierarchy node whose child node aliases should be obtained. If, for example, a device builds up an XFN address hierarchy as shown in Figure 149, and a controller would like to know how many audio outputs the device has, the controller will sent a *get child node aliases* command to the device specifying the address of the *audio section type* of the *signal output section block*. The device responds with the number of child nodes that the addressed nodes has, and the aliases of each one of those nodes.

When the Connection Manager wants to know how many inputs a particular device has, it queries the device to determine how many child nodes the *audio section type* of the *signal input section block* has. Similarly, when the Connection Manager would like to know how many outputs a device

has, it queries the device to determine how many child nodes the *audio section type* of the *signal output section block* has. Once these numbers have been obtained, it is possible to query the parameters of each individual input (each represented with a unique *section number* of the *audio section type* of the *signal input section block*), and it is possible to query the parameters of each output (each represented with a unique *section number* of the *audio section type* of the *signal output section block*).

Listing 43 shows two functions of the Connection Manager's XFN Stack component. The `getNumAudioInputs` function allows the Connection Manager to query a specific device to determine the number of inputs it has. The `getNumAudioOutputs` function allows the Connection Manager to query a specific device to determine the number of outputs it has. In each instance, the `theDeviceIPAddress` argument should be supplied with the IP address of the device that is being queried.

```
const int getNumAudioInputs (
     const uint32 theDeviceIPAddress);

const int getNumAudioOutputs (
     const uint32 theDeviceIPAddress);
```

Listing 43: Obtaining input and output numbers

Each input and each output has associated with it a *matrix pin name parameter type*. By default, the devices assign names to their inputs and outputs and the *matrix pin name parameter type* allows for retrieval of these. For example, the analogue input of an Ethernet AVB endpoint device is named *Analogue Input 1*, and the first AVTP input is named *AVTP Input Multicore 1*. The Connection Manager retrieves the name of each input and output, and the state of the patches between these inputs and outputs and stores them. Listing 44 shows two functions of the Connection Manager's XFN Stack component that allow for the retrieval of input and output names of remote XFN capable devices. The `getAudioInputName` function returns the name of a specific input on a particular device. The `getAudioOutputName` function returns the name of a specific output on a particular device. The `deviceIPAddress` argument should be supplied with the IP address of the device that is being queried and the `inputIndex` and the `outputIndex` arguments should be supplied with the input's index and output's index, respectively.

```
const String getAudioInputName (
     const uint32 deviceIPAddress,
     int inputIndex);

const String getAudioOutputName (
     const uint32 deviceIPAddress,
     int outputIndex);
```

Listing 44: Obtaining input and output names

Listing 45 shows two functions of the Connection Manager's XFN Stack component that allow for obtaining and setting patches between a particular device's inputs and outputs. The `getAudioOutputInputPatch` function returns the state of a patch between a particular input (as specified by the `audioInputIndex` argument) and a particular output (as specified by the `audioOutputIndex` argument). This function returns the value of the *crosspoint enable* parameter for the specified input and output. The `setAudioOutputInputPatch` function sets the state of a patch between a particular input and a particular output as specified by the value of the `isPatched` argument. If the value of the `isPatched` argument is true, the patch is created. If the value of the `isPatched` argument is false, the patch is broken. This happens by setting the value of the *crosspoint enable* parameter for the specified input and output.

```
const bool getAudioOutputInputPatch (
     const uint32 deviceIPAddress,
     const int audioOutputIndex,
     const int audioInputIndex);

const bool setAudioOutputInputPatch (
     const uint32 deviceIPAddress,
     const int audioOutputIndex,
     const int audioInputIndex,
     const bool isPatched);
```

Listing 45: Obtaining and setting patches

With the Connection Manager, it is possible to view the retrieved inputs, outputs, and patches. When one of the devices on the Connection Manager's display is selected, a graphical routing matrix is displayed that shows these inputs, outputs and the device's internal routing configurations, as shown in Figure 150, Figure 151 and in Figure 152. In these routing matrices, the labels along the left hand side are signal inputs to the routing matrix, and thus are viewed as signal source points internally within the device. The labels along the top of the routing matrix are signal outputs from the routing matrix, and thus are viewed as signal destination points internally within the device. The cross-points on the routing matrix show whether or not inputs are patched through to particular outputs.

Figure 150 shows the internal routing matrix of an IEEE 1394 endpoint device. Shown are various analogue inputs being patched to sequences of output multicores, and the third and fourth ADAT inputs being patched to sequence seven and eight of the first output multicore.

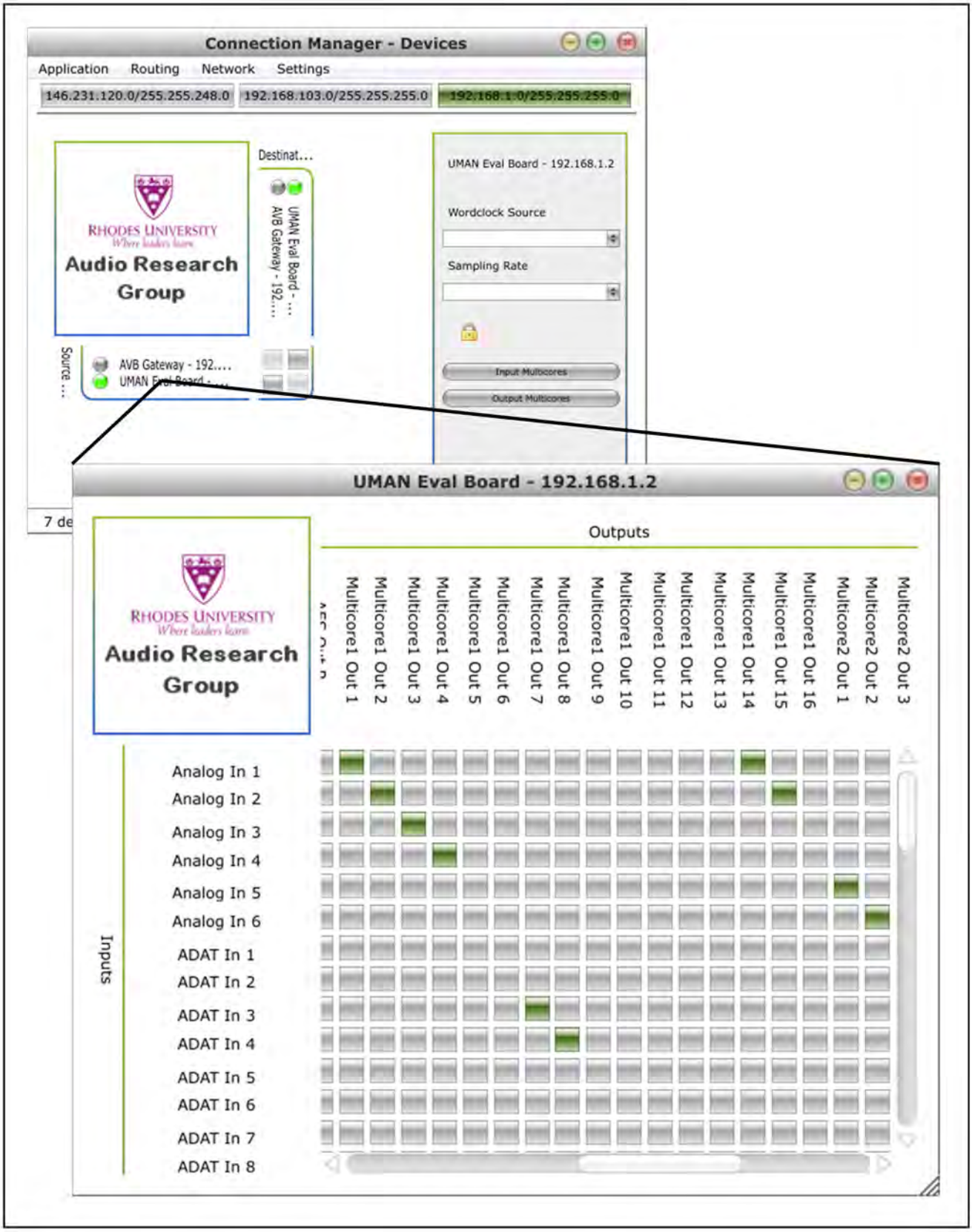

Figure 150: IEEE 1394 endpoint device internal routing matrix

Figure 151 shows the internal routing matrix of an Ethernet AVB endpoint device. Shown is one of the input AVB multicores being patched through to the device's analogue output.

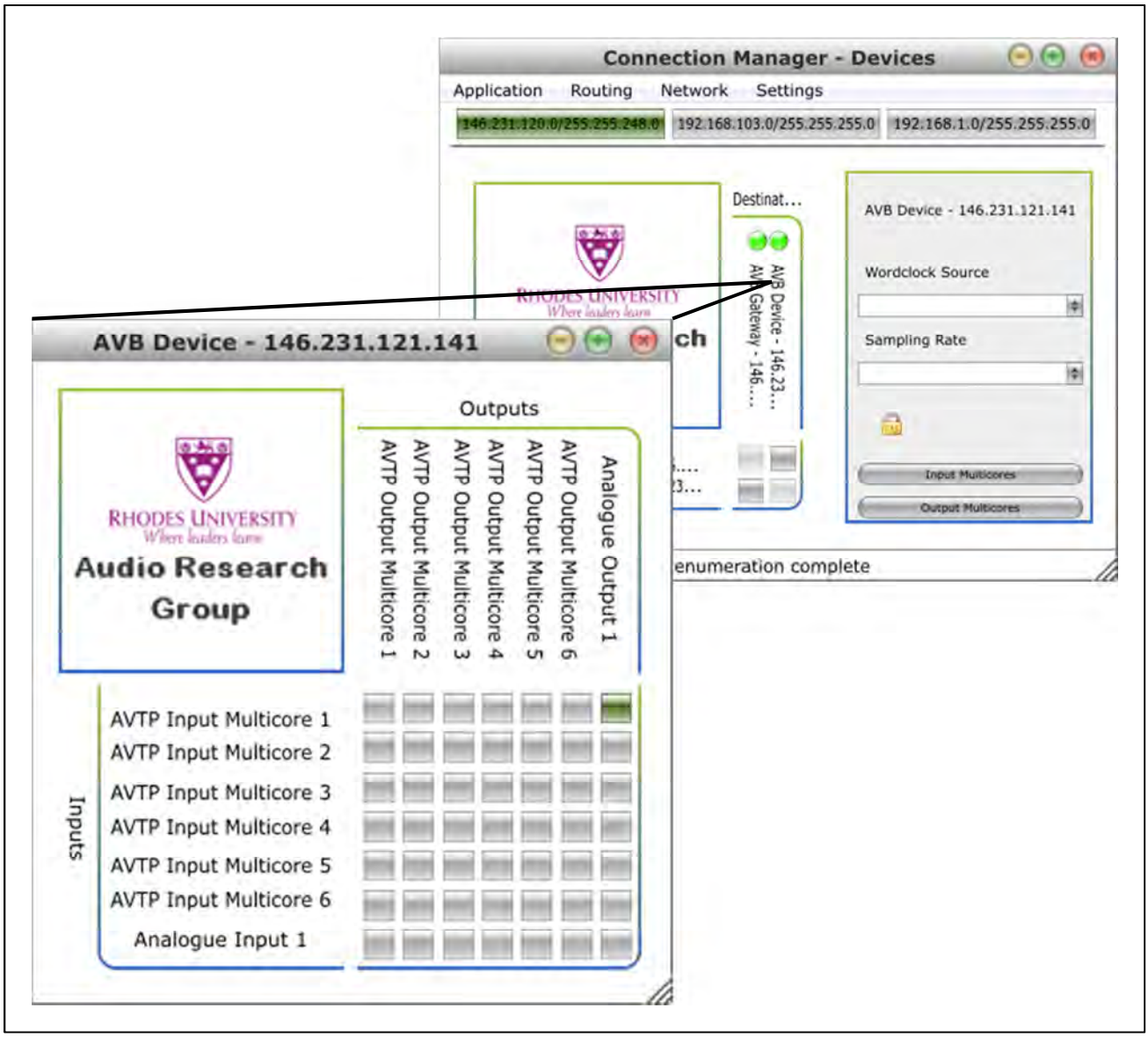

Figure 151: Ethernet AVB endpoint device internal routing matrix

Figure 152 shows the internal routing matrix of an IEEE 1394/AVB audio gateway device. This device has been configured to route the audio arriving at 1394 input multicores to AVB output multicores, and vice versa.

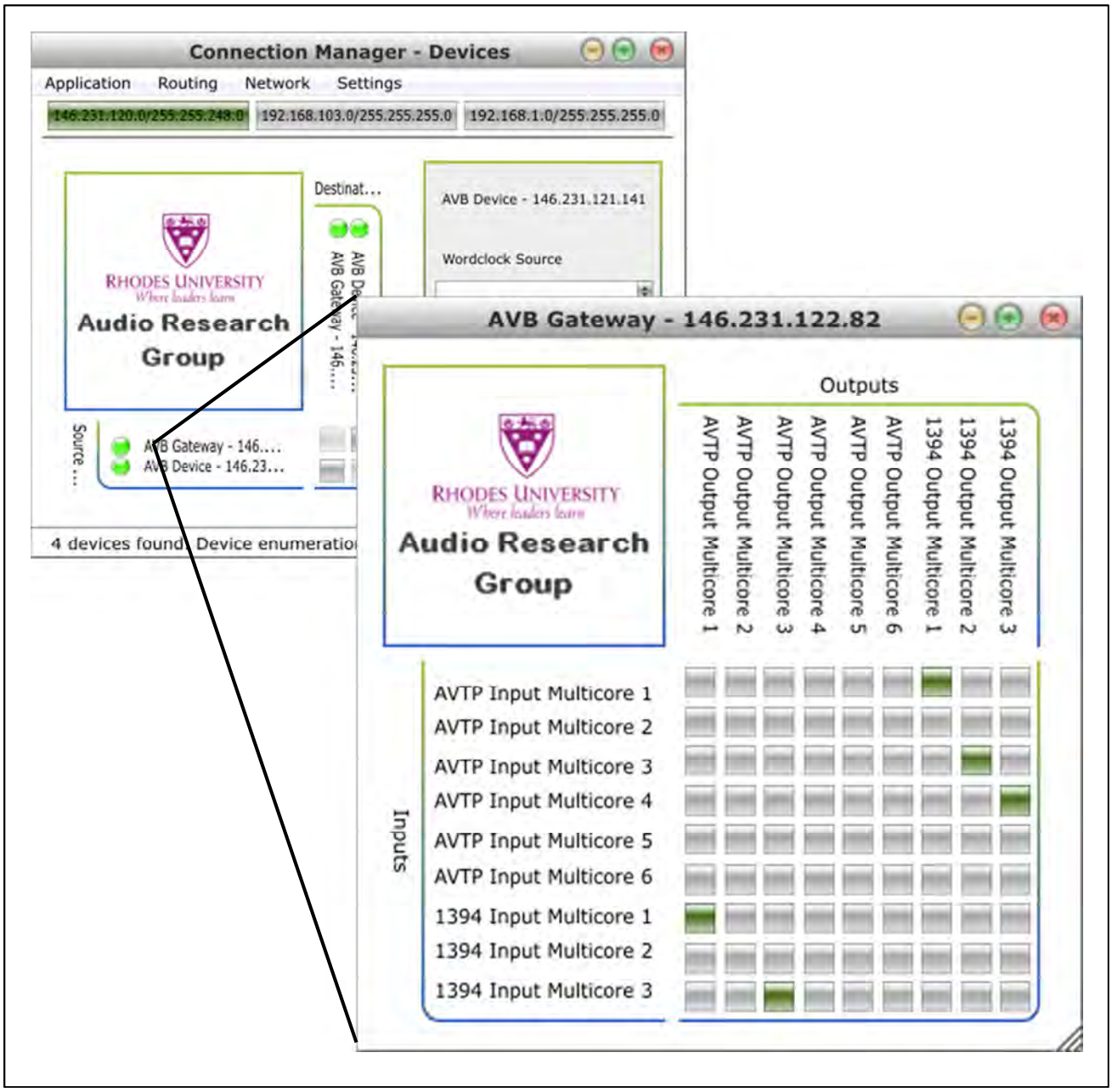

Figure 152: IEEE 1394/AVB audio gateway internal routing matrix

Via these routing matrices, it is possible to route audio signals entering the device via its inputs to the outputs of the device by selecting the cross points on the matrix. Selecting a cross point on a matrix causes an XFN *set value* message for the *crosspoint enable* parameter to be sent to the appropriate device. When this happens, the callback function for the parameter is called within the device. This callback function handles the device specific method of configuring the input to output routing.

