Application Level Resource Management for the IEEE 1394 Serial Bus

by

John R. Foxgord
B.Sc., University of Victoria, 1994

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. G. C. Shoja, Supervisor (Department of Computer Science)

Dr. J. C. Muzio, Departmental Member (Department of Computer Science)

Dr. E. G. Manning, Outside Member (Department of Electrical and Computer Engineering)

Dr. T. A. Gulliver, External Examiner (Department of Electrical and Computer Engineering)

© John R. Foxgord, 2001
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part,
by photocopying or other means, without the permission of the author.

Supervisor: Dr. G. C. Shoja

Abstract

The IEEE 1394 standard defines a serial data-bus which is well suited to the transmission of real-time media data between devices within the home entertainment environment. A 1394 network can support multiple high-bandwidth media sources, while also accommodating asynchronous messaging. While a fair-share scheme is specified for the scheduling of asynchronous messages, the bandwidth and related resources used for the streaming of isochronous data are freely acquired by the individual media applications. Therefore, isochronous resources are acquired in a first-come-first-served fashion and the devices which cannot acquire the necessary resources will not commence transmission.

We address these inefficiencies by shifting from the anarchic acquisition of resources to a centralized management scheme which allocates resources based on the individual requirements of applications and the common management goals of the network. For robustness, if the node that houses the manager fails, then another node can assume the management responsibility after a bus reset. Khan's Utility Model is employed to address our requirements for resource management and application adaptation. Our approach to these challenges can be classified as Application Level Resource Management.

This thesis presents the design and implementation of network management tools for IEEE 1394 based on the principles of the Utility Model. These tools consist of a centralized resource manager and distributed client management components. An OPNET simulation of the IEEE 1394 network with our added management components illustrate the possible efficiency gains in the context of a residential environment. The simulation also verifies our approach for transporting the management information between client and manager components. We conclude by proposing several directions for future work.

Examiners:

Dr. G. C. Shoja, Supervisor (Department of Computer Science)

Dr. J. C. Muzio, Departmental Member (Department of Computer Science)

Dr. E. G. Manning, Outside Member (Department of Electrical and Computer Engineering)

Dr. T. A. Gulliver, External Examiner (Department of Electrical and Computer Engineering)

Table of Contents

Abstract ii

Table of Contents iv

List of Tables vii

List of Figures ix

Acknowledgments. xv

1: Introduction 1
- 1.1: The Problem 3
- 1.2: Major Issues 6
- 1.3: Focus and Approach 8
- 1.4: Outline 10

2: Overview of IEEE 1394 Serial Bus 11
- 2.1: Functional Overview 12
- 2.2: Architectural Overview 15
- 2.3: The IEEE 1394 Address Space 17
- 2.4: Bus Reset and Configuration 18
- 2.5: The Bus Arbitration Model 19
- 2.6: Asynchronous Transactions 21
- 2.7: Isochronous Transactions 25

3: The Utility Model 33
- 3.1: The Adaptive Multimedia System 33
- 3.2: The Adaptive Multimedia Problem 37
- 3.3: Mapping the AMP to the MMKP 38
- 3.4: The HEU Algorithm 39

4: Enhanced Management of the 1394 Serial Bus 43
4.1: Application Resource Requirements 43
4.2: Partitioning IEEE 1394 Resources 45
4.3: Reservation Semantics 47
4.4: Resource Management Architecture 48
4.5: Resource Management Server 50
4.6: Resource Management Client 51

5: Enhanced Resource Management Protocols 53
5.1: RMS/RMC Start-up and Re-configuration 53
5.2: The RMC to RMS Signalling Protocol 56
5.3: Request and Allocation Protocol 62

6: IEEE 1394 Simulator 67
6.1: Physical Layer 71
6.2: Link Layer 76
6.3: Transaction Layer 78
6.4: Driver Layer 79
6.5: Application Level Processes 82

7: IEEE 1394 Management Implementation 84
7.1: Management Modules 85
7.2: Legacy Modules 93

8: Experiments and Results 101
8.1: Topological Parameters 101
8.2: Module Parameters and Statistics 104
8.3: Experimental Results 113
8.4: Summary of Results 134

9: Conclusion 136
9.1: Summary 136

References 139

Appendix A: IEEE 1394 Asynchronous Packet Formats 141
A.1: Support for the Asynchronous Read Transaction 141
A.2: Support for the Asynchronous Lock Transaction 146
A.3: Support for the Asynchronous Write Transaction 150

Appendix B: RMS Management Messages 156
B.1: Packaging Management Messages 156
B.2: Resource Management Message Formats 157
B.3: Resource Management Packet Formats 158

Appendix C: Resource Management Client 163
C.1: RMC Architecture 163
C.2: RMC Operational Scenarios 167

Appendix D: Resource Management Server 173
D.1: RMS Architecture 173
D.2: RMS Operational Scenarios 178

Appendix E: OPNET IEEE 1394 API 181
E.1: Asynchronous API 181
E.2: Resource Acquisition API 186
E.3: Isochronous API 190
E.4: Miscellaneous API 192
E.5: Isochronous Buffer API 194

List of Tables

Table 2.1: Isochronous Packet Field Contents 26

Table 3.1: Session Quality Profile with Resource and Utility Mappings ... 36

Table 4.1: Specification of Resource Requirements 48

Table 6.2: Link Layer API 77

Table 6.3: Transaction Layer API 79

Table 6.4: Driver Layer API 81

Table 7.1: Resource Management Server FSM States 87

Table 7.2: Resource Management Client FSM States 89

Table 7.3: Managed Isochronous Source FSM States 91

Table 7.4: Isochronous Resource Manager FSM States 94

Table 7.5: Isochronous Resource Manager FSM States 95

Table 7.6: Asynchronous Source FSM States 98

Table 7.7: Asynchronous Sink FSM States 99

Table 8.1: Supported Modules in Managed/Unmanaged Topologies 102

Table 8.2: Parameters for Session Spacing 105

Table 8.3: Statistics for Session Spacing 105

Table 8.4: Managed Isochronous Session Phases 106

Table 8.5: Managed Isochronous Session Parameters 109

Table 8.6: Managed Isochronous Session Statistics 109

Table 8.7: Unmanaged Isochronous Session Phases 111

Table 8.8: Unmanaged Isochronous Session Parameters 111

Table 8.9: Unmanaged Isochronous Session Statistics 112

Table 8.10: Asynchronous Session Parameters 113

Table 8.11: Parameter Values for Simulations 114

Table 8.12: MISRC quality profile with requirements and utilities 115

Table A.1: Quadlet Read Transaction Request Packet Fields 142

Table A.2: Quadlet Read Transaction Response Packet Fields 143

Table A.3: Block Read Transaction Request Packet Fields 144

Table A.4: Block Read Transaction Response Packet Fields 146

Table A.5: Lock Transaction Request Packet Fields 147

Table A.6: Lock Transaction Response Packet Fields 149

Table A.7: Quadlet Write Transaction Request Packet Fields 151

Table A.8: Block Write Transaction Request Packet Fields 153

Table A.9: Write Transaction Response Packet Fields 154

Table B.1: Resource Request Packet Fields 160

Table B.2: Resource Allocation Packet Fields 161

Table B.3: Resource Release Packet Fields 162

List of Figures

Figure 1.1: A generalized distributed multimedia system 2
Figure 1.2: Accessing the IRM 3
Figure 1.3: Residential IEEE 1394 installation 4
Figure 1.4: Quality Adaptation 5
Figure 2.1: IEEE 1394 interconnectivity throughout the home 12
Figure 2.2: Topology and Services 14
Figure 2.3: IEEE 1394 Protocol Layers 15
Figure 2.4: Serial Bus Address Space 18
Figure 2.5: Cycle Timeline 20
Figure 2.6: Cycle Start Skew 21
Figure 2.7: Acknowledgement Packet 23
Figure 2.8: Request and Response Phases 24
Figure 2.9: Isochronous Packet Format 26
Figure 2.10: Non-concatenated Isochronous Transactions 27
Figure 2.11: Concatenated Isochronous Transactions 27
Figure 2.12: Location of Resource Registers in Memory Space 29

Figure 2.13:	Channels Available Register	29
Figure 2.14:	Acquisition of Isochronous Channels	30
Figure 2.15:	Bandwidth Available Register	31
Figure 2.16:	Acquisition of Bandwidth Allocation Units	32
Figure 3.1:	Adaptive Multimedia System Requirements	37
Figure 3.2:	The Knapsack Problem	38
Figure 3.3:	Mapping the AMP to the MMKP	39
Figure 3.4:	Residential IEEE 1394 Installation	40
Figure 3.5:	Using the Utility Model in a 1394 Home Installation	41
Figure 4.1:	Distribution of Bandwidth	46
Figure 4.2:	Distribution of Channel Identifiers	47
Figure 4.3:	RMC and RMS Association	49
Figure 4.4:	RMS Architecture	50
Figure 4.5:	Resource Management Client	51
Figure 5.1:	Serial Bus and RMS Setup Sequence	54
Figure 5.2:	RMS Node Register Format	56
Figure 5.3:	RMS Election Protocol: Election Victory	58
Figure 5.4:	RMS Election Protocol: Election Defeat	59
Figure 5.5:	RMS Address Acquisition Protocol	61
Figure 5.6:	RMS and RMC - Request and Allocation Protocol Phases	62
Figure 5.7:	RMC Request - Successful Transaction	63
Figure 5.8:	RMC Request - Lost Request Packet	64
Figure 5.9:	RMC Request - Rejected Request Packet	65

Figure 5.10: RMC Request - Failed Write Transaction 65

Figure 6.1: IEEE 1394 3-Port Node Model (Node Domain) 69

Figure 6.2: 1394 Model Network Model (Network Domain) 70

Figure 6.3: A Connection between Ports 72

Figure 6.4: Connection Model Attributes 73

Figure 6.5: Node Model Attributes 74

Figure 7.1: Application Layer of 1394 OPNET Simulation 84

Figure 7.2: An OPNET FSM 85

Figure 7.3: Resource Management Server FSM 86

Figure 7.4: Resource Management Client 89

Figure 7.5: Managed Isochronous Source FSM 91

Figure 7.6: Isochronous Resource Manager FSM 94

Figure 7.7: Asynchronous Source FSM 97

Figure 7.8: Asynchronous Sink FSM 99

Figure 8.1: Simulation Topology 102

Figure 8.2: Probe Node Architecture 103

Figure 8.3: Inter-Session Spacing for a Single Source 104

Figure 8.4: Managed Isochronous Session Phases 105

Figure 8.5: Managed Isochronous Source: Request/Allocation Phase 106

Figure 8.6: Managed Isochronous Source: Acquisition Phase 107

Figure 8.7: Managed Isochronous Source: Session Data/Adaptation Phase 107

Figure 8.8: Managed Isochronous Source: Reacquisition Phase 108

Figure 8.9: Managed Isochronous Source: Deallocation Phase 108

Figure 8.10: Managed Isochronous Source: Session Release Phase 109

Figure 8.11: Unmanaged Isochronous Source: Session Phases 110

Figure 8.12: Unmanaged Isochronous Source: Session Data Phase 111

Figure 8.13: Asynchronous Source: Asynchronous Session 113

Figure 8.14: Session Startup Delay versus Number of Packets/Session 116

Figure 8.15: Session Startup Delay versus Inter-Session Gap 117

Figure 8.16: Concurrent Sessions versus Number of Packets/Session 118

Figure 8.17: Concurrent Sessions versus Inter-Session Gap 119

Figure 8.18: Asynchronous Utilization versus Number of Packets/Session . 120

Figure 8.19: Asynchronous Utilization versus Inter-Session Gap 121

Figure 8.20: Managed Async Utilization versus Inter-Session Gap 122

Figure 8.21: Unmanaged Async Utilization versus Inter-Session Gap 123

Figure 8.22: Isochronous Utilization versus Number of Packets/Session . . . 124

Figure 8.23: Isochronous Utilization versus Inter-Session Gap 125

Figure 8.24: Active Operating Qualities versus Number of Packets/Session 126

Figure 8.25: Active Operating Qualities versus Inter-Session Gap 127

Figure 8.26: Req/Alloc Delay versus Number of Packets/Session 128

Figure 8.27: Startup delays versus Inter-Session Gap 129

Figure 8.28: Session Data Efficiency versus Number of Packets/Session . . . 130

Figure 8.29: Session Data Efficiency versus Inter-Session Gap 131

Figure 8.30: Reallocations/session versus Number of Packets/Session 132

Figure 8.31: Reallocations/session versus Inter-Session Gap 132

Figure 8.32: Reallocation Overhead versus Number of Packets/Session 133

Figure 8.33: Reallocation Overhead versus Inter-Session Gap 133

Figure 8.34: RMS Overhead versus Inter-Session Gap 134

Figure A.1: Quadlet Read Transaction Request Packet 141

Figure A.2: Quadlet Read Transaction Response Packet 143

Figure A.3: Block Read Transaction Request Packet 144

Figure A.4: Block Read Transaction Response Packet 145

Figure A.5: Lock Transaction Request Packet 147

Figure A.6: Lock Transaction Response Packet 149

Figure A.7: Quadlet Write Transaction Request Packet 151

Figure A.8: Block Write Transaction Request Packet 152

Figure A.9: Write Transaction Response Packet 154

Figure B.1: FCP Packet Format 156

Figure B.2: RESLEVEL Structure 157

Figure B.3: QOSPROFILE Structure 158

Figure B.4: RESALC Structure 158

Figure B.5: Resource Request Packet (RMREQ) 159

Figure B.6: Resource Allocation Packet (RMALC) 160

Figure B.7: Resource Release Packet 161

Figure C.1: RMC Modules 163

Figure C.2: RMC Profile DB Record 166

Figure C.3: Resource Request Scenario 168

Figure C.4: Adaptation Indication Scenario 170

Figure C.5: Release Request Scenario 171

Figure D.1: RMS Modules 173

Figure D.2: RMS Profile DB Record 176

Figure D.3: Request/Release Indication Scenario 179

Acknowledgments

I would like to take this opportunity to thank my supervisor Dr. Ali Shoja for his continuous support during my M.Sc. program here at the University of Victoria. I am very grateful for his guidance, encouragement, patience, throughout the development of this thesis and other work.