11.6 Stream Establishment

Before any audio streaming is able to take place between devices on either an IEEE 1394 bus, or an Ethernet AVB network, each device that is part of a stream needs to be appropriately configured, and any network resources required for streaming need to be reserved. Devices that source audio onto these networks need to be configured to start streaming with certain parameters, and devices that receive streams on these networks need to be configured to receive specific audio streams. Each networking technology has its own unique methods of establishing and tearing down audio streams, as shown in Chapter 3 “Resource Reservation”.

11.6.1 IEEE 1394

On an IEEE 1394 bus, a transmitting device needs to be configured to transmit a particular audio stream on a particular isochronous channel, and it needs to be configured to start the transmission of the audio stream. Any devices that are required to receive an audio stream need to be configured with the isochronous channel number of the audio stream of interest. They also need to be configured to start listening for the particular audio stream. These parameters allow for streams to be established and torn down. For example, a connection management application may perform the following steps to establish a stream on an IEEE 1394 bus:

1. Read the appropriate register of a transmitting device to find out the isochronous channel number of the stream that is to be transmitted onto the bus.
2. Set the appropriate register in the receiving device to the isochronous channel number obtained from the transmitter of the stream.
3. Write to the appropriate register to instruct the transmitting device to start transmitting its stream.
4. Write to the appropriate register to instruct the receiving device to receive the stream being transmitted by the transmitter.

In order for the XFN protocol to establish stream connections across an IEEE 1394 bus, the XFN specification defines XFN address hierarchies to represent the parameters of IEEE 1394 multicores. These parameters allow for stream establishment to take place (see Section 11.6.3 “Multicore Representation”).

11.6.2 Ethernet AVB

On an Ethernet AVB network, each audio stream is uniquely identified via a 64-bit stream ID. Each stream may also be transmitted using an Ethernet multicast MAC address such that it may be received by multiple stream listeners. When one of the Ethernet AVB audio devices starts up, a unique stream ID is associated with each transmitting AVB stream and MAAP reserves a block of multicast MAC addresses for use by these streams.

Before streaming may take place, a device that is to receive a particular stream needs to know the stream ID of that stream. The device that is going to transmit the audio stream needs to advertise it to the AVB network. It does so by transmitting an Ethernet frame containing a *talker advertise* attribute to the network. This *talker advertise* attribute contains the stream's characteristics and requirements, and is propagated throughout the network.

The device that is to receive the stream sends an Ethernet frame containing a *listener ready* attribute to the talker to request attachment to the stream. This *listener ready* attribute contains the stream ID of the audio stream that it would like to receive. The *listener ready* attribute is forwarded through the network towards the talker device.

When a talker device receives a *listener ready* or a *listener ready failed* attribute that contains a stream ID that matches the stream ID of one of its advertised streams, it knows that there are sufficient resources available from itself to the listener(s) to support the stream. The talker is then able to start streaming.

In order for the XFN protocol to establish stream connections across an Ethernet AVB network, XFN level hierarchies were defined to represent the various parameters of Ethernet AVB streams to enable stream establishment to take place.

11.6.3 Multicore Representation

In terms of XFN, each stream (whether it is a stream across an IEEE 1394 network, or a stream across an Ethernet AVB network) is known as a *multicore*, as each stream may contain multiple channels of audio.

11.6.3.1 IEEE 1394

Figure 153 shows a section of the XFN address hierarchy that is built up to represent the IEEE 1394 multicores on the IEEE 1394 endpoint devices and on the IEEE 1394/Ethernet AVB audio gateway devices. An IEEE 1394 device may have a number of input and output IEEE 1394 multicore endpoints through which it may receive and transmit audio streams.

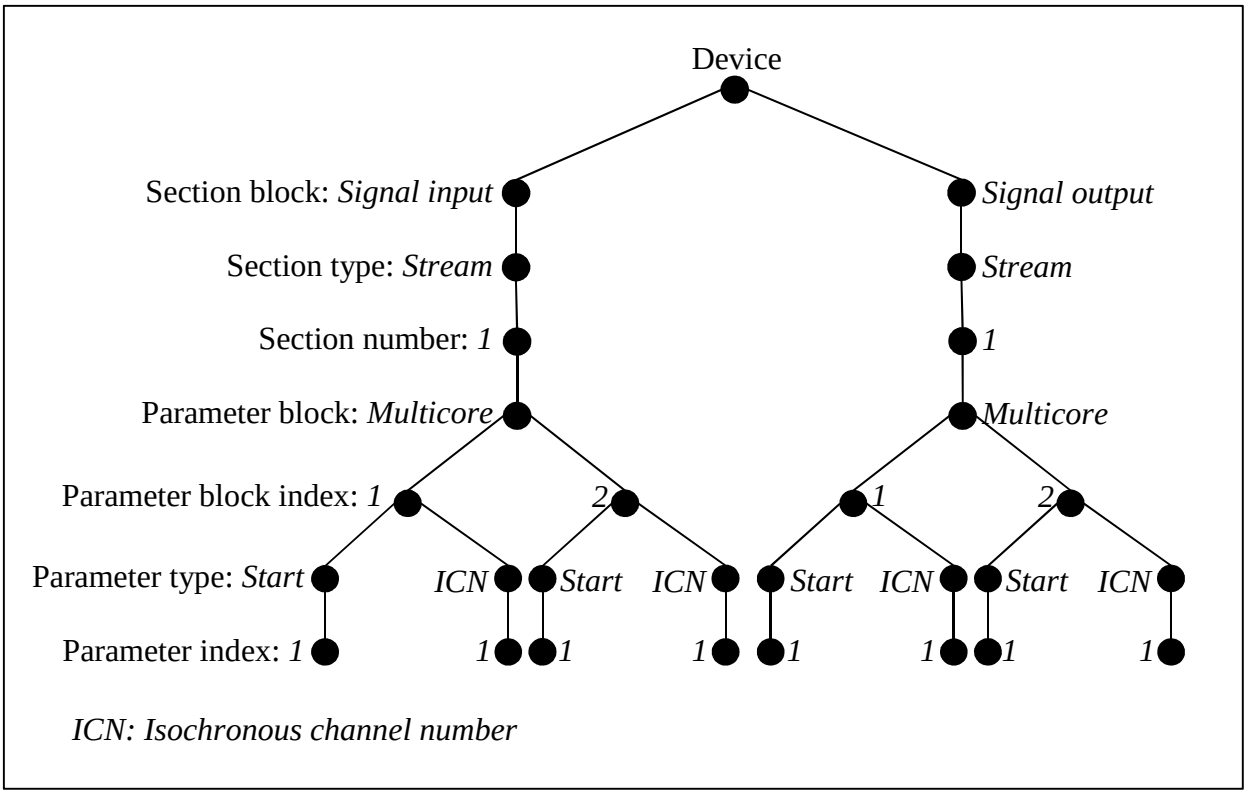

Figure 153: The XFN address hierarchy for IEEE 1394 multicores

The XFN address hierarchy for IEEE 1394 multicores is structured as follows:

- Input multicores are represented under the *signal input section block* as they are inputs to the system from external sources. Output multicores are represented under the *signal output section block* as they are outputs from the system to external destinations.
- As multicores are used for streaming, they are represented under the *stream section type* of the *signal input* and *signal output section blocks*.
- The *section number* is used to represent the interface that the multicores are associated with. Interfaces are numbered from one. An IEEE 1394 bridge, for example, has a number of IEEE 1394 interfaces, and these are uniquely identified under the *section number* level.

- Multicores and their parameters are represented under the *multicore parameter block*.
- Unique *parameter block index* values are used to uniquely identify multicores, as a number of input and output multicores may exist per interface. Multicores are indexed from one. Shown in Figure 153 are two input multicores and two output multicores.
- Under each *parameter block index* is listed the parameters that are associated with each multicore. Shown are an *isochronous channel number* parameter, and a Boolean *start* parameter. In the case of a transmitting multicore, the *isochronous channel number* parameter value specifies which isochronous channel the multicore transmits on and the Boolean *start* parameter starts and stops streaming. In the case of a receiving multicore, the *isochronous channel number* parameter specifies which isochronous channel the multicore will receive its audio stream on, and the Boolean *start* parameter starts and stops the reception of the audio stream.
- For each of the multicore parameters, the *parameter index* value is set to one as there exists only one of each of these parameters for each multicore.

Associated with each of these parameters is a callback function that allows the device to respond to requests from remote XFN devices. When the callback function is called for the *start* parameter with a *get value* command (as a result of the remote device sending a *get value* request for the parameter), the callback function obtains the streaming state of the requested input or output and sends the state back to the requester. The AVB Device component of an audio gateway device has the following two functions that allow for the retrieval of the streaming state of a 1394 multicore:

- `AVBDevice_is1394OutputStarted`: This function returns the streaming state of an output 1394 multicore. It determines whether the device is configured to transmit a particular stream.
- `AVBDevice_is1394InputStarted`: This function returns the streaming state of an input 1394 multicore. It determines whether the device is configured to receive the stream identified by the *isochronous channel number* parameter.

When the callback function associated with a *start* parameter is called indicating a *set value* command, the callback function sets the streaming state of the requested input or output. The AVB Device component has the following two functions that allow for the 1394 streaming of an audio gateway device to be started:

- `AVBDevice_start1394Output`: This function instructs the device to start the transmission of a particular source IEEE 1394 stream. Before the transmission goes ahead, bandwidth and an isochronous channel number are reserved on the IEEE 1394 bus for the stream.
- `AVBDevice_start1394Input`: This function instructs the device to start the reception of a particular isochronous stream. The isochronous stream is identified by the value of the isochronous channel number associated with the sink stream.

The AVB Device component has the following two functions that allow for the IEEE 1394 streaming of an audio gateway device to be stopped:

- `AVBDevice_stop1394Output`: This function instructs the device to stop the transmission of a particular IEEE 1394 stream.
- `AVBDevice_stop1394Input`: This function instructs the device to stop the reception of the stream identified by the value of the sink stream's isochronous channel number.

Similarly, if the callback function associated with an *isochronous channel number* parameter is called with a *get value* command, the AVBDevice component's

- `AVBDevice_get1394InputIsochChannelNumber` function is called to obtain the isochronous channel number associated with a particular IEEE 1394 input multicore.
- `AVBDevice_get1394OutputIsochChannelNumber` function is called to obtain the isochronous channel number associated with a particular IEEE 1394 output multicore.

Once the value of the multicore's isochronous channel number has been obtained, it is returned to the requesting device.

If the callback function associated with an *isochronous channel number* parameter is called with a *set value* command, the AVBDevice component's

- `AVBDevice_set1394InputIsochChannelNumber` function is called to set a particular IEEE 1394 input multicore's isochronous channel number.
- `AVBDevice_set1394OutputIsochChannelNumber` function is called to set a particular IEEE 1394 output multicore's isochronous channel number.

11.6.3.2 Ethernet AVB

Figure 154 shows the section of the XFN address hierarchy that is built up to represent the parameters associated with the AVB multicores on the Ethernet AVB endpoint devices and the IEEE 1394/Ethernet AVB audio gateway devices. The address hierarchy is similar to the address hierarchy for IEEE 1394 multicores, except that the multicores have different parameters associated with them. The figure shows the address hierarchy used to address *stream ID* (SID) parameters, as well as Boolean *advertise* and Boolean *listen* parameters. The *stream ID* parameter that is associated with a transmitting multicore is used to represent the stream ID that uniquely represents that stream. The *stream ID* parameter that is associated with a receiving multicore is used to represent the stream ID of the stream that it is interested in receiving. The transmitting multicore's Boolean *advertise* parameter is used to allow a stream to be advertised, or for that advertisement to be withdrawn. The Boolean *listen* parameter that is associated with a receiving multicore is used to allow a device to request reception of a stream, or to withdraw that request.

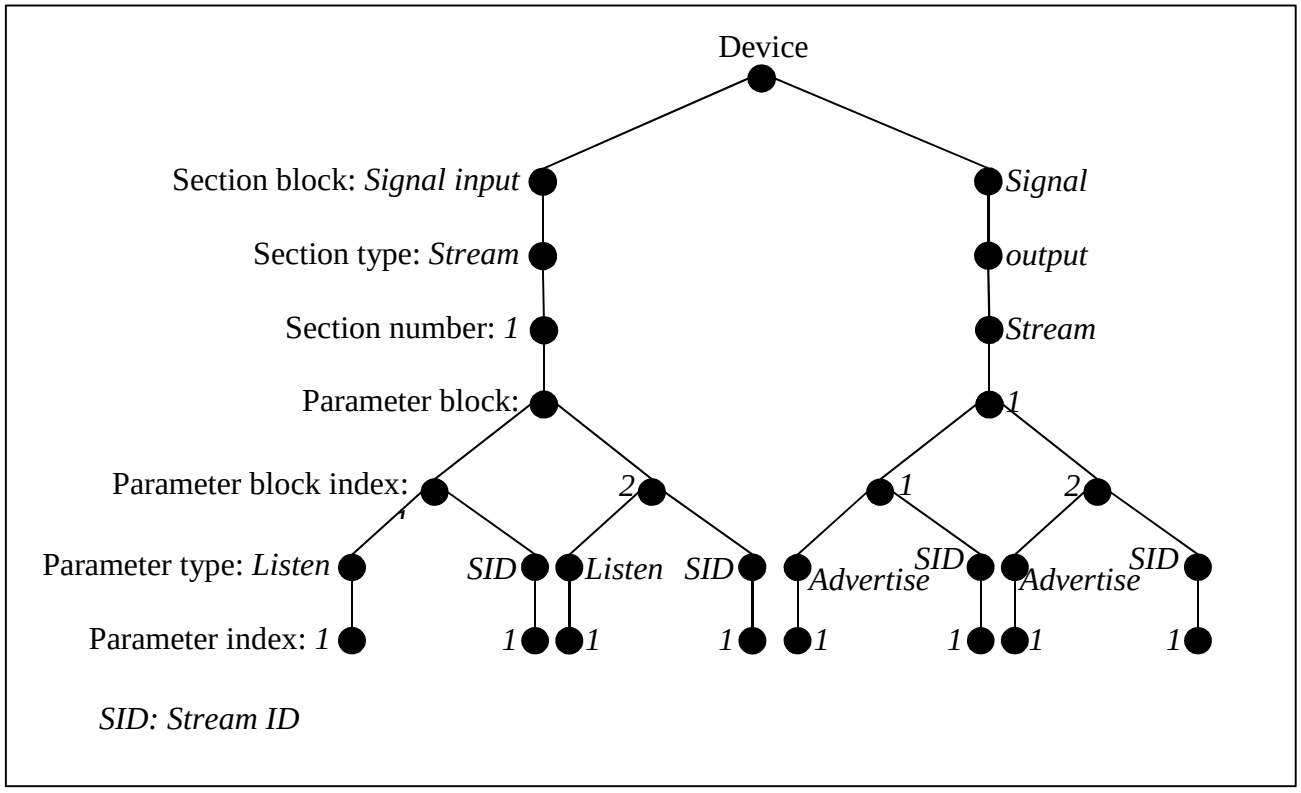

Figure 154: The XFN address hierarchy for AVB multicores

Each one of these parameters is associated with a callback function that allows for acquiring and setting these parameters values. Performing a *get value* request on the *Stream ID* parameter results in

the value of the stream ID of the stream being returned to the caller. Performing a *set value* request on an AVB input causes its stream ID to be set. It is not possible to set the value of the stream ID associated with an AVB output as these are allocated when the system is initialised and remain as such for the lifetime of the system.

The *advertise* and *listen* parameter callbacks respond as follows:

- Performing a *set value* request on an *advertise* parameter with a value of true causes the device's `AVBDevice_registerAVTPStream` function to be called for the addressed AVB output stream. This causes the particular stream to be advertised to the attached AVB network.
- Performing a *set value* request on an *advertise* parameter with a value of false causes the device's `AVBDevice_deregisterAVTPStream` function to be called for the addressed AVB output stream. This causes the particular stream's advertisement to be withdrawn from the AVB network.
- Performing a *set value* request on a *listen* parameter with a value of true causes the device's `AVBDevice_registerAVTPStreamAttach` function to be called for the addressed AVB input stream. This causes a *listener ready* attribute to be declared for the stream (as identified by the value of the input's *stream ID* parameter).
- Performing a *set value* request on the *listen* parameter with a value of false causes the device's `AVBDevice_deregisterAVTPStreamAttach` function to be called for the addressed AVB input stream. This causes the particular stream's *listener* attribute to be withdrawn.

11.6.3.3 Distinguishing Between Multicore Types

With the level hierarchies shown in Figure 153 and Figure 154, it is possible to determine that multicores exist on a device, but it is not possible to determine the type of multicore being dealt with. For each multicore (i.e., for each *parameter block index* of the *multicore parameter block*), there exists a *multicore type* at the *parameter type* level. Unique values have been defined to represent an IEEE 1394 multicore and an Ethernet AVB multicore. Performing an XFN *get value* on this parameter returns the type of multicore being represented.

11.6.4 External Device Multicore Routing

Each XFN device that is discovered by the Connection Manager is enumerated to discover the multicores that exist on each of the devices. To determine the number of input multicores that exist on a device, the Connection Manager queries the device to determine the number of child nodes that the *multicore parameter block* of the *signal input section block* has. To determine the number of output multicores that exist on a device, the Connection Manager queries the device to determine the number of child nodes that the *multicore parameter block* of the *signal output section block* has. Once these have been obtained, the Connection Manager is able to query the parameters of each multicore within a *multicore parameter block* (each identified with a unique *parameter block index* value).

For each multicore that is discovered on a device, its type and name is retrieved and stored by the Connection Manager. The XFN Stack component of the Connection Manager defines the `getInputMulticoreType` function (shown in Listing 46) that allows for the retrieval of a specific input multicore type (represented by a `MulticoreType` enum (shown in Listing 47)). The `interfaceNumber` argument should be supplied with the interface number that the multicore is associated with. The `multicoreIndex` is the index of the particular input multicore whose type is being requested. Similarly, the XFN Stack component defines the `getOutputMulticoreType` function that allow for the retrieval of an output multicore's type. Currently, the `MulticoreType` enumeration defines an IEEE 1394 and AVB multicore type.

```
MulticoreType getInputMulticoreType (
     const uint32 deviceIPAddress,
     const int interfaceNumber,
     const int multicoreIndex);
```

Listing 46: `getInputMulticoreType` function

```
enum MulticoreType
{
     INVALID_MULTICORE = 0,
     FW_MULTICORE,
     AVB_MULTICORE
};
```

Listing 47: `MulticoreType` enum

If the multicore is an IEEE 1394 multicore, the values of the *isochronous channel number* and *start* parameters are retrieved by the Connection Manager. If the output multicore is an AVB multicore, the value of the *stream ID* and *advertise* parameters are retrieved. If an input multicore is an AVB multicore, then the value of the *stream ID* and the *listen* parameters are retrieved. All of these values are stored by the Connection Manager.

The XFN Stack component defines functions that allow for the retrieval of this information:

- `getInputMulticoreIsochChannelNumber`: retrieves the isochronous channel number associated with a particular input on a particular device.
- `getOutputMulticoreIsochChannelNumber`: retrieves the isochronous channel number associated with a particular output on a particular device.
- `setInputMulticoreIsochChannelNumber`: sets the isochronous channel number of a specific input on a particular device.
- `setOutputMulticoreIsochChannelNumber`: sets the isochronous channel number of a specific output on a particular device.

Similarly, the XFN Stack component provides functions that allow for the values of AVB multicore *stream ID* parameters to be obtained and set, as well as the states of the *advertise* and *listen* parameters:

- `getInputAVBMulticoreStreamID`: retrieves the stream ID associated with a particular input AVB multicore on a particular device.
- `getOutputAVBMulticoreStreamID`: retrieves the stream ID associated with a particular output AVB multicore on a particular device.

- `setInputAVBMulticoreStreamID`: sets the value of the stream ID associated with a particular input AVB multicore on a particular device.
- `setOutputAVBMulticoreAdvertise`: sets the state (either true of false) of the *advertise* parameter associated with a particular AVB output multicore on a particular device.
- `setInputAVBMulticoreListen`: sets the state (either true or false) of the *listen* parameter associated with a particular AVB input multicore on a particular device.

With the Connection Manager, multicores are displayed by selecting one of the cross points on the matrix of the main device display of the application. This displays a second matrix that is used to represent the transmitting multicore(s) of the selected source device, and the receiving multicore(s) of the selected destination device. Figure 155 shows an example of the multicores display for AVB multicores. The source multicores are represented with labels along the left hand side of the matrix. In the figure, these are the output AVB multicores of an IEEE 1394/Ethernet AVB audio gateway device. Destination multicores are represented with labels along the top of the matrix. In the figure, these are the input AVB multicores of an Ethernet AVB endpoint device. The buttons on the matrix show whether or not a stream connection exists between a transmitting multicore of one device, and a receiving multicore of another device.

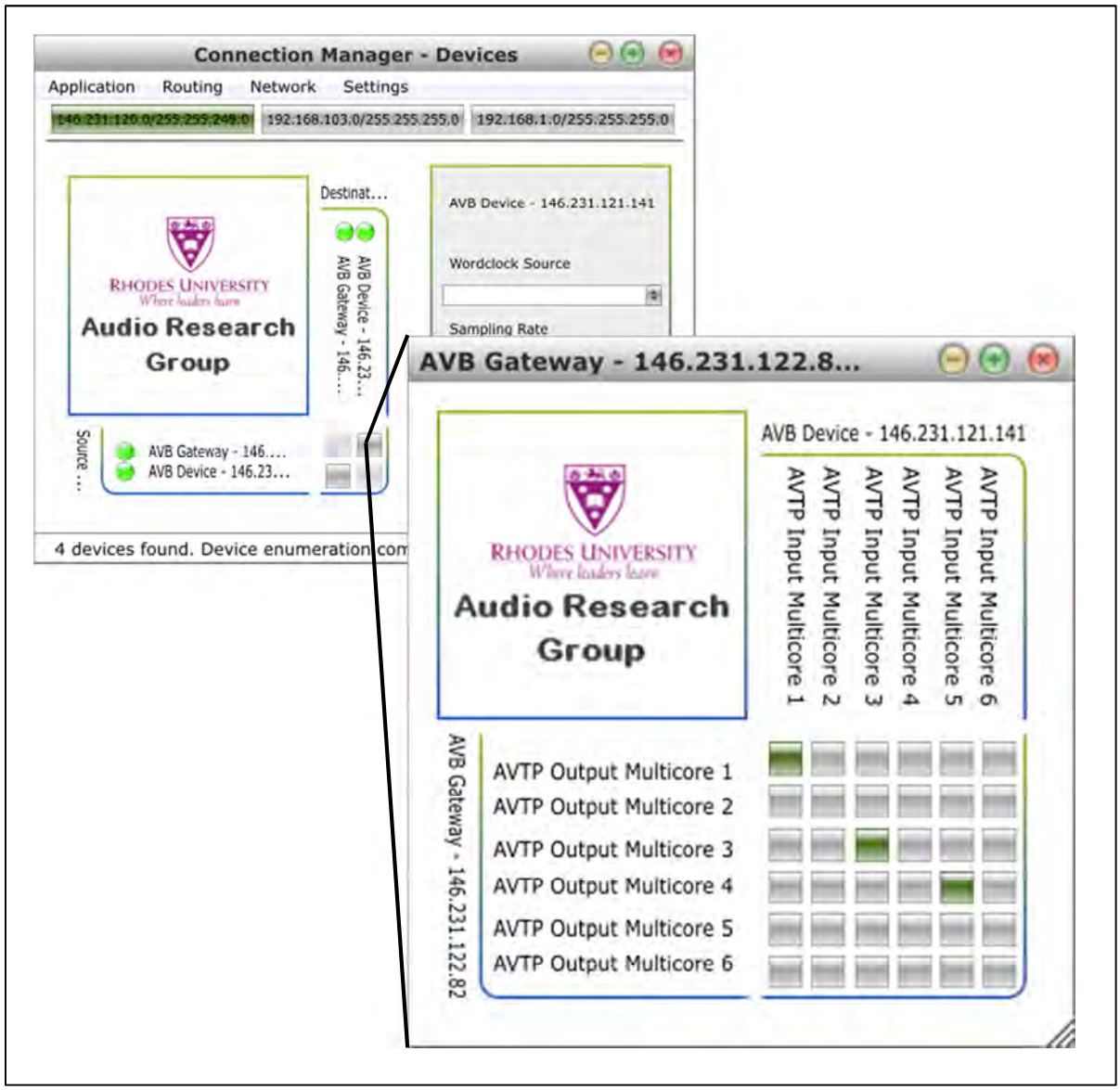

Figure 155: Ethernet AVB multicore patching

Figure 156 shows examples of the multicore displays for IEEE 1394 multicores. As with the multicores display for the AVB multicores, source multicores are displayed along left hand side of the matrix, and destination multicores are displayed along to the top of the matrix. The patching matrix shown on the right hand side of the figure is displayed when the cross point between the AVB Gateway and the UMAN Eval Board is selected. Shown are the transmitting IEEE 1394 multicores of the IEEE 1394/Ethernet AVB audio gateway device, and the receiving IEEE 1394 multicores of an IEEE 1394 end point device. The selected cross points on the patching matrix show that the first transmitting multicore of the audio gateway device is patched through to the first receiving multicore of the IEEE 1394 endpoint device, and the second transmitting multicore of the audio gateway is patched through to the second receiving multicore of the IEEE 1394 endpoint device. The patching

matrix shown at the bottom of the figure is displayed when the cross point between the IEEE 1394 endpoint device and the audio gateway device is selected. This patching matrix shows that the first transmitting multicore of the IEEE 1394 endpoint device is patched through to the first reception multicore of the audio gateway device, and the second transmitting multicore of the of the IEEE 1394 endpoint is patched through to the third input multicore of the audio gateway device.

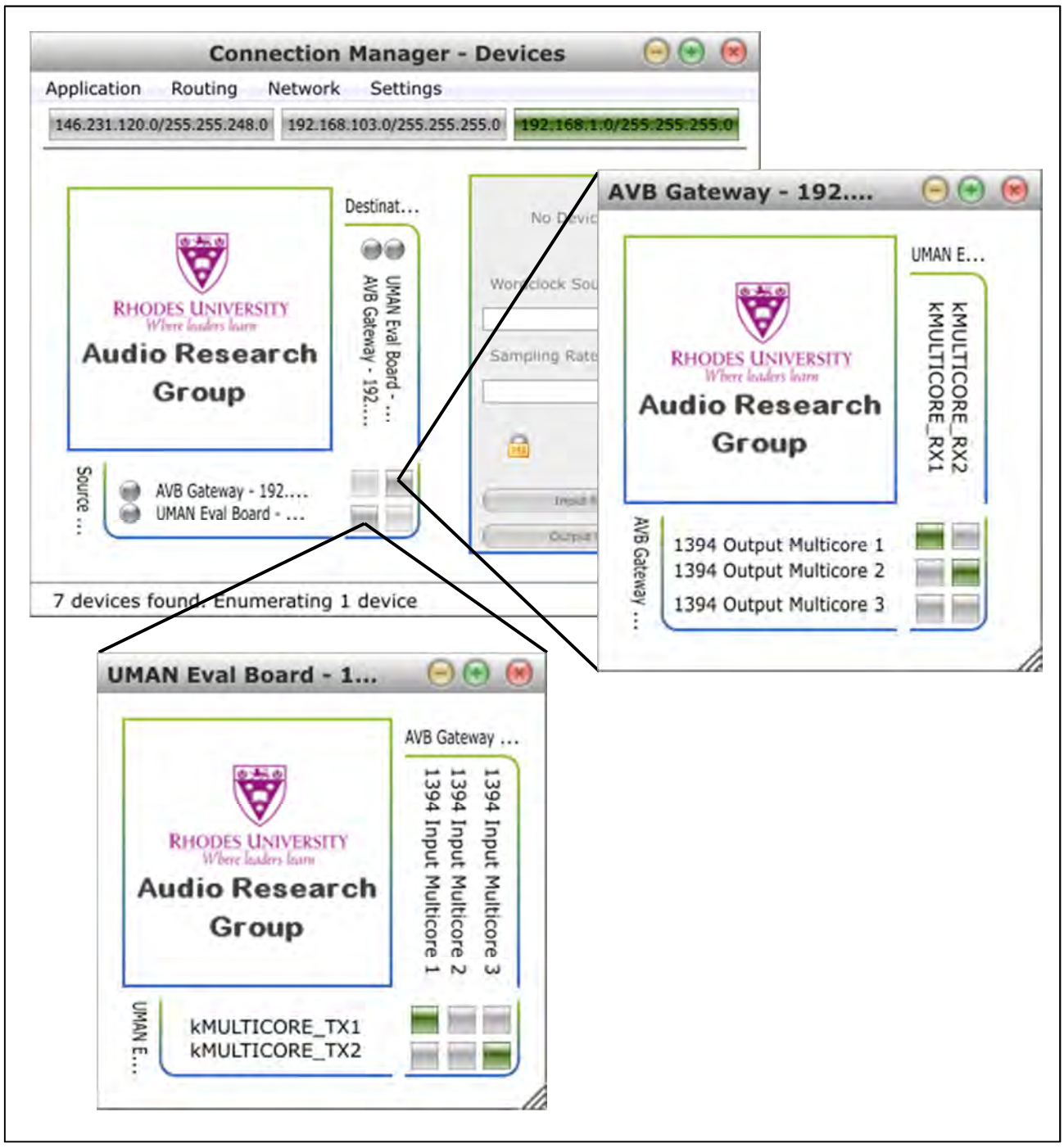

Figure 156: IEEE 1394 multicore patching

Making a connection between the multicores involves selecting the cross points between the required multicores. When a cross point is selected, a sequence of events is initiated that ensures that a connection between the multicores on the devices is established. This sequence of events is dependent on the specific type of multicore being represented.