I would also like to thank Dr. Eric Manning, and other members of my thesis committee for taking time from their busy schedules to offer valuable suggestions on this thesis.

In addition, I would like to thank PANDA group members for their companionship during the course of my graduate study. I also acknowledge Mostofa Ackbar for providing me with a stable and readable implementation of Shahadat Khan's HEU algorithm.

I acknowledge the software engineers at OPNET who provided precise advice when my simulation did not perform as required. The 1394 simulator would not have been completed without their assistance.

Moreover, I gratefully acknowledge Sony DSL, San Jose, California for their financial support.

Finally, I would like to express my sincere gratitude to my parents. Without their selfless support, this would not have been possible.

1 Introduction

A major obstacle to the realization of distributed multimedia systems is the lack of a network infrastructure that can provide soft-realtime guarantees to high-bandwidth media sources. The Internet provides adequate evidence that best-effort networks fall short of supporting even the simplest of media streams. Our work addresses the management of network resources within the context of a home entertainment system based on the IEEE 1394 Serial-bus standard [4][6]. Our proposed management tools provide the capability of admission control and the efficient utilization of network resources.

The IEEE 1394 Serial-bus standard defines a networking architecture which provides the bandwidth guarantees required to support Digital Versatile Disk (DVD), High Definition Digital Television (HDTV), Digital Video (DV), and other bandwidth intensive technologies within distances acceptable for the home entertainment environment. The standard also defines support for asynchronous messages; these messages commonly carry session setup and control information between multimedia devices. Transmissions of both traffic types are scheduled to provide guaranteed medium access to the isochronous traffic, and a fair distribution of the remaining bandwidth amongst the sources of asynchronous messages. Moreover, 1394 is a desirable technology for inter-connecting consumer electronics devices because it provides a user-friendly means for adding and removing devices from the network. Generally, users can simply modify the network topology without restarting the sessions in progress.

To place our work in a realistic context, we pose a generalized model for a distributed multimedia system. According to the model, a system consists of two or more devices

which can communicate by way of symmetric protocol layers over some physical medium. In our generalized model, each device can support one or more applications; each application is a source or sink for a session, and each session is composed of one or more continuous media streams.

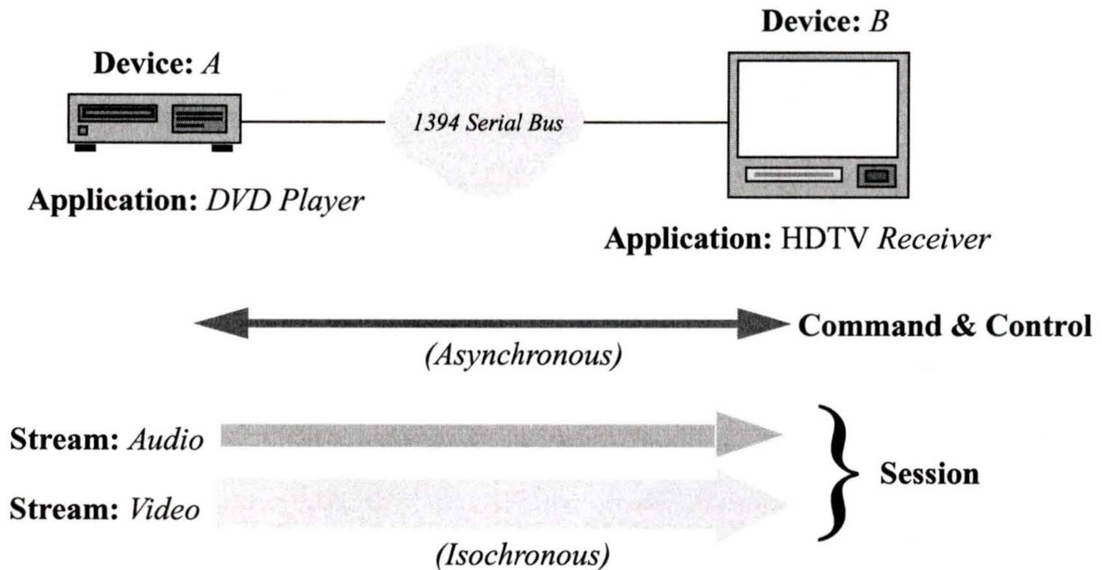


Figure 1.1: A generalized distributed multimedia system

An example which follows the model is illustrated above in Figure 1.1. Our scenario depicts a simple configuration of two devices interconnected over a 1394 Serial bus. The DVD Player application is a source for a session composed of both audio and video streams. The (Digital Television) DTV Receiver application is a sink for these streams. The end-to-end transport of these streams is accomplished by the isochronous services provided by 1394. Additional traffic composed of command & control messages occurs between both applications; these are carried by asynchronous messages. Although the Figure focuses on a pair of devices, it possible that other devices with active applications and sessions are sharing the network resources.

In this Chapter we identify the problems and issues associated with resource management over 1394.

1.1 The Problem

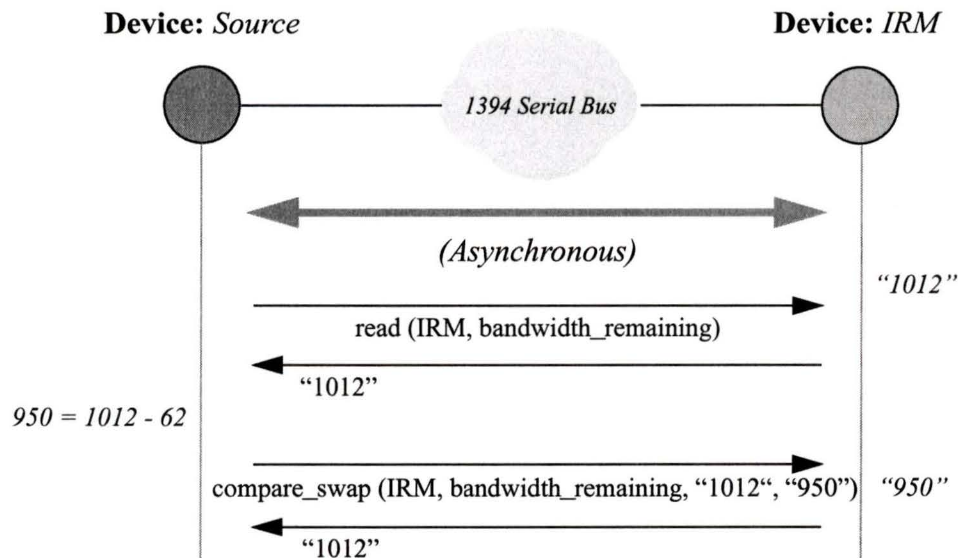


Figure 1.2: Accessing the IRM

In the resource management scheme defined by the 1394 standard, an individual application must obtain isochronous resources by reading and modifying a specific status register on a specific network-resident node also known as the Isochronous Resource Manager (IRM). A simplified illustration of this procedure is presented above in Figure 1.2. Initially, the source device reads the value contained in the *bandwidth_remaining* register; this denotes the total bandwidth which remains available. The source device subtracts its own requirements and then attempt to write the modified value back into the status register using a *compare_swap* operation. Upon the successful completion of the swap operation, the source device can begin streaming isochronous data over the bus. Alternatively, if an application cannot acquire adequate resources, the candidate session does not commence.