When the Connection Manager wishes to establish a multicore connection between two XFN devices on an IEEE 1394 bus, it performs the following sequence of events:

1. It issues an XFN *get value* request to the transmitting device to get the value of the isochronous channel number parameter of the multicore that is to transmit the stream.
2. It issues an XFN *set value* request to the receiving device to set the channel number of multicore that is to receive the stream.
3. It issues XFN *set value* requests to the devices to set their *start* parameters to initiate streaming.

The XFN Stack component provides a function that allows a stream connection to be established between two IEEE 1394 devices. The signature of this function is shown in Listing 48. The caller specifies the IP addresses of the devices involved in the connection, the interface numbers of the interfaces associated with the multicores, the multicores indexes, and the isochronous channel number to use. The `connect` argument is used by a caller to specify whether a connection is being created or broken.

```
const bool connectOutputFWMulticoreSocketToInputFWMulticoreSocket
(
    const uint32 senderIPAddress,
    const int senderInterfaceNumber,
    const int senderMulticoreIndex,
    const uint32 receiverIPAaddress,
    const int receiverInterfaceNumber,
    const int receiverMulticoreIndex,
    const int isochChannelNumber,
    const bool connect);
```

Listing 48: `connectOutputFWMulticoreSocketToInputFWMulticoreSocket` function

When the Connection Manager wishes to establish a connection between two XFN devices on an Ethernet AVB network, it performs the following sequence of events:

1. It issues an XFN *get value* request to the transmitting device to get the value of the *stream ID* parameter of the multicore that is to transmit the stream.
2. It issues an XFN *set value* request to the receiving device to set the *stream ID* parameter of multicore that is to receive the stream.
3. It issues an XFN *set value* request to the transmitting device to set the *advertise* parameter of the multicore that is to transmit the stream.
4. It issues an XFN *set value* request to the receiving device to set the *listen* parameter of the multicore that is to receive the stream.

The XFN Stack component provides a function that allows a stream connection to be established between two AVB devices. The signature of this function is shown in Listing 49. The caller specifies the IP addresses of the devices involved in the connection, the interface numbers of the interfaces associated with the multicores, the multicores indexes, and the stream ID to use. The `connect` argument is used by a caller to specify whether a connection is being created or broken.

```
const bool
connectOutputAVBMulticoreSocketToInputAVBMulticoreSocket (
    const uint32 senderIPAddress,
    const int senderInterfaceNumber,
    const int senderMulticoreIndex,
    const uint32 receiverIPAaddress,
    const int receiverInterfaceNumber,
    const int receiverMulticoreIndex,
    const uint8 streamID [8],
    const bool connect);
```

Listing 49: `connectOutputAVBMulticoreSocketToInputAVBMulticoreSocket` function

11.7 Rationale for Matrix Patching

As discussed in [97], digital audio networks often make use of a software patchbay to configure audio routing between audio devices on the network. These patchbays display the devices on the network, along with their associated inputs and outputs, and allow sound engineers to make and break connections between the inputs and outputs of the devices. There are a range of software patchbay types that have been developed, including list-based patchbays, tree-view-based patchbays, tree-grid-based patchbays and graphic-based patchbays. This section provides a comparison of these patchbays in order to motivate for the grid approach to connection management.

11.7.1 List-Based Patchbays

List-based patchbays display the devices on an audio network along with their inputs and outputs in the form of lists. Sound engineers are able to select the output and input from the lists and make connections between the two. Figure 157 shows the mLAN Graphic Patchbay's List View [12]. This window shows mLAN[3] [98] devices present on an IEEE 1394 network, their various audio and MIDI inputs and outputs, and the connection settings of the devices. The annotations in the figure are explained below:

1. This list displays the various mLAN devices on the IEEE 1394 network, and the outputs present on those devices. These are the various signal sources present on the network.
2. This list displays the various mLAN devices on the IEEE 1394 network along with the inputs present on those devices. These are the various signal destination points that exist on the network.
3. This list displays the connections between the outputs and inputs of the mLAN devices on the IEEE 1394 network.

[3] Music Local Area Network (mLAN) is a technology that allows for the transmission of deterministic, synchronised, low latency audio data over a network. mLAN makes use of IEEE 1394 in order to achieve its goals.

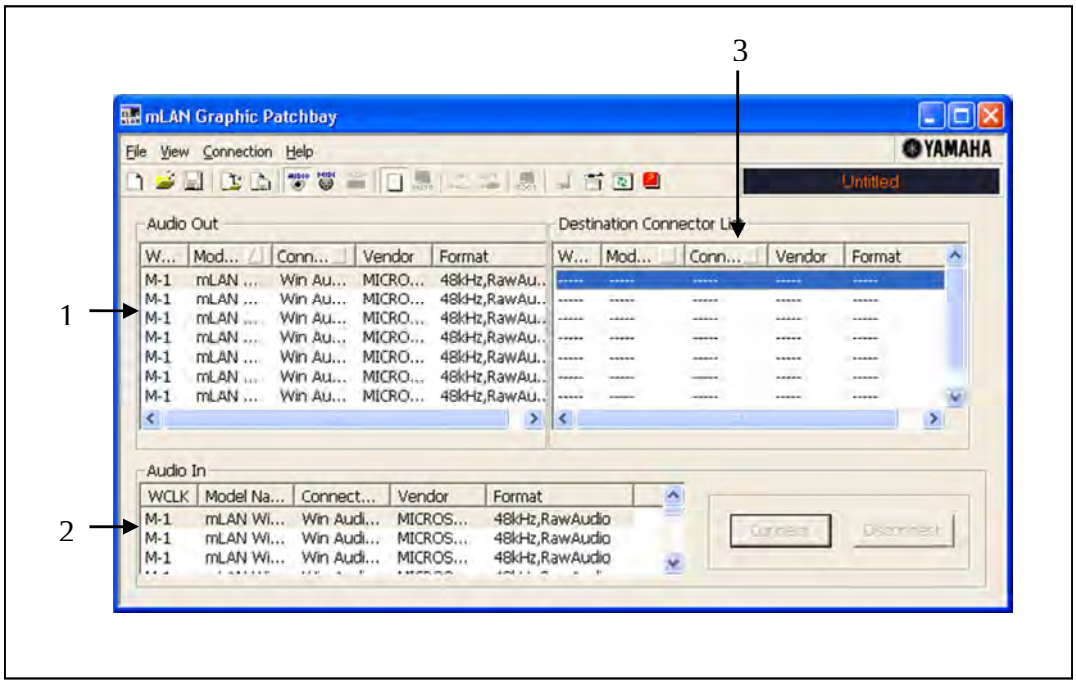

Figure 157: The Yamaha mLAN Graphical Patchbay's List View

Via this list-based patchbay, making connections between the various outputs and inputs of the mLAN devices is done by scrolling to the required signal output and selecting it (see annotation 1 in the above figure), scrolling to the required signal input and selecting it (see annotation 2 in the above figure) and then selecting the Connect button. The output and input of connected devices will be shown as having connections between them by displaying the destination device in the Destination Connector List (see annotation 3 in the above figure) next to the output.

11.7.2 Tree-View-Based Patchbays

Tree-view-based patchbays present the layout of the network in the form of tree structures. Figure 158 shows the NAS Explorer Patchbay [99] used to control mLAN devices. The tree views of this patchbay are organized in such a way as to represent the hierarchical nature of the IEEE 1394 network it is representing. This patchbay contains two tree-view structures: one to represent the signal outputs available on the mLAN devices, and one to represent the signal inputs available on the mLAN devices.

The root node of each tree represents the entire IEEE 1394 network. Its child nodes represent the various IEEE 1394 buses that make up the network. Each node representing the various buses of the

network has child nodes representing the mLAN devices present on the bus. Each node in the tree representing the devices on the buses has child nodes representing either the inputs, or the outputs of the devices, depending on the specific tree. This way, sound engineers can logically locate the require inputs and outputs by navigating their way down the tree structure.

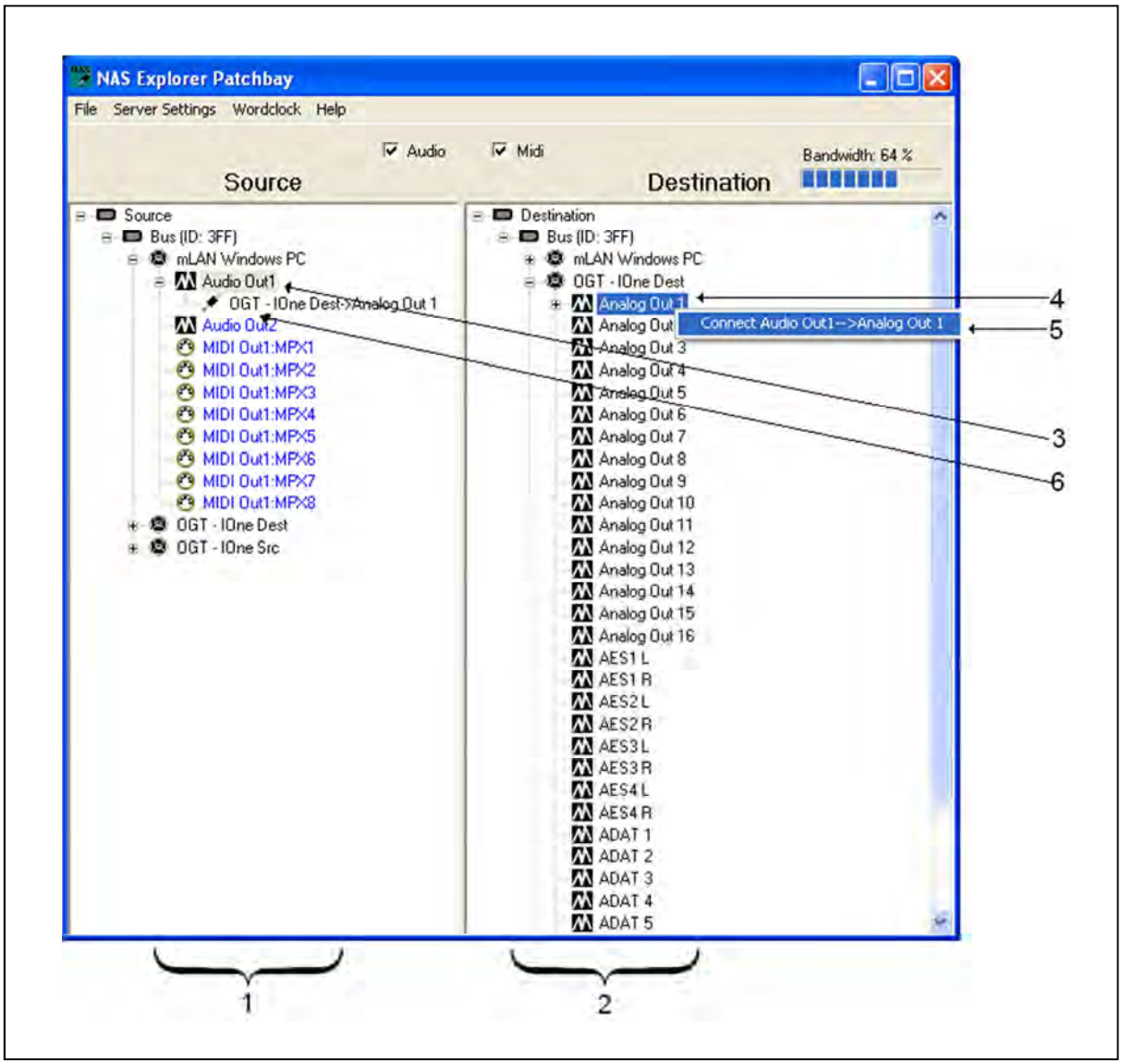

Figure 158: The NAS Explorer Patchbay

The annotations in Figure 158 are explained below:

1. This tree-view represents the structure of the IEEE 1394 network along with the mLAN devices and their signal output plugs, hierarchically.
2. This tree-view represents the structure of the IEEE 1394 network along with the mLAN devices and their signal input plugs, hierarchically.

3. Making a connection between a signal output and a signal input requires that a sound engineer first select the signal output.
4. The second stage for a connection requires selecting the signal input.
5. Once the output and input have been selected, a connection between the two plugs is made by right clicking on one of the selected inputs or outputs, and selecting the Connect menu item.
6. The NAS Explorer Patchbay displays a connection by making the connected output a child of the input, and vice versa. Breaking the connection is performed by right-clicking this child node, and selecting the Disconnect menu item.

11.7.3 Tree-Grid-Based Patchbays

Tree-grid-based patchbays display the devices on a network, along with their associated inputs and outputs, along the axes of grids. The cross points on the grids allow for connections between the outputs and inputs of devices to be made or broken by selecting or deselecting the points on the grids, respectively. Figure 159 shows the Routing Matrix of the Otari ND 20B mLAN Control Software [100]. This tree-grid-based patchbay displays Otari ND 20B units on an mLAN network, their associated inputs and outputs and the connections between them. The Otari ND 20B units, along with their inputs and outputs, are shown hierarchically on the axes of the grid with tree views. Making and breaking connections is performed by selecting and deselecting the cross points on the grid. The annotations in this figure are explained below:

1. The left hand column shows the names of the devices present on the network and the input channels available on each device, hierarchically.
2. The top row shows the names of the devices on the network and the output channels associated with each device, hierarchically.
3. The checkered section of the grid shows which output channels are routed through to which input channels. Making and breaking connections is performed by selecting and de-selecting the cross points on the grid where the required output channels intersect the required input channels.

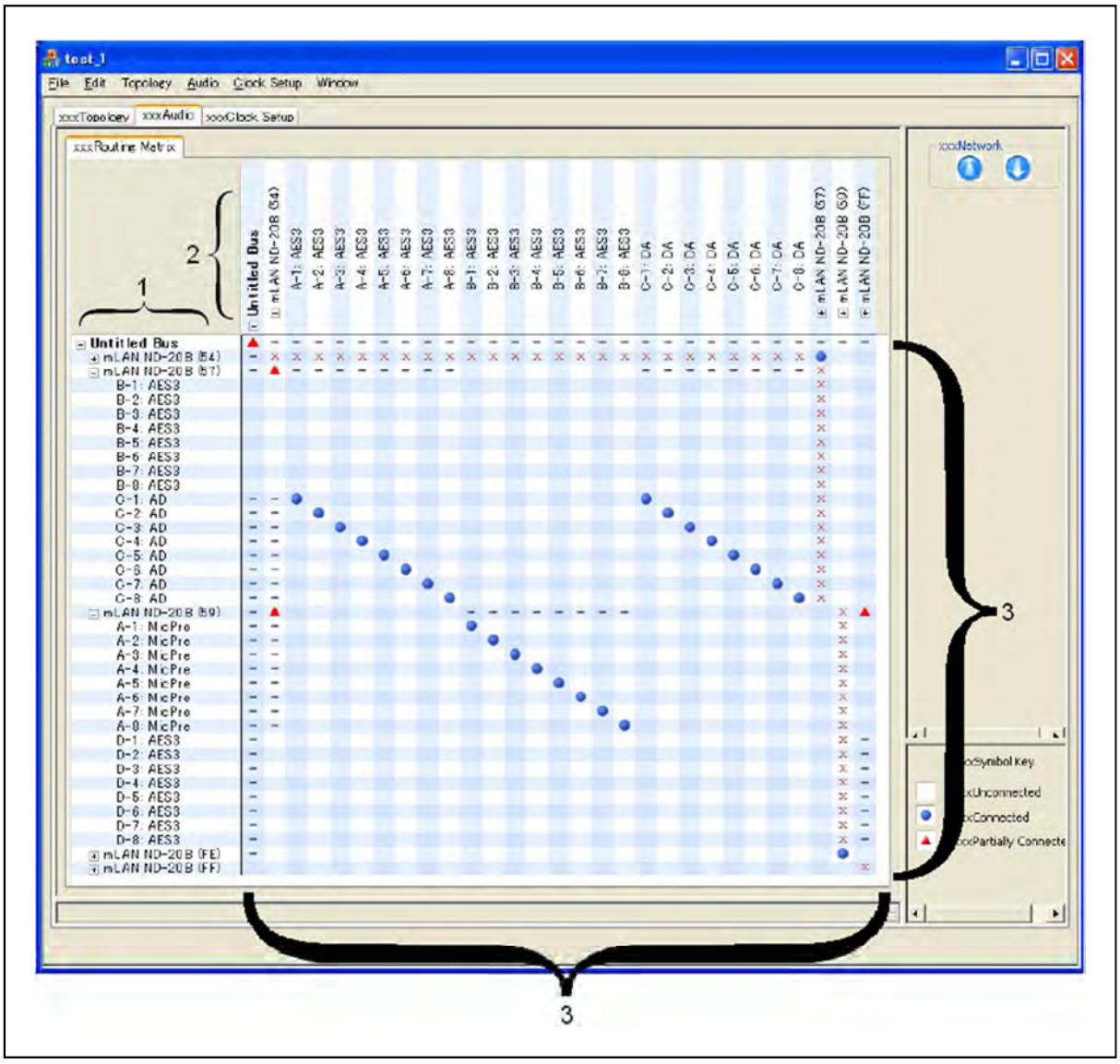

Figure 159: The Otari ND 20B mLAN Control Software Routing Matrix

11.7.4 Graphic-Based Patchbays

A graphic-based patchbay represents devices on an audio network with the aid of icons. Each icon represents a device and the inputs and outputs associated with it. The connections between the devices are represented with cable-like lines drawn between the plugs. The Yamaha Graphic Patchbay [12] is an example of a graphic-based patchbay. This patchbay is used to make connections between mLAN devices on an IEEE 1394 network. The primary window of this patchbay is shown in Figure 160. The annotations in this figure are explained below:

1. The mLAN devices on the IEEE 1394 network are shown on the workspace with the aid of different coloured icons, and each icon may contain detail pertaining to the particular device.

2. The inputs and outputs on each device are shown on the side of the icons that represent the different mLAN devices.
3. The connections between the devices are shown with graphic cable-like connectors. Different colours are used to distinguish the different virtual cables.
4. In order to make a connection between an output and input on two different devices, the ‘out’ section of one device, and the ‘in’ section of another device is selected. This displays two Connector windows that display the relevant plugs available on the selected devices. A connection is then made by selecting the required output, and then selecting the required input. Connections are then visually shown with graphical cables between the output plugs, and the input plugs.

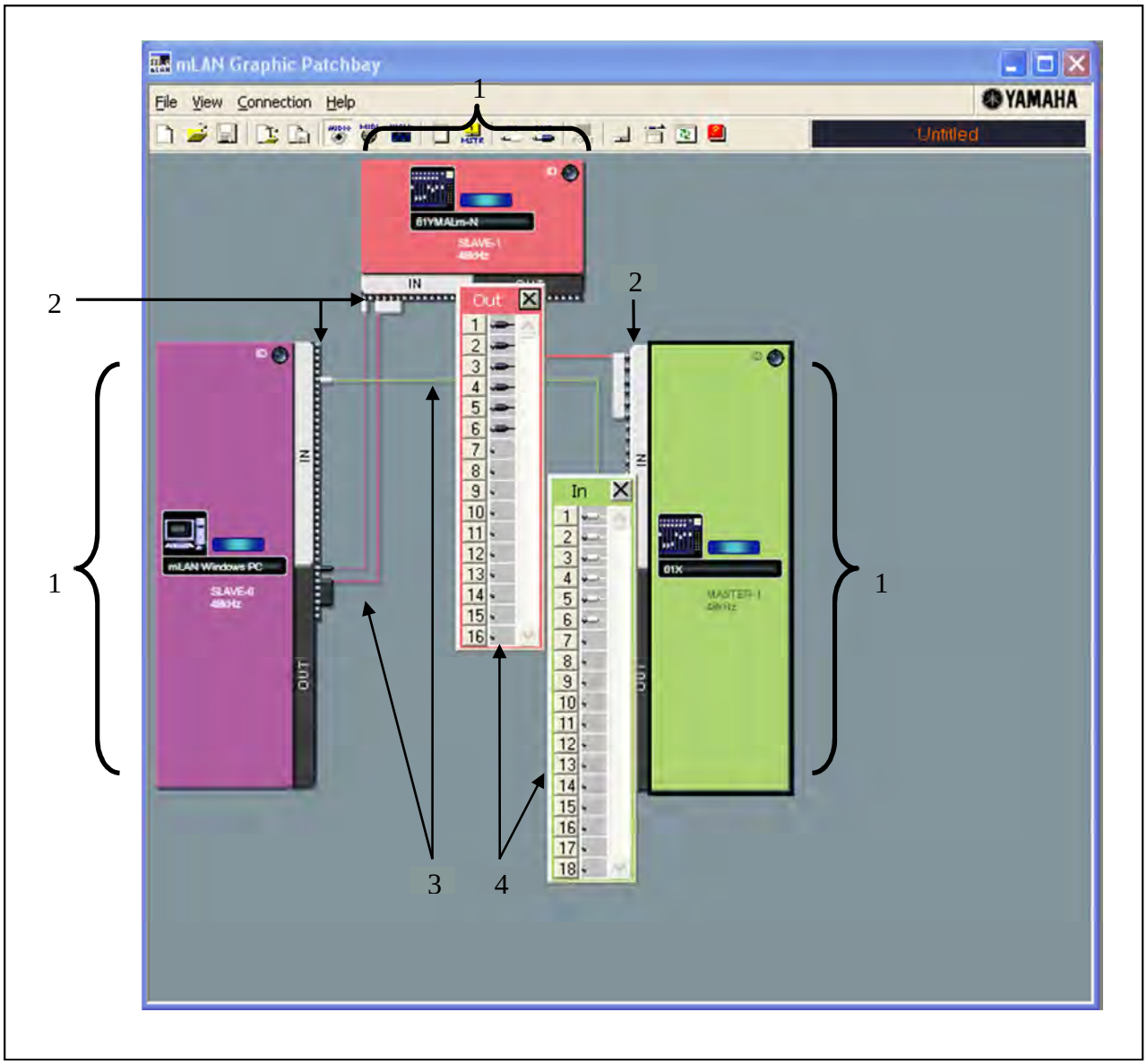

Figure 160: The Yamaha mLAN Graphic Patchbay

11.7.5 A Comparison Of Patchbays

As the number of devices on an audio network increases, the complexity of the patchbay representing the devices on the network may increase as well. Besides the list-based patchbay discussed above, all of the patchbays represent the network hierarchically. This hierarchical representation reflects the structure of the network itself. This makes navigating to required devices and their inputs and outputs take place in a logical way. It also reduces the amount of clutter on the displays of the patchbays, as sound engineers have the option of only displaying the information that they are interested in.

Table 44 shows the number of mouse button clicks it would take a sound engineer to make a connection between two devices on a single bus IEEE 1394 network, using the patchbays discussed above. The table distinguishes between the number of mouse button clicks it takes to navigate to the required inputs and outputs to make the connection.

	List-based	Tree-view-based	Tree-grid-based	Graphic-based
Navigate to plugs	0 (Scroll)	6	4	2
Make connection	3	4	1	2

Table 44: The number of mouse clicks to make a soft connection

With all the patchbays discussed above, a sound engineer is required to individually navigate to the required output and input to make a connection.

The patchbays discussed above, except the tree-grid-based patchbay, require that sound engineers individually select the output and input before a connection can be made. Even though the list-based patchbay has the least number of mouse clicks to make a connection, the process of selecting the plugs could become tedious. The lists in which the inputs and outputs are listed may grow to the point where each list contains literally hundreds of individual items even for a small studio.

When making a connection with the tree-grid-based patchbay, there is no need to explicitly select the outputs and inputs required for a connection. A connection is created by selecting the cross point on the grid that is used to represent the output and input. The output and input are implicitly selected.

11.8 Connection Manager Grid Displays

The grid approach (as opposed to the approaches discussed above) allows for the implicit selection of source and destination devices, and the implicit selection of a signal source point and a signal destination point when patching. The grid based approach implemented for the Connection Manager mimics the process that an audio engineer would follow when manually routing audio between devices that are part of an audio system. When patching audio signals manually, audio devices may be connected by cables that are grouped into multicore cables. A multicore cable can be routed to

another location where the individual cables within the multicore can be sent to the sockets of a patch panel. Here individual signals can be patched through to a device, or be patched into another multicore cable. The Connection Manager mimics this process (refer to Figure 145 on page 299 for the sample network that the following examples apply to):

- First, connections are made within a source device. If, for example, an audio engineer would like to patch the signal from a microphone through to one of the sources device's output multicore sequences, then, with the Connection Manager, the source device is selected to display the internal routing configuration of the device. A connection can then be made between the input (that the microphone is connected to) and output (representing the sequence of the output multicore) of the device by selecting the cross point between these two on the patching matrix. An example of this can be seen in Figure 150 on page 308.
- Next, connections between two devices can be created by selecting the cross point between the signal source device and the signal destination device on a device display grid (as seen in Figure 156 on page 322). This displays the multicore outputs of the implicitly selected source device, and the multicore inputs of the implicitly selected destination device in the form of a matrix. Multicore connections between the devices are created by selecting the cross point between the signal source and the signal destination. In the example, the microphone's signal is routed into a particular multicore. This multicore is then patched through to a multicore input on the receiving device (the IEEE 1394 bridge, for example) by selecting the cross point between the two devices.
- Once inter-device patching is complete, connections within the destination device can be made (the IEEE 1394 bridge). The destination device is selected to display the internal routing configuration of the device. Connections on this device can be made between its inputs and outputs by selecting the cross points on a patching matrix. In the example, the incoming multicore signal that contains the microphone signal can be patched through to an output multicore on the other portal of the bridge. This process continues until the audio signal arrives at the destination device.

11.9 Conclusion

The IEEE 1394 and Ethernet AVB devices that were used as part of this study each contain an XFN stack. This chapter demonstrated how the XFN protocol has been used to represent and control both IEEE 1394 and Ethernet AVB audio devices:

- Each device is able to build up a seven-level XFN address hierarchy that reflects the hierarchical structure of the device itself.
- Via the protocol, remote devices are able to determine the presence of these devices. A graphical patchbay application was developed to discover and represent these devices.
- The protocol was used to provide a common view of signal source points, signal destination points, and signal patch points within the devices. The graphical patchbay application has been used to represent these signal source, destination, and patch points for both the IEEE 1394 audio devices, and Ethernet AVB audio devices. The patchbay application allows a user to create and break these patches.
- The protocol has been used to provide a common multicore view of the streams that may exist between IEEE 1394 devices, and streams that may exist between Ethernet AVB devices. Parameters are defined for each multicore type to enable stream connections across their respective networks. The graphical patchbay application is able to represent these multicores, and allows for patches to be created and destroyed between multicores of the same type.

Chapter 12 Conclusion

Digital networks are increasingly being used for the simultaneous transport of a variety of types of non-realtime and realtime data. The transmission of these various types of data places different demands on the networks that they are transmitted on. Non-realtime data can be delivered on a best effort basis, whereas realtime stream data requires the ability to:

- Be able to transfer data uninterrupted at a constant rate. The data needs to be delivered deterministically with bounded latency.
- Synchronise multiple realtime streams to allow for their simultaneous playback, and to allow for wordclock regeneration.

These requirements necessitate that the underlying network technology provides:

- A means to ensure that network resources (e.g., bandwidth) are reserved for realtime data. There need to be mechanisms in place allowing end stations involved in realtime data streaming to communicate their resource requirements to the network. This allows the network to reserve these resources for the streams.
- Mechanisms to guarantee that the data is transmitted in a deterministic and timely fashion across the network.
- A common sense of time amongst nodes on the network to allow for the synchronisation of realtime streams. The network needs to provide the ability to allow all stations on the network to have a common understanding of time.

The two dominant open standards-based networking technologies that natively provide the above mentioned capabilities are IEEE 1394 and Ethernet AVB. These networking technologies implement diverse mechanisms in order to achieve the above requirements. Audio transport protocols have been standardised for each of these networking technologies. These protocols allow audio data to be streamed between the audio devices that reside on these networks. Also defined are diverse incompatible command and control protocols that allow connection management to take place in and between the devices that reside on these networks.

12.1 Goals

The goals of this work were to:

- Demonstrate the ability of audio devices residing on IEEE 1394 networks to transmit audio data to audio devices residing on Ethernet AVB networks, and vice versa.
- Develop a common method of representing and controlling audio devices that reside on these disparate networks.
- Develop a graphical patchbay application that is able to graphically represent and control these diverse devices in a congruent manner.

12.2 Conclusions

In order to achieve the above goals, this work was divided into a number of sections. Before the compatibility of, and control over, these networking technologies could be explored, a thorough knowledge of the underlying workings of IEEE 1394 and Ethernet AVB networks was needed. This investigation focused on the core requirements that make these networks suitable for transferring realtime audio data, as identified above. These two networking technologies each provide their own mechanisms for achieving these goals.