After an application has completed its session and ceased transmitting isochronous data, resources are released by accessing the status register and adding back the resources previously acquired. This is similar to the process shown above.

Although the individual devices obtain these resources from a central location, there is

no coordinated management; each device obtains the desired amount without regard for the requirements of other network-resident devices. Hence, there is no mechanism to enforce a priority scheme amongst the devices and their users.

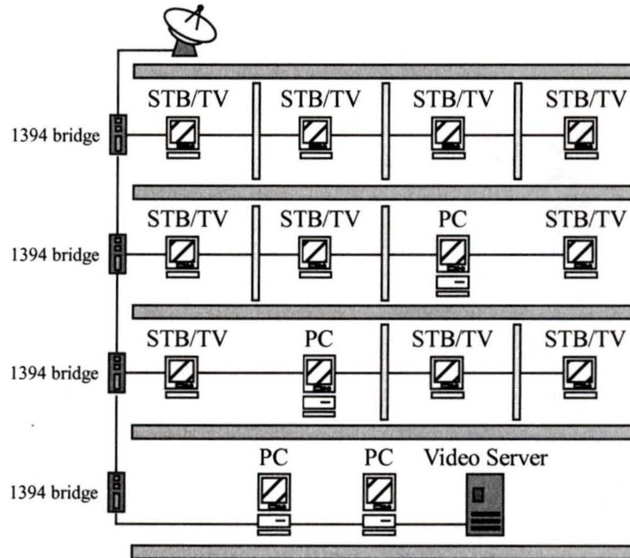


Figure 1.3: Residential IEEE 1394 installation

This standard mechanism for resource management is adequate for small localized topologies, as sufficient resources are likely to be available. However, for a studio/home setting with many audio/video, PC and HDTV devices, as well as for larger residential topologies such as depicted above in Figure 1.3, network resources become scarce, therefore, prudent management is necessary. Moreover, devices which cannot secure the desired amount of resources may not be capable of adapting to operate within the bounds of the available resources. We now consider these two problems in further detail.

1.1.1 Coordination of Resource Management

Currently, a typical network configuration may consist of two devices, for instance, a DV camera connected to a PC. As newer models of consumer electronics devices appear and interconnectivity standards such as IEC 61883 [1] and HAVi [15] prevail, the 1394 network is likely populated with an increasing number of entertainment and productivity devices, spanning the entire home or office.

Although the 1394 standard provides a mechanism so that device resident applications can reserve bandwidth for isochronous traffic, the standard does not define a management scheme which considers the priorities of individual devices and the management goals of the entire network. Resource management on small 1394 installations is unnecessary as one or two devices are unlikely to generate enough traffic that would saturate the total available bandwidth. However, as installations scale to fill the home or office, and as media sources require an increasing amount of bandwidth, contention for resources is anticipated. Hence, the anarchistic approach to resource acquisition does not adequately address resource contention.

1.1.2 Adaptive Support for Applications

Commonly, a device which cannot initially obtain the required resources may periodically retry the above acquisition process. If the acquisition cannot be secured within a reasonable period, the multimedia session fails. An alternative to this all-or-nothing approach is to design applications and their underlying libraries to adapt their behavior to fluctuating resource availability; several qualities of media could be supported, where each quality requires a different quantity of resources.

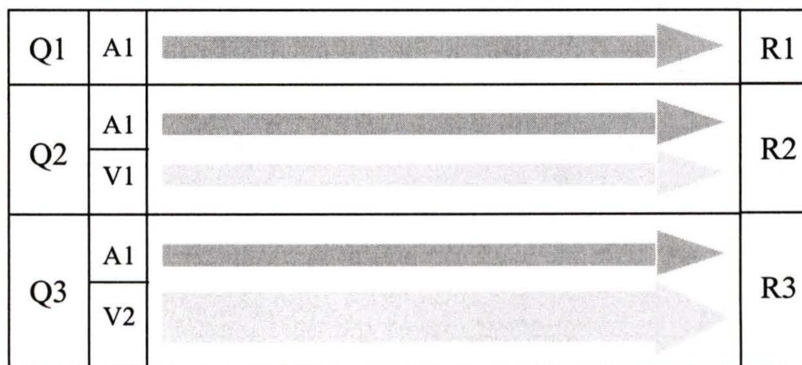


Figure 1.4: Quality Adaptation

Figure 1.4 illustrates a multi-quality approach where the different session qualities {Q1, Q2, Q3} correspond to tuples which specify the constituent audio and video qualities

{(A1,-),(A1,V1),(A1,V2)}, where V2 is a better quality than V1. And finally, each session quality corresponds to a resource requirement Rn where R1 < R2 < R3.

Upon failing to secure the R3 level of resources, an adaptive application can then attempt to acquire the R2 level of resources, and then the R1 level if necessary. Each of these resource levels corresponds to a session quality. Hence, the adaptive application must be capable of adjusting its behavior to ensure its resource utilization does not exceed its allocation.

1.2 Major Issues

In providing resource management for 1394 networks, we must consider the following issues:

1.2.1 Allocation of Resources

Multiple applications on multiple devices contend for a limited quantity of bus resources. If ample resources exist, then all applications receive the desired quantity. However, if available resources do not meet the total requirements, a management entity must dictate how the resources are allocated amongst the applications. After applications have received their allocations and begun to utilize the resources, a new application may request network resources. Depending on the priority of the new application, the resource management components may reallocate resources amongst the existing sessions. Therefore, each resource client component must be informed of, and adapt to its reallocation.

We require a mathematical model whose inputs are: the resource requirements for each application based on the assumption of scalable sessions and the applications' relative priorities or allocation-relative benefits. With these parameters, the model must implement the management goals of the entire network and yield a resource allocation to each application. This process is repeated each time a new session is considered for admission or when an admitted session completes.

1.2.2 Support for Application Adaptation

A device may not always be allocated an ideal quantity of resources. If a device resident

application cannot acquire the resources necessary for supporting a specific session quality, it can either fail to start, or it could renegotiate for a lower quantity. Additionally, an application may receive a reallocation during the duration of a session. This renegotiation or reallocation implies that the application can accommodate a varied quantity of resources, and therefore is able to yield or process an alternate representation of the media data. We assume that the generation and transmission of these different representations require different amounts of resources.

Adaptive applications must support different modes of operation which correspond to specific network conditions. Such flexibility requires the use of scalable media data representations. For instance, a video stream can vary in quality by scaling its temporal or spatial resolutions. An adaptive application should be able to generate or interpret each level of quality. Moreover, for each quality level, the application must know the required quantity of the resources.