On an IEEE 1394 bus, the elected *isochronous resource manager* (IRM) node keeps track of the amount of bandwidth remaining on the bus, and keeps track of isochronous channel numbers that are available for use. Nodes wishing to transmit isochronous stream packets request bus bandwidth and an isochronous channel number from the IRM. If the request succeeds, these resources are reserved for the node that requested them (see Section 3.1 “Resource Reservation for IEEE 1394”). On an Ethernet AVB network, a talker wishing to transmit a realtime stream sends a frame onto the network containing a *talker advertise* attribute. This attribute contains a description of the resource requirements of the stream. The attribute is propagated throughout the network and, given sufficient resources, is received by all bridges and end stations. When a bridge receives a *talker advertise* attribute, it is able to determine whether each of its ports are able to support the stream or not. The *talker advertise* attribute is transmitted out of each port that is able to support the stream. When a listener would like to receive a particular stream, it transmits a frame to the network containing a *listener ready* attribute. This attribute contains the stream ID of the stream that listener would like to

receive. The network sends this attribute directly to the talker. As the attribute propagates through the bridges of the network, if the stream resources are available, they are reserved for the stream. When the *listener ready* attribute is received by the talker, the talker knows that the network has reserved the resources required for the stream, and is able to start transmission of the stream (see Section 3.2 “Resource Reservation for Ethernet AVB”).

IEEE 1394 and Ethernet AVB each provide mechanisms that allow for the deterministic and timely transmission of realtime stream data. IEEE 1394 operates on a 125 µs isochronous cycle. Each node that has reserved bandwidth (via the IRM) is given an opportunity to transmit data (per requested isochronous channel) once per isochronous cycle (i.e., each node is guaranteed that it will be able to transmit its isochronous data once per isochronous cycle) (see Section4.1 “Determinism for IEEE 1394”). Ethernet AVB devices tag stream data with specific priority values used to identify stream data. This allows Ethernet AVB bridges to apply specific frame forwarding to these frames. Each outbound port on an Ethernet AVB bridge contains a number of queues containing frames awaiting transmission. Higher numbered queues are given higher preference than lower numbered queues for frame transmission. Stream frames are placed into the highest numbered queues (with class A streams going into the highest numbered queue). Each queue that supports stream reservation keeps track of the amount of bandwidth reserved for it, and through the use of the credit-based shaper algorithm, ensures that stream frames are transmitted evenly onto the network, and that the bandwidth is not oversubscribed (see Section 4.2 “Determinism for Ethernet AVB”).

IEEE 1394 and Ethernet AVB natively provide mechanisms that allow the nodes on the network to all share a common sense of time. Each network elects a master node from which all other nodes on the network derive their time (see Chapter 5 “Timing and Synchronisation”).

Media transport protocols have been defined to allow for the transportation of audio data between devices on IEEE 1394 and Ethernet AVB networks. The formatting of the media packets, the formatting of the media data itself, and the timing and synchronisation mechanisms employed by the protocols was examined in Chapter 6 “Media Transport Protocols”. The AVTP protocol, used to transfer audio data between devices on an AVB network, borrows heavily from the CIP and AM824 protocols defined for transferring audio data between devices on an IEEE 1394 network. Conceptually, audio data is transferred between devices in a stream, with a stream being composed

of individual channels of audio. The timing and synchronisation mechanisms employed by the protocols vary in that they use different presentation timestamp formats.

During the course of this work, the Ethernet AVB standards were under development and commercially available Ethernet AVB capable devices did not exist. Proof of concept Ethernet AVB endpoint devices were developed to run on general purpose computers running the Linux operating system. Ethernet AVB endpoint devices were developed to allow for the transmission and reception of audio streams across Ethernet networks. These devices are able to capture analogue audio through an analogue sound card, and encapsulate that audio in AVTP frames for transmission over an Ethernet AVB network. These devices are also able to receive AVTP frames containing audio data, and transfer this audio to the analogue interface of the device. IEEE 1394/Ethernet AVB audio gateway devices were also developed. These devices allow for audio streams to be transferred between IEEE 1394 and Ethernet AVB networks. The channels of audio data contained within packets received via one interface of the device are extracted and placed into packets for the other interface of the device to transmit (see Chapter 10 “Ethernet AVB Devices”).

The use of these devices, along with the use of existing IEEE 1394 endpoint devices, allowed for the development of a means to transfer audio streams between IEEE 1394 and Ethernet AVB networks with the protocols and procedures defined in IEC 61883 and IEEE 1722. These devices make use of the CIP, AM824 and AVTP protocols in order to achieve this goal. These protocols make use of the same audio formats, thus alleviating the need to do any transcoding between audio data received on one interface, and the audio data transmitted on the other interface. IEEE 1722 does not define cross network timing and synchronisation mechanisms, thus this work proposed a means to provide timing and synchronisation mechanisms between IEEE 1394 and Ethernet AVB networks. Timing and synchronisation mechanisms were proposed to enable the synchronisation of multiple streams being transferred across these disparate networks. The timing and timestamps used by these networks are not compatible, thus timestamp values conveyed in packets have to be converted from one format to another when transferred across the audio gateway device. These timestamps also have to be regenerated relative to the time of the network on which they are transmitted. This work proposed a set of formulae for achieving these goals (see Section 10.3 “Timing and Synchronisation”).

In Chapter 7 “Standards-Based Command and Control Protocols”, a number of command and control protocols were examined at a high-level to determine a suitable protocol for providing control over

the IEEE 1394 and Ethernet AVB audio devices that were developed as part of this study. Of the protocols examined, it was observed that they provide similar hierarchical parameter addressing and parameter access functionality. It was decided that the XFN command and control protocol would be used during this study. During this study, the XFN protocol was under development as an AES standard. It provides a rich set of features and functionality geared towards networked realtime audio and video equipment. Source code of an implementation of the XFN protocol was made available. This source code was developed to be cross platform such that it could be built on Windows, Linux, and other platforms. The XFN protocol has a simple design and natural parameter addressing. The API functions provided by the source code are easy to use and provide the ability to build XFN address hierarchies. Application specific callback functions can be associated with these hierarchies. This enables minimal application code to be written in order to implement an XFN controllable application.

In order to initially examine the capabilities of the XFN protocol, simple proof of concept devices were developed. These devices provide a means to tunnel Ethernet traffic over IEEE 1394 networks. Packet formats were developed to allow for the fragmentation of Ethernet frames (used when an Ethernet packet is larger than the maximum payload size of the isochronous packets), and for the packing of Ethernet frames (used when multiple Ethernet frames can be accommodated in an isochronous packet). The XFN protocol was used to successfully provide control over the parameters of the devices to enable tunnelling to take place across IEEE 1394 buses: the protocol was used to establish stream connections across IEEE 1394 buses, and was used to manipulate other parameters relating to the streams, such as the maximum payload size of each transmitted isochronous packet. The isochronous stream used to carry Ethernet traffic between tunnel nodes was modelled as a multicore. A graphical patchbay application was developed and successfully used to represent and control the parameters of the tunnelling devices. The application used the XFN protocol to discover the availability of the tunnelling devices on a network, the availability of multicore endpoints on the devices, and to create and break stream connections between the multicore endpoints (see Chapter 8 “Tunnelling”).

Following this proof of concept, each of the developed Ethernet AVB devices, as well as the existing IEEE 1394 endpoint devices were implemented such that they are remotely controllable via the XFN protocol. Each device contains an XFN stack that allows for the creation of an XFN address hierarchy, and allows for the reception and transmission of XFN messages. As with the tunnelling

nodes, the Ethernet AVB devices implemented pre-existing address hierarchies and parameters from the XFN specification that allow for the discovery of the devices by other XFN capable devices. These devices also implement parameters that allow for control over the internal audio routing configuration of the devices, and allow for control over the parameters that allow for stream connections to be established across digital networks.

As with the tunnelling devices, the IEEE 1394 endpoint devices and the IEEE 1394/Ethernet AVB audio gateway devices implement parameters that enable IEEE 1394 isochronous streams to be remotely established between devices on IEEE 1394 buses. In each instance, the XFN protocol views each isochronous stream as a multicore. Each one of the device's multicore endpoints has parameters that enable streams to be established. Starting a source IEEE 1394 multicore, by setting its *start* parameter, causes the device to request bus bandwidth and an isochronous channel number from the IRM. If these resources have been obtained, stream transmission begins. Starting a sink multicore, by settings its *start* parameter, causes a device to be configured to receive the stream identified by the multicore's *isochronous channel number* parameter. The Ethernet AVB endpoint and IEEE 1394/Ethernet AVB devices were implemented such that the XFN protocol also views streams across Ethernet AVB networks as multicores. The AVB multicores have parameters that enable stream establishment to take place between devices on an Ethernet AVB network. The Ethernet AVB devices introduce additional parameters to the XFN specification that allow for stream connections to be established across Ethernet AVB networks. A source AVB multicore has an *advertise* parameter associated with it that, when set, causes the device to advertise the stream to the attached Ethernet AVB network via MSRP. A sink AVB multicore has a *listen* parameter associated with it that, when set, causes the device to request attachment to a specified stream via MSRP (Section 11.6 "Stream Establishment").

The XFN specification defines parameters that allow for the routing of audio signals internal to a device. These parameters represent a patching matrix where a *crosspoint* parameter enables the routing of an audio signal from a signal source point to a signal destination point within a device. The existing IEEE 1394 audio devices use the parameters to route single audio channels from the inputs of the devices, to the outputs of the devices. In the Ethernet AVB endpoint devices, *crosspoint* parameters are used to route audio signals from the reception multicores of the device to its analogue output, and vice versa. In the IEEE 1394/Ethernet AVB audio gateway devices, *crosspoint*

parameters were successfully used to route bundles of audio from their reception multicores to their transmission multicores.

The consistent use of parameters across a diverse range of devices simplifies the control over these devices:

- The consistent use of the XFN *crosspoint* parameter enables a patchbay application to simply discover the signal sources, signal destinations, and signal cross points within a device without concern for the signal formats and the complexities involved in routing the audio data from one signal point and format to another signal point and format within a device. An application is able to instruct a device to enable a crosspoint, and the device deals with the device specifics involved in routing the audio signals.
- The consistent view of a stream as a multicore simplifies the process of discovering stream endpoints that exist on a device. A controller is able to determine the existence of multicore endpoints on a device (regardless of whether they are IEEE 1394 or Ethernet AVB multicore endpoints), and is able to represent them. Due to the disparate methods used to form connections across these networks, unique XFN connection management parameters are defined for each multicore type.

The XFN protocol's generic representation of devices and their connection management parameters allows for software applications to graphically represent audio devices with a consistent user interface. The Connection Manager, developed to represent and control the IEEE 1394 and Ethernet AVB audio devices used during this study, provides a consistent graphical representation to the various internal audio inputs and outputs of a device, the patches between these inputs and outputs, the source and sink multicores that a device has, and the connections between these multicore endpoints. The use of common parameters from the XFN specification allows the Connection Manager to represent devices without having to have knowledge of the internal workings of the devices themselves.

Figure 161 and Figure 162 show the Connection Manager representing IEEE 1394 and Ethernet AVB devices. Also shown are the inputs, outputs and internal patches of an IEEE 1394 and an Ethernet AVB device.

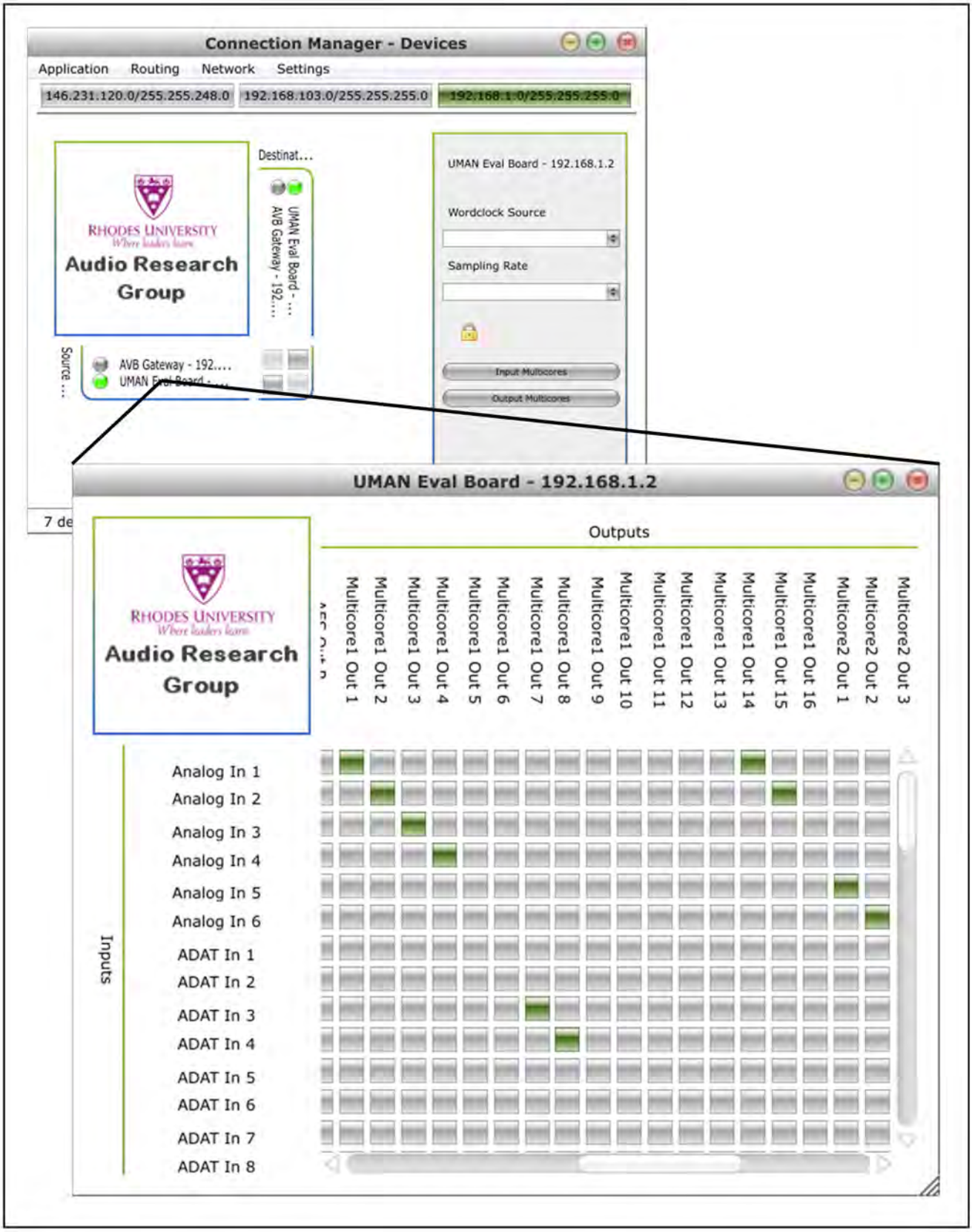

Figure 161: IEEE 1394 endpoint device internal routing matrix

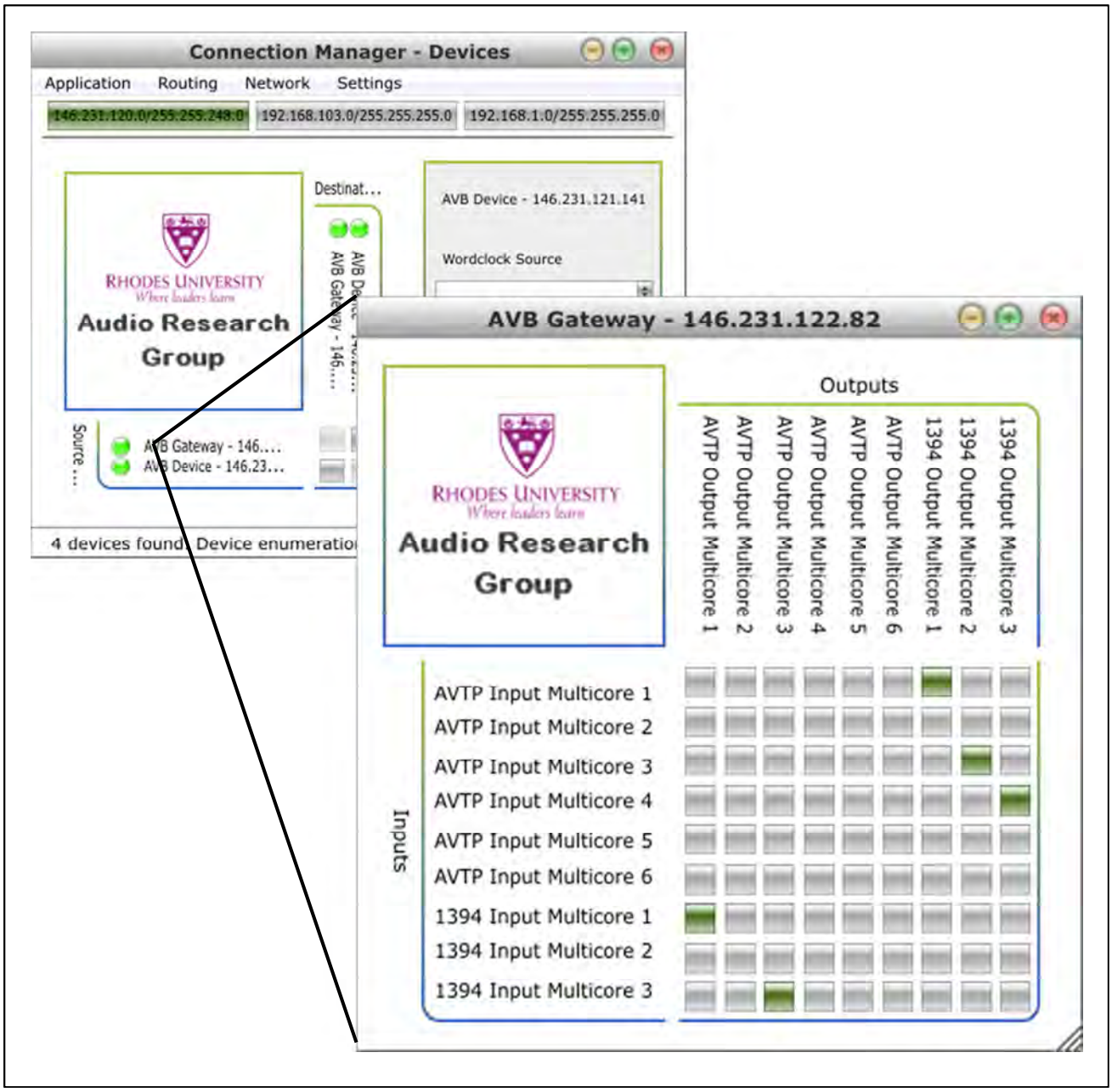

Figure 162: IEEE 1394/AVB audio gateway internal routing matrix

Figure 163 and Figure 164 show how the Connection Manager consistently represents the multicores that exist on IEEE 1394 and Ethernet AVB devices, as well as the connections that exist between these multicore endpoints.

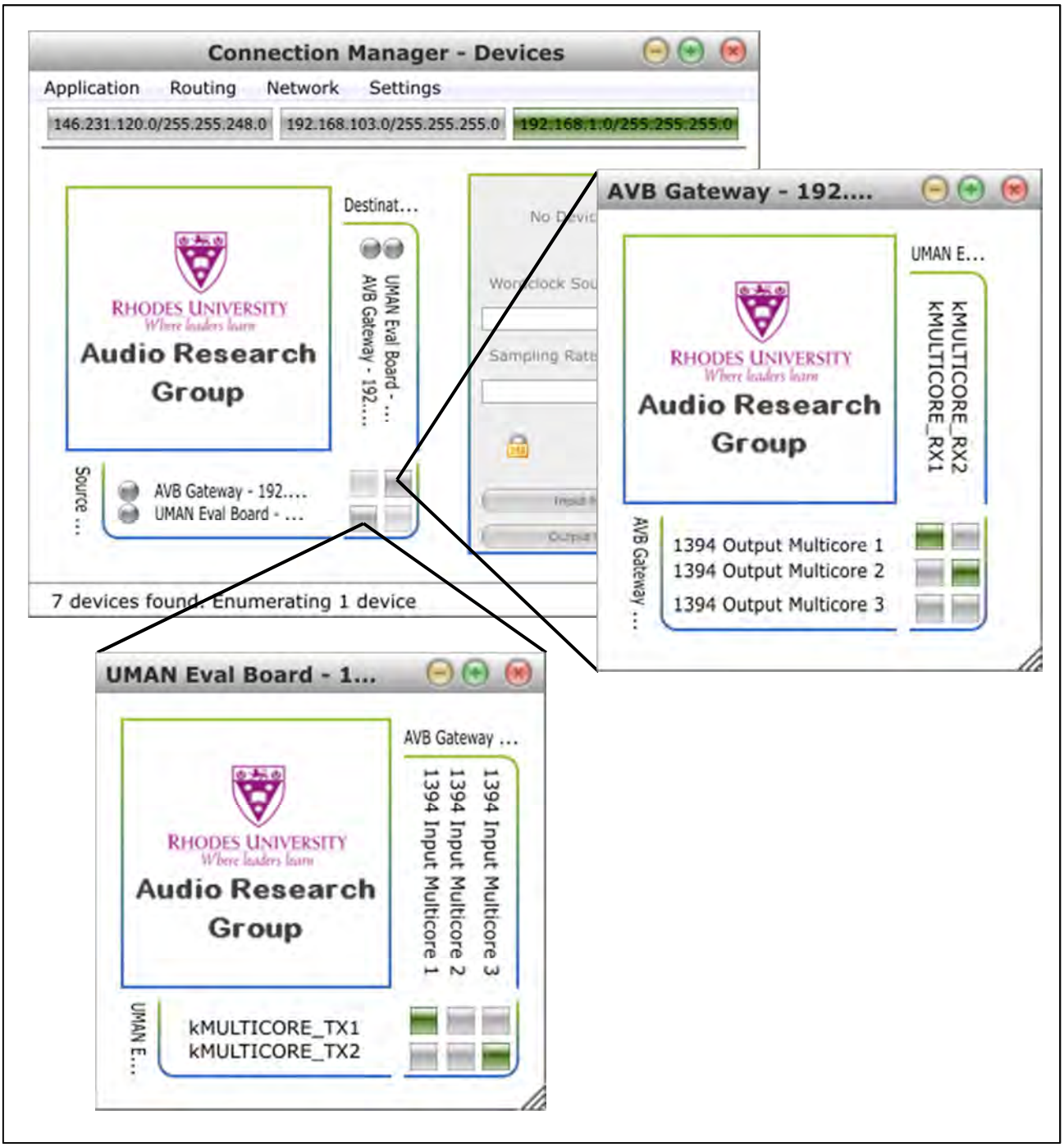

Figure 163: IEEE 1394 multicore patching

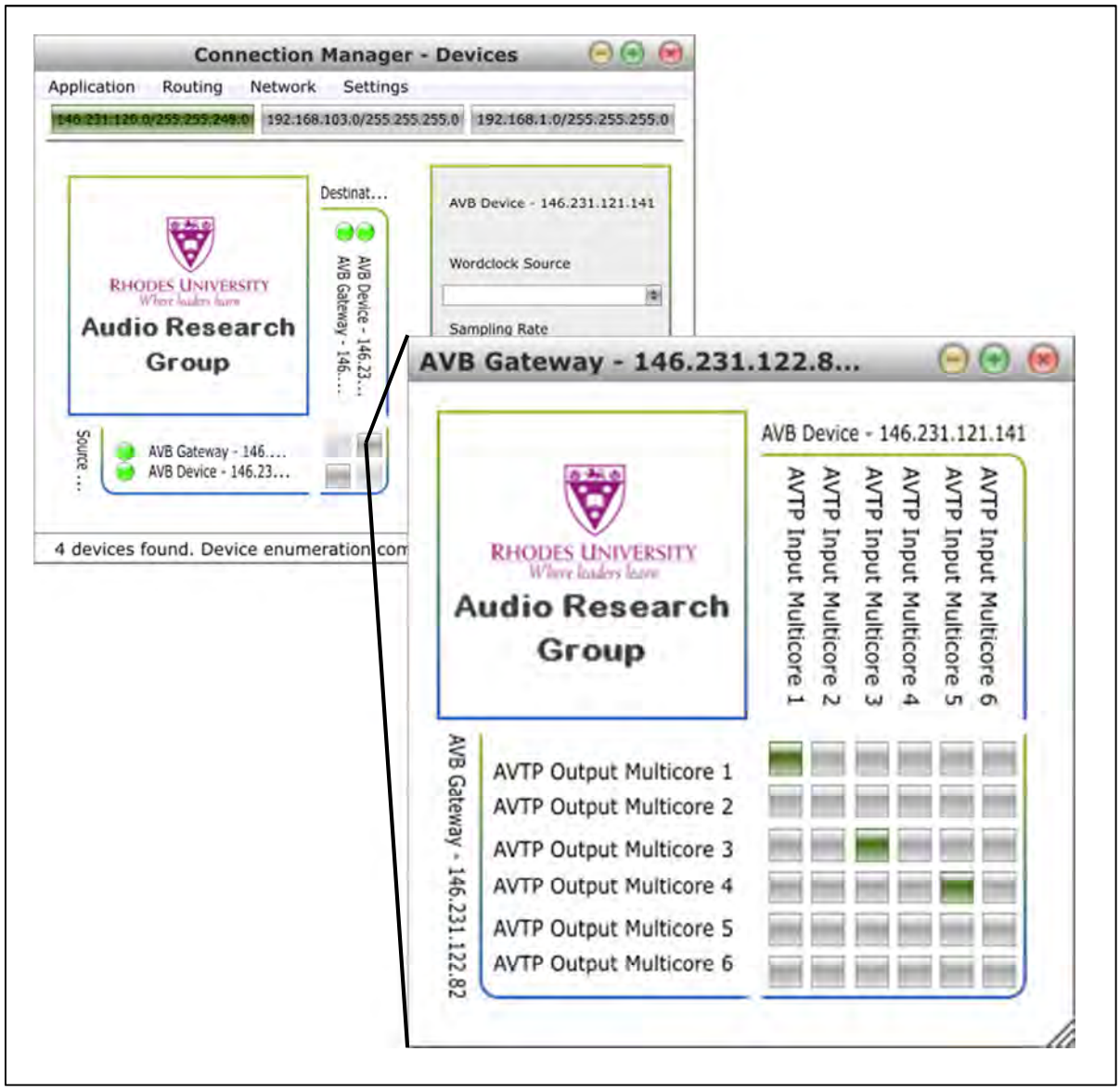

Figure 164: Ethernet AVB multicore patching

The natural hierarchical matrix-styled graphical layout of the Connection Manager is such that audio signals can be routed through complex audio networks by methodically patching from a source device to a destination device:

- Selecting a source device displays its internal routing configurations on a patching matrix. This patching matrix allows patches to be enabled and disabled between the inputs of a source device and its outputs.
- Selecting a crosspoint between a signal source device and a signal destination device displays the signal routing configurations between the devices on a patching matrix. The patching matrix allows for connections between the devices to be created and broken.

- Selecting a destination device displays its internal routing configurations on a patching matrix. This patching matrix allows patches to be enabled and disabled between the inputs of a destination device and its outputs.

This approach to patching has been applied to a number of devices, including the UMAN evaluation breakout boxes, UMAN amplifier nodes, UMAN IEEE 1394 bridges, the developed Ethernet AVB endpoint devices, and the developed IEEE 1394/Ethernet AVB audio gateway devices. These devices have diverse capabilities and implementations, yet have been represented using a common set of XFN parameters, and a common graphical user interface. It is the intention that other networked audio devices (in addition to IEEE 1394 and Ethernet AVB audio devices) can be controlled via the same set of parameters and graphical patching matrix layout thus enabling seamless device integration and control.

12.3 Future Work

As a result of the work performed during this study, a number of possible future projects have emerged:

- The AVB devices implemented as part of this study were done so from the specifications obtained from the IEEE. It would be useful to verify that these device implementations are interoperable with other AVB implementations. Ideally, this verification should involve AVB aware bridges and AVB aware end stations from a number of different manufacturers.
- This work proposed a number of cross network timing and synchronisation mechanisms. It would be useful to implement these mechanisms and verify their accuracy. Their accuracy should be tested over a number of hops.
- The AVB devices implemented as part of this study were done so on general purpose Linux computers. The implementation of the AVB protocols and procedures for the professional audio market requires fine control over hardware devices to enable strict control over latency and the timing and synchronisations mechanisms. It would be useful to verify the cross network streaming capabilities by implementing these mechanisms on specifically designed hardware devices.

- For consumer grade audio applications, it will be useful to create PC sound card drivers to allow audio to be streamed from PC applications to AVB networks. This should also allow for audio applications to receive audio streams from AVB networks.

Bibliography

1. **SMPTE 259M-2008.** *Television – SDTV Digital Signal/Data – Serial Digital Interface.* New York : Society of Motion Picture and Television Engineers, 2008. SMPTE 259M-2008.

2. **SMPTE 292M.** *1.5 Gb/s Signal / Data Serial Interface.* New York : Society of Motion Picture and Television Engineers, 2008. SMPTE 292M.

3. **AES3-2003.** *AES Standard for Digital Audio Engineering - Serial Transmission Format for Two-channel Linearly Represented Digital Audio Data.* New York : Audio Engineering Sociery, 2003. AES3-2003.

4. **Anderson, D.** *FireWire System Architecture.* Massachusetts : MindShare Inc, 1999.

5. IEEE 802.3 Ethernet Working Group. *IEEE 802.3 Ethernet Working Group.* [Online] Institute of Electrical and Electronics Engineers, 19 June 2010. [Cited: 14 September 2010.] http://www.ieee802.org/3/.

6. **IEEE 802.1Q.** *Virtual Bridged Local Area Networks.* New York : Institute of Electrical and Electronics Engineers, 2005. IEEE 802.1Q.