1.2.3 Distribution of Management Information

The 1394 standard has defined the IRM as a central entity from which distributed devices can acquire resources; this ensures that the aggregate bandwidth of these acquisitions does not exceed the capacity of the network. Our application level resource management scheme must also guarantee this invariant.

For every session admission or completion, the resource manager may yield a new distribution of resources. The initial request from a prospective device simply requires that the request parameters are efficiently encapsulated within an asynchronous packet and then sent to the resource manager. However, the resulting distribution of the new allocation, along with any necessary reallocations, is not so trivial. This distribution phase may span a sufficient length of time such that resource usage is not synchronized. For instance, some devices may base their bandwidth usage on a recently received reallocation while other devices have not yet received this reallocation. Therefore, global network utilization is based on an inconsistent set of allocations; some nodes may be obeying the most recent allocation, whereas other nodes may be assuming the resource levels dictated by a previous allocation. Hence, the aggregate utilized bandwidth at a specific point in

time may exceed the network's capacity, therefore, violating the 1394 standard.

1.2.4 Support for Legacy Devices

For devices wishing to acquire isochronous resources, the 1394 standard specifies that the device must directly access and modify several IRM-resident status registers which indicate the remaining available resources. As depicted above in Section 1.1, this process requires the direct use of several asynchronous transactions. Our resource management scheme proposes the use of application level management components as an alternative to this direct access.

It is likely that a network which supports application level resource management also harbors legacy devices which rely on the standard IEEE 1394 resource acquisition mechanism. Therefore, our proposed scheme for application level resource management must not monopolize all available network resources.

1.2.5 Bus Reconfiguration

A change in the topology of a 1394 network triggers a bus reset. After the reset, the network reconfigures itself to accommodate the change. After which, as defined by the 1394 standard, each device reclaims any previously acquired isochronous resources from the IRM. A 1394 network that supports application layer resource management may also contain legacy devices which follow the above procedures in the event of a bus reset. Furthermore, our approach to management must be resilient to faults.

1.3 Focus and Approach

Our goal was to examine the performance of individual applications and the utilization of the overall network while under the direction of a resource management system which is based on Khan's Utility Model [8]. The steps we took to design and implement this management system are as follows:

- The first phase was to understand the current approach used for resource management within a 1394 network, and to determine its deficiencies. Although the standard method for resource acquisition was adequate for small 1394 networks, we

believed it to be unsuitable for larger topologies;

- The second phase involved defining the scope of what we required to manage. We have focused on the resources which are shared amongst the nodes on the network - specifically, isochronous channels and bandwidth. Resources which are local to a 1394 device, such as buffer space, also require prudent management, however, they are not within the scope of the our scheme;
- The third phase was to determine a mathematical model which could allocate multiple resources amongst applications which support scalable sessions. We adopted the Utility Model for its completeness in addressing these issues;
- The fourth phase was to design a distributed management architecture, where a single server-side resource management component receives requests from several client components; in return, the management component replies with allocations for each client. The design for client-side architecture includes support for the scalability of sessions and their individual streams, therefore, being able to adapt to their individual allocations. The Utility Model addresses resource management at the highest level - that of the specific management policy. However, it does not address implementation specific issues. Our architecture had to resolve such issues as the allocation protocol between server and client side components, media specific resource mappings, fault-tolerance, and interfacing with 1394's native resource management.
- And finally, the fifth phase involved the modeling of a 1394 network with our added support for resource management. This was accomplished by implementing a general-purpose model of a 1394 node in the OPNET Modeler simulation environment. The server and client side management components were then implemented based on the 1394 device driver API that is supported by the 1394 node model. With the model in place, we could easily measure: the performance of our reservation protocol, the performance of the individual sessions, and the utilization of the whole network under a variety of loads.

1.4 Outline

The remainder of the thesis is organized as follows:

- Chapter 2 presents an overview of the IEEE 1394 Serial Bus Standard.
- Chapter 3 describes the mathematical model for resource management and the heuristic solution as defined in Khan's Utility Model.
- Chapter 4 presents the architecture for application level resource management and describes how the concepts within the Utility Model were applied.
- Chapter 5 describes the protocols used by our application level management architecture.
- Chapter 6 describes the implementation of the 1394 OPNET model.
- Chapter 7 describes the implementation of our resource management architecture.
- Chapter 8 gives simulation results for our management architecture.
- Finally, Chapter 9 summarizes our contributions and gives directions for future work.

2 Overview of IEEE 1394 Serial Bus

Home entertainment devices have traditionally been inter-connected by a variety of analog cables. For instance, the coaxial cable carries both audio and video signals between the VCR and TV. RCA cables interconnect stereo components. And workstations can be connected to these devices with limited success. The inter-connectivity of these devices is awkward and often impossible, and therefore prohibitive to many potential collaborations.

A single unified network has obvious advantages. Digital versions of the VCR, TV, and stereo could coexist and cooperate over a single digital network which offers a uniform method of physical connectivity and support for the transport of digitized multimedia data. However, such a digital network must be engineered to accommodate the high bandwidths and strict timing constraints of continuous media sources. For instance, when processing digital video, the on-screen rendering of video frames yields the perceived effect of motion. This effect is dependent on the reception and processing of the video data within a limited time period - miss the deadline and the video may appear jittery and unsynchronized. Therefore, there is a requirement for the reception of frames at the same (i.e., *iso*) time (i.e., *chronous*); the streaming of continuous media demands *isochronous* transport.

Network support for high-bandwidth multimedia data must address issues of delay, throughput, and delay variation. Moreover, the high-bandwidth traffic cannot completely saturate the network to the extent that other asynchronous messages are obstructed. Command and control messaging which supports the setup and control of isochronous flows can occur at any (i.e., *asyn*) time (i.e., *chronous*), hence, the requirement for

asynchronous services for inter-device control messaging.

The IEEE 1394-1995 Serial Bus Standard [5][6] fulfils the requirements for both high-bandwidth real-time sources and spurious traffic. Real-time data is supported by isochronous services which guarantee a timeslice of a predetermined length every 125 μs. A fraction of the bandwidth capacity is reserved for the asynchronous services. Therefore, the 1394 Standard is well suited to the interconnectivity of Consumer electronics (CE) devices (e.g., DVD, D-VCR, DTV) since the single medium can support both the data and control.

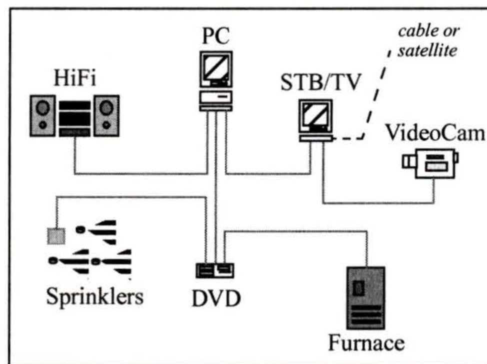


Figure 2.1: IEEE 1394 interconnectivity throughout the home

Figure 2.1 illustrates the potential applications for the 1394 serial bus. For instance, devices like the DVD player, the set-top-box (STB), and the Videocam are sources of high-bandwidth multimedia data, whereas, the PC, sprinklers, furnace, and the like exchange a sparse volume of control data. Although, the two sets of devices have dissimilar requirements for networking, the convenient and inexpensive connectivity provided by IEEE 1394 provides a practical solution for both types of applications. In this chapter we present IEEE 1394 and its related protocols.

2.1 Functional Overview

The IEEE 1394 standard defines an inexpensive and user-friendly network architecture based on the IEEE Std 1212-1991, Command and Status Register (CSR) standard [4]. Support for guaranteed bandwidth is 1394's main strength.

Originally intended as a replacement for several different buses used by the Macintosh

computer, an engineering team at Apple provided the fundamental design work on Firewire. This provided a basis for the IEEE 1394 standardization effort [18].

Available implementations currently include 100, 200 and 400 Mbps data rates. However, this bandwidth is limited to a short geographic span. A series of nodes can be daisy chained together with cables up to 4.5 meters in length, resulting in a maximum physical length of 72 meters, and 16 hops. A single bus can accommodate 64 nodes and the standard provides for an address space which accommodates up to 1024 separate buses.

2.1.1 Node Identification and Bus Configuration

The IEEE 1394 Standard specifies a self-configuring network which is suited to a home environment. Consumer Electronics (CE) devices with 1394 support can be easily interconnected to form a tree; cycles are not permitted. Each node of the network is assigned a unique identifier (ID) which is determined during the bus reset phase.

Users can take advantage of the hot-plug-and-play feature; new devices can be added or removed from the bus while other devices are actively operating. Such a topology change triggers a bus reset period, during which the topology and configuration of the bus is determined and advertised to other nodes. After this reset phase, each node is identifiable by a dynamically generated node ID. Hence, nodes may be assigned a different ID after a topology change.

When sending asynchronous messages between nodes, the node ID is used as part of the destination address. The bus ID and the memory offset within the node make up the remaining parts of the address.

When sending isochronous traffic, Individual nodes must acquire one of 64 isochronous channels; a single host transmits to the channel, while other hosts listen, thus providing a uni-directional multicast service.

2.1.2 Support for Isochronous and Asynchronous Traffic

The 1394 architecture provides support for both asynchronous and isochronous messaging. Asynchronous messaging provides a one-to-one best-effort transport with re-

transmission. The isochronous service provides a guaranteed Quality of Service (QoS), one-to-many, unreliable transport. These channels are typically used for continuous media transmission. Arbitration mechanisms preserve isochronous guarantees and maintain a fair-share policy amongst asynchronous traffic sources. Isochronous transactions receive up to 80% of the usable bandwidth and asynchronous transactions receive the remaining 20%, although these proportions can be redefined. Bus activity is based on a 125 μs cycle. Within every cycle the isochronous transactions are first completed and then the remaining cycle time is allocated to asynchronous transactions.

2.1.3 Physical Connectivity and Bus Topology

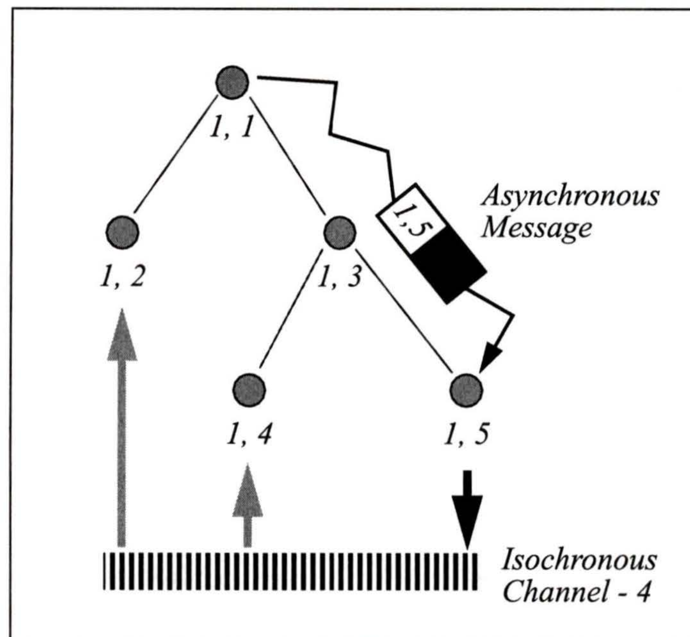


Figure 2.2: Topology and Services

Figure 2.2 above depicts the tree topology of a 1394 bus, and it also illustrates both the asynchronous and isochronous messaging services. An address *x,y* specifies the specific bus *x* and the node *y*. From the node *1,1* we see an asynchronous message originating and then propagating to node *1,5*. In fact, asynchronous messages are broadcast to all nodes, but only the destination node accepts the message. An example of isochronous messages is also given. The node *1,5* acquires and begins to transmit on channel 4. These messages

are also broadcast throughout the tree but only interested nodes, in this case nodes *1,2* and *1,4*, listen. And finally, it is interesting to note that the topology can support a mixture of link speeds: 100, 200, and 400 Mbps. A source can transmit at the highest rate which is also supported by its consumers and the intermediate nodes which repeat the transmissions.

2.2 Architectural Overview

The IEEE 1394 standard has been decomposed into several distinct functional units. In this section we describe the responsibilities of each unit and layer.

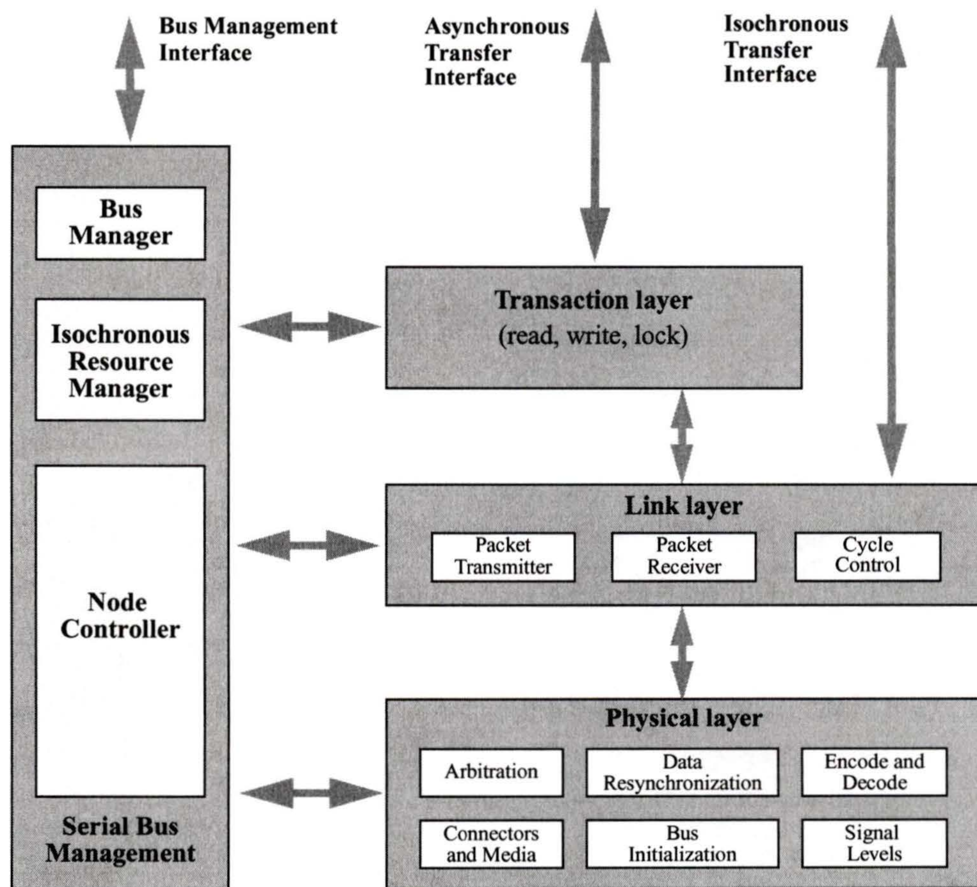


Figure 2.3: IEEE 1394 Protocol Layers

Figure 2.3 above illustrates the functional layers of the IEEE 1394 network architecture. The left side of the figure depicts the bus management components and the right side

shows the communications stack.

We start with the communications stack, whose main components are as follows:

- **Transaction Layer** - The Transaction layer is responsible for asynchronous services. The Control and Status Registers (CSR) Architecture as defined in IEEE 1212 is implemented at this level. Primarily, the transport of small messages or non-real-time blocks of data is accomplished. Addresses used in these transactions represent node addresses with a memory offset. In the event of loss or corruption, the Transaction layer supports re-transmission.
- **Link Layer** - The link layer is responsible for transmitting and receiving packets. It provides isochronous services to applications and also supports the Transaction layer. Isochronous packets are represented by a channel ID, not a node address, therefore providing a one-to-many (one sender and many receivers) communications model.
- **Physical Layer** - The physical layer specifies both the electrical and mechanical interface for transmission and reception. Issues of power-up initialization, bus-reset sensing, bus arbitration, and data signalling are addressed at the physical layer.

The 1394 architecture is based on several management components. The level of Bus Management offered depends on the functionality provided by the individual nodes. During bus reset, nodes which are capable of specific management tasks enter an election phase (see [6] for details). A 1394 bus can provide varying degrees of management, depending on the services which are implemented by the nodes on the bus. A node is chosen for each management task that is supported on the bus. However, because the election protocol is biased to the nodes which are closest to the Root node (see below), a single node commonly is elected for all the management services which it implements.

The 1394 specification identifies three components which support a completely managed bus - these are listed below:

- **Bus Manager** - The 1394 Bus Manager is an optional component. It is responsible for maintaining a map of bus topology, maintaining information about the speed capacities of individual nodes, bandwidth reservation for asynchronous transac-

tions, power management, and optimization of timing.

- **Isochronous Resource Manager** - The Isochronous Resource Manager (IRM) on the 1394 bus is optional. However, if absent, the bus cannot support isochronous transactions. The IRM holds information about resource availability, including channel number and bandwidth capabilities.
- **Cycle Master** - The cycle master is mandatory. It broadcasts a packet which indicates that a new cycle has started. Once this packet is received, nodes wishing to establish an isochronous transaction can send an arbitration request packet. The Cycle Master also provides a common time base for the entire bus. This is provided by a Cycle Synchronization signal every 125 μs.

At bus reset, nodes identify themselves based on their physical location on the bus. A *Root node* is initially elected by identifying the node which resides at the root of the tree topology. Primarily, the Root accepts arbitration requests and then signals the winner to start transmission. The outcome of the arbitration request is generally based on the distance which the requesting nodes are from the Root node; a closer node may win arbitration before others, but it cannot arbitrate again before other nodes have won access to the bus. It is not uncommon for all the management components to reside on the Root node.

2.3 The IEEE 1394 Address Space

The 1394 standard specifies an address space which is delimited by the individual bus, the node, and the address offset within the node.

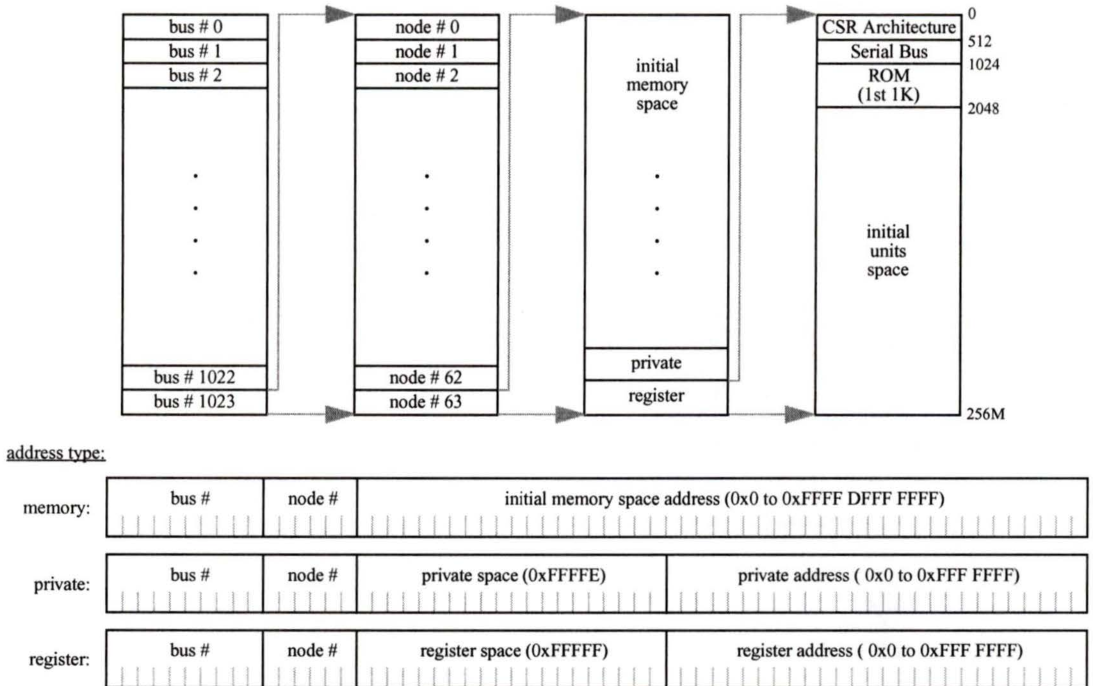


Figure 2.4: Serial Bus Address Space

Each asynchronous transaction operates on, or fetches operands residing at some portion of the address space. For instance, the *read* transaction fetches the contents of memory specified by a 64 bit address and a byte count. This address space is illustrated above in Figure 2.4. The bus and node identifiers are specified by 10 and 6 bits, respectively. We note that the node address 63 specifies a broadcast on the local bus. The address space or *memory offset* within a single node is specified by the remaining 48 bits.

The three address types are depicted in the above figure:

1. Memory - used by user applications;
2. Private - reserved for node's local use; and
3. Registers - contains registers which are part of the CSR architecture, as well as registers specific to the serial bus. The initial units space are reserved for node-specific resources.

2.4 Bus Reset and Configuration

The bus is reset immediately after a power-up or topology change. The reset provides a

means for individual devices to be identified and advertised. Additionally, services which facilitate the operation of the bus are elected from possibly several candidate nodes.

The phases of the bus reset and configuration process are as follows:

1. Bus Initialization - An active bus is supported by information which describes the current bus configuration. This includes information about the topology and resident nodes. During the bus initialization phase (also known as Bus Reset) a signal propagates throughout the bus, purging this information;
2. Tree Identification - A tree topology defines the parent-child relationships between the resident nodes. During the Tree Identification phase, a protocol establishes these relationships and ultimately identifies the Root node;
3. Self Identification - During this phase a protocol establishes the individual node IDs. Individual nodes broadcast their IDs, along with their speed and management capabilities to other resident nodes in a depth-first traversal of the topology. As the numbering scheme is based on the topology, node numbers are only valid until the next bus reset; and
4. Bus Management - Nodes which are capable of various management tasks (i.e., Cycle Master, Isochronous Resource Manager, and Bus Manager) are elected and made known to other nodes on the bus.

2.5 The Bus Arbitration Model

The principal property of the IEEE 1394 serial bus is its support for both realtime and non-realtime traffic. The isochronous streams are provided realtime guarantees while residual bandwidth is allocated to the asynchronous messages in a fair-share discipline. In this section we discuss the scheduling of these mixed traffic types over the bus.

2.5.1 Isochronous and Asynchronous Arbitration

Bandwidth for both isochronous traffic with guaranteed QoS, and asynchronous traffic with a fair-share performance must be allocated from the 125 μs cycle time. At the beginning of every cycle, the node which was chosen to be the cycle manager broadcasts a cycle_start packet. Upon receiving the cycle start message, a node with a isochronous

packet waiting to be sent enters into an arbitration phase along with other nodes that are waiting to transmit an isochronous packet.

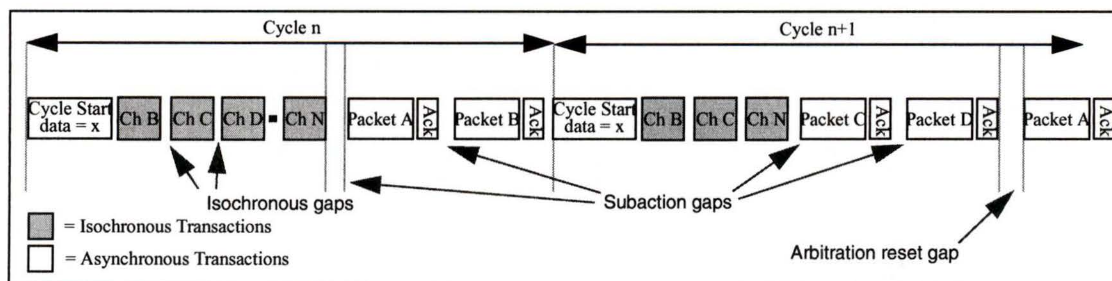


Figure 2.5: Cycle Timeline

The isochronous cycle is illustrated above in Figure 2.5. This depicts the transmission of isochronous and asynchronous transactions, and the regularity of the cycle length. At the left side of the figure a cycle start packet is initially transmitted by the Cycle Master, after which the nodes that are waiting to send an isochronous packet enter into an arbitration phase. The Root node grants the next isochronous transmission to the node whose arbitration signal was first detected. After a node has won arbitration and transmitted the packet, an idle time referred to as an *isochronous gap* occurs, after which, this arbitration and transmission process repeats itself until each node requesting transmission has been able to send exactly one packet. After the last node has sent its isochronous packet, a longer period of idle time, referred to as the *subaction gap*, occurs.

Upon detecting the subaction gap, nodes that wish to send asynchronous packets enter into a similar arbitration process. Upon winning arbitration, a node can transmit an asynchronous packet whose size is limited by a constant based on the transmission speed of the bus. After the packet is transmitted, another subaction gap occurs. Upon detecting this gap, other nodes can enter into arbitration for asynchronous transmission.

After all nodes have had the opportunity to send one asynchronous packet, an even longer idle time, referred to as the *arbitration reset gap*, occurs. This gap may occur before or after the next cycle start packet and the subsequent round of isochronous transactions. Once the arbitration reset gap has been detected, nodes can begin to arbitrate and send additional asynchronous packets. The period between arbitration reset gaps is

referred to as the *fairness interval.* When utilizing this round-robin policy, the fairness interval may span several cycles, depending on the size of asynchronous packets and the remaining time in the cycle during which asynchronous packets can be sent. Isochronous allocations are generally restricted to 80% of the entire bus bandwidth leaving the remaining 20% for asynchronous transactions.

2.5.2 Clock Skew

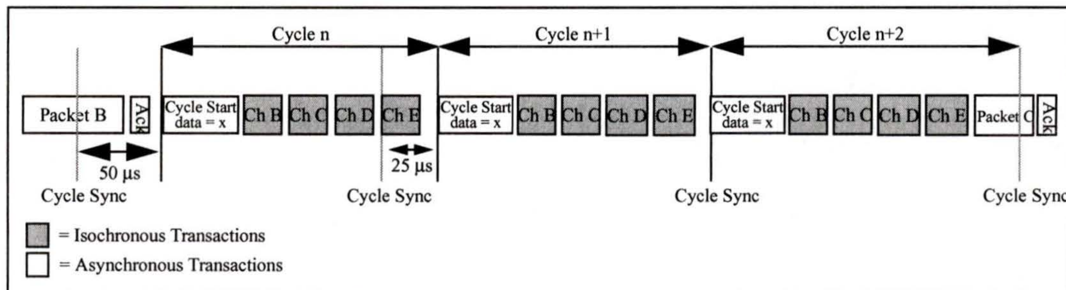


Figure 2.6: Cycle Start Skew

Although the speed of the bus limits the size of asynchronous packets as per the 1394 standard, a large packet could be transmitted shortly before the next Cycle Synchronization signal. Shown in Figure 2.6, the transmission of Packet B causes the next Cycle Start packet to be late by 50 µs. After the packet Cycle Start packet is broadcast, isochronous packets are sent, but Cycle n is still 25 µs late, therefore, no asynchronous packets are sent and the next Cycle Start packet is sent. After isochronous transactions are completed in Cycle n+1, there is no surplus time for the transmission of asynchronous packets, therefore, the Cycle Start packet is broadcast. Cycle n+2 is on time and asynchronous transactions are resumed. This recovery process ensures that the start of isochronous transactions is never be later than 65 µs [9] from the Cycle Synchronization signal.

2.6 Asynchronous Transactions

Traffic sources other than continuous media are supported by asynchronous transactions.

The defining features of the asynchronous protocol are those of reliability, variable payload size, and non-realtime