7. The Audio/Video Bridging Task Group. *IEEE 802.1.* [Online] 18 September 2008. [Cited: 28 April 2009.] http://www.ieee802.org/1/pages/avbridges.html.

8. **Yamaha.** Yamaha - Global. [Online] 2011. [Cited: 12 September 2011.] http://www.yamaha.com/.

9. —. *01X Digital Mixing Studio Owner's Manual.* 2003.

10. —. *01V96 Digital Mixing Console Version 2 Owner's Manual.* 2004.

11. —. *Motif XS8 Music Production Synthesizer Owner's Manual.* 2007.

12. —. *mLAN Graphic Patchbay Owner's Manual.* 2004.

13. —. n8 - n12/n8 - Mixers - Music Production Tools - Products - Yamaha United States. [Online] 2011. [Cited: 12 September 2011.] http://usa.yamaha.com/products/music-production/mixers/n12_n8/n8/.

14. —. n12 - n12/n8 - Mixers - Music Production Tools - Products - Yamaha United States. [Online] 2011. [Cited: 12 September 2011.] http://usa.yamaha.com/products/music-production/mixers/n12_n8/n12/.

15. **PreSonus.** PreSonus. [Online] 2011. [Cited: 12 September 2011.] http://www.presonus.com/.

16. —. StudioLive 16.4.2. [Online] 2011. [Cited: 12 September 2011.] http://www.presonus.com/products/Detail.aspx?ProductId=52.

17. **M-Audio.** M-Audio. [Online] 2011. [Cited: 12 September 2011.] http://www.m-audio.com/.

18. —. ProFire 610 - High-Definition 6-in/10-out FireWire Audio Interface with Octane Preamp Technology. [Online] 2011. [Cited: 12 September 2011.] http://www.m-audio.com/products/en_us/ProFire610.html.

19. —. ProFire 2626 - High-Definition 26-in/26-out FireWire Audio Interface with Octane Preamp Technology. [Online] 2011. [Cited: 12 September 2011.] http://www.m-audio.com/products/en_us/ProFire2626.html.

20. **Focusrite.** Audio Interfaces, EQ & Compression, Mic Pres - Focusrite. [Online] [Cited: 12 September 2011.] http://www.focusrite.com/.

21. —. Liquid Saffire 56 Audio Interfaces Professional 28 In / 28 Out with 2 Liquid and 6 Focusrite Pre-amps. [Online] [Cited: 12 September 2011.] http://www.focusrite.com/products/audio_interfaces/liquid_saffire_56/.

22. —. Saffire PRO 40 Audio Interfaces Professional 20 In / 20 Out Firewire interface with eight Focusrite Pre-amps. [Online] [Cited: 12 September 2011.] http://www.focusrite.com/products/audio_interfaces/saffire_pro_40/.

23. **Mackie.** Mackie. [Online] 2011. [Cited: 12 September 2011.] http://www.mackie.com/.

24. —. Mackie Onyx Blackbird: 16x16 Firewire Recording Interface. [Online] 2011. [Cited: 12 September 2011.] http://www.mackie.com/products/onyxblackbird/.

25. —. Onyx 1640i. [Online] 2011. [Cited: 12 September 2011.] http://www.mackie.com/products/onyx1640i/.

26. **Alesis.** Alesis. [Online] 2011. [Cited: 12 September 2011.] http://www.alesis.com/.

27. —. iO|14. [Online] 2011. [Cited: 12 September 2011.] http://www.alesis.com/io14.

28. —. iO|26. [Online] 2011. [Cited: 12 September 2011.] http://www.alesis.com/io26.

29. —. MultiMix 12 FireWire. [Online] 2011. [Cited: 12 September 2011.] http://www.alesis.com/multimix12firewire.

30. —. MultiMix 16 FireWire. [Online] 2011. [Cited: 12 September 2011.] http://www.alesis.com/multimix16firewire.

31. **Allen and Heath.** ALLEN & HEATH // WORLD CLASS MIXING. [Online] 2011. [Cited: 12 September 2011.] http://www.allen-heath.com/.

32. —. ZED-R16. [Online] 2011. [Cited: 12 September 2011.] http://www.allen-heath.com/uk/Products/Pages/ProductDetails.aspx?CatId=ZEDSeries&ProductId=ZEDR16.

33. **TerraTec.** TerraTec. [Online] 2011. [Cited: 12 September 2011.] http://www.terratec.net/.

34. —. Phase X24. [Online] 2011. [Cited: 12 September 2011.] http://www.terratec.net/en/products/PHASE_X24_FireWire_1632.html.

35. **Walker, M.** Sound on Sound. *Terratec Phase 88 Rack FW.* [Online] May 2005. [Cited: 12 September 2011.] http://www.soundonsound.com/sos/may05/articles/terratec88.htm.

36. **Universal Media Access Networks.** [Online] 2011. [Cited: 08 March 2011.] http://www.umannet.com/.

37. **LabX.** Lab X Technologies, LLC. [Online] 2010. [Cited: 12 September 2011.] http://www.labxtechnologies.com/.

38. —. Titanium 411 Ruggedized AVB Ethernet Bridge. [Online] 2010. [Cited: 12 September 2011.] http://www.labxtechnologies.com/connectivity/titanium-411-ruggedized-avb-ethernet-bridge/.

39. —. FPGA AVB Audio Demo Platform. [Online] 2010. [Cited: 12 September 2011.] http://www.labxtechnologies.com/ip-solutions/fpga-based-avb-audio/fpga-avb-audio-demo-platform/.

40. **BSS Audio.** BSS Audio Professional Audio Signal Processing. [Online] 2010. [Cited: 12 September 2011.] http://www.bssaudio.com/.

41. —. BSS AudioGS724T. [Online] 2011. [Cited: 12 September 2011.] http://www.bssaudio.com/productpg.php?product_id=72.

42. **Broadcom Corporation.** Broadcom. [Online] 2011. [Cited: 12 September 2011.] http://www.broadcom.com/.

43. **Broadcom.** BCM57765 - Integrated Gigabit Ethernet and Memory Card Reader Controller. [Online] 2011. [Cited: 12 September 2011.] http://www.broadcom.com/products/Ethernet-Controllers/Enterprise-Client/BCM57765.

44. **Apple Inc.** Apple. [Online] 2011. [Cited: 12 September 2011.] http://www.apple.com/.

45. **Apple.** iMac. [Online] 2011. [Cited: 12 September 2011.] http://www.apple.com/imac/.

46. —. Mac Mini. [Online] 2011. [Cited: 12 September 2011.] http://www.apple.com/macmini/.

47. —. MacBook Pro. [Online] 2011. [Cited: 12 September 2011.] http://www.apple.com/macbookpro/.

48. **Marvell.** Marvell Technology Group Ltd. [Online] 2011. [Cited: 12 September 2011.] http://www.marvell.com/.

49. —. Marvell First to Launch Suite of 802.1 AVB Compliant Products for Robust Delivery of Multimedia Content over Ethernet. [Online] 27 August 2009. [Cited: 12 September 2011.] http://www.marvell.com/technologies/avb/audio_video_bridging_yukon_linkstreet_kirkwood_avb/release/1322/.

50. **Crown International.** Crown International. [Online] 2011. [Cited: 12 September 2011.] http://www.crownaudio.com/.

51. —. PIP-USP4 Module. [Online] 2011. [Cited: 12 September 2011.] http://www.crownaudio.com/pip_htm/usp4.htm.

52. —. CTs Series. [Online] 2011. [Cited: 12 September 2011.] http://www.crownaudio.com/amp_htm/cts.htm.

53. **IEC 61883-6.** *Consumer audio/video equipment. Digital interface – Part 6: Audio and music data transmission protocol.* Geneva : International Electrotechnical Commission, 2005. IEC 61883-6.

54. **IEC 61883-1.** *Consumer audio/video equipment. Digital interface – Part 1: General.* Geneva : International Electrotechnical Commission, 2003. IEC 61883-1.

55. **IEEE 1722.** *Standard for Layer 2 Transport Protocol for Time Sensitive Applications in a Bridged Local Area Network.* New York : Institute of Electrical and Electronics Engineers, 2011. IEEE 1722.

56. **IEEE 1722.1.** *Draft Standard for Standard for Standard Device Discovery, Connection Management and Control Protocol for IEEE 1722 Based Devices.* New York : Institute of Electrical and Electronics Engineers, 2011. IEEE 1722.1 Draft 16.

57. **Case, J, et al.** *A Simple Network Management Protocol (SNMP).* 1990. RFC 1157.

58. **Open Sound Control.** [Online] The Center For New Music and Audio Technology (CNMAT), UC Berkeley. [Cited: 21 February 2011.] http://opensoundcontrol.org/.

59. **AES-X170.** *DRAFT AES Standard for Audio Applications of Networks - Integrated Control, Monitoring, and Connection Management for Digital Audio and Other Media Networks.* New York : AES, 2011. AES-X170.

60. —. *DRAFT AES Informative Document for Audio Applications of Networks - Integrated Control, Monitoring, and Connection Management for Digital Audio and Other Media Networks.* New York : AES, 2011. AES-X170.

61. **Postel, J.** *Internet Protocol.* 1981. RFC 791.

62. **IEEE 1394.1.** *Standard for High Performance Serial Bus Bridges.* New York : Institute of Electrical and Electronics Engineers, Institute of Electrical and Electronics Engineers, 2004. IEEE 1394.1.

63. **Cisco.** *Token Ring-to-Ethernet Migration.* [Online] [Cited: 25 October 2010.] http://www.cisco.com/en/US/solutions/collateral/ns340/ns394/ns74/ns149/net_business_benefit09186a00800c92b9_ps6600_Products_White_Paper.html.

64. **Teener, M and Gaél, M.** Ethernet in the HD studio. *Broadcast Engineering.* [Online] 1 May 2008. [Cited: 15 February 2010.] http://broadcastengineering.com/hdtv/ethernet-hd-studio/.

65. **IEEE 802.5.** IEEE 802.5 Web Site. *IEEE 802.5 Web Site.* [Online] Institute of Electrical and Electronics Engineers, 22 March 2004. [Cited: 14 September 2010.] http://www.ieee802.org/5/.

66. **IEEE 802.11.** IEEE 802.11 Wireless Local Area Networks. *IEEE 802.11 Wireless Local Area Networks.* [Online] Institute of Electrical and Electronics Engineers. [Cited: 14 September 2010.] http://www.ieee802.org/11/.

67. **IEEE 801.1Qat.** *Stream Reservation Protocol.* New York : Institute of Electrical and Electronics Engineers, 2010. IEEE 802.1Qat.

68. **IEEE 802.1Qav.** *Forwarding and Queuing Enhancements for Time-Sensitive Streams.* New York : Institute of Electrical and Electronics Engineers, 2009. IEEE 802.1Qav.

69. **IEEE 802.1AS.** *Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks.* New York : Institute of Electrical and Electronics Engineers, 2011. IEEE 802.1AS.

70. **IEEE 802.1BA.** *Audio Video Bridging (AVB) Systems.* New York : Institute of Electrical and Electronics Engineers, 2011. IEEE 802.1BA Draft 2.3.

71. **IEEE 802.1ak.** *Multiple Registration Protocol.* New York : Institute of Electrical and Electronics Engineers, 2007. IEEE 802.1ak.

72. **Garner, G and Ryu, H.** Synchronization of Audio/Video Bridging Networks Using IEEE 802.1AS. *IEEE Communications Magazine.* 2011, Feburary 2011.

73. **Stanton, K.** *802.1AS Tutorial.* 2008.

74. **International Electrotechnical Commission.** [Online] http://www.iec.ch/.

75. **IEC 62379-1.** *Common control interface - Part 1: General.* s.l. : International Electrotechnical Commission, 2007. IEC 62379-1.

76. **Postel, J.** *User Datagram Protocol.* 1980. RFC 768.

77. CobraNet. [Online] [Cited: 26 May 2011.] http://www.cobranet.info/.

78. **Gross, K and Holtzen, T.** *Controlling and Monitoring Audio Systems with Simple Network Management Protocol (SNMP).* Boulder : Audio Engineering Society, 1998.

79. **Gross, K.** *AVBC via SNMP for 1722.1.* s.l. : AVA Networks, 2010.

80. **Postel, J.** *Transmission Control Protocol.* 1981. RFC 793.

81. **International Telecommunication Union.** ASN.1 & OID Project. [Online] International Telecommunication Union, 2011. [Cited: 26 May 2011.] http://www.itu.int/ITU-T/asn1/.

82. **International Organization for Standardization.** ISO - International Organization for Standardization. [Online] 2011. [Cited: 26 May 2011.] http://www.iso.org/.

83. **US Department of Defense.** The Official Home of the Department of Defense. [Online] [Cited: 26 May 2011.] http://www.defense.gov/.

84. **Internet Assigned Numbers Authority.** Internet Assigned Numbers Authority. [Online] http://www.iana.org/.

85. **Net-SNMP.** Net-SNMP. [Online] [Cited: 29 March 2011.] http://www.net-snmp.org/.

86. **IEC 62379.** IEC 62379 Common Control Interface. [Online] 2010. [Cited: 09 January 2011.] http://www.iec62379.org/. 62379.

87. **IEC 62379-2.** *Common control interface – Part 2: Audio.* s.l. : International Electrotechnical Commission, 2008. IEC 62379-2.

88. **IEC 62379-5-1.** *DRAFT Common control interface for networked digital audio and video products – Part 5-1: Transmission over networks – General.* Geneva : International Electrotechnical Commission, 2011. IEC 62379-5-1.

89. **IEC 62379-5-2.** *DRAFT Common control interface for networked digital audio and video products – Part 5-2: Transmission over networks – Signalling.* Geneva : International Electrotechnical Commission, 2011. IEC 62379-5-2.

90. **Koftinoff, J.** *AVBC – A protocol for connection management and system control for AVB.* 2010. Version 1.3.

91. **Bencina, R.** oscpack. *A simple C++ Open Sound Control (OSC) packet manipulation library.* [Online] [Cited: 26 April 2011.] http://code.google.com/p/oscpack/.

92. **Audio Engineering Society.** AES Standards. [Online] [Cited: 26 April 2011.] http://www.aes.org/standards/.

93. **Raw Material Software.** *JUCE.* [Online] [Cited: 15 March 2011.] http://www.rawmaterialsoftware.com/.

94. **MIDI Manufacturers Association.** MIDI Manufacturers Association. [Online] 2010. [Cited: 15 May 2011.] http://www.midi.org/.

95. The Linux Home Page at Linux Online. [Online] [Cited: 14 February 2010.] http://www.linux.org/.

96. AlsaProject. [Online] [Cited: 28 March 2011.] http://www.alsa-project.org/.

97. **Foulkes, P.** A grid based approach for the control and recall of the properties of IEEE1394 audio devices. *A thesis submitted in fulfilment of the requirements for the degree of MASTER OF SCIENCE of RHODES UNIVERSITY.* 2008.

98. **mLAN Central.** mLAN Central. [Online] [Cited: 21 July 2011.] http://www.mlancentral.com/.

99. **Chigwamba, N. and Foss, R.** *Enhanced End-User Capabilities in High Speed Audio Networks.* s.l. : AES, 2007.

100. **Otari.** *mLAN Control Software Operation Manual.* 2005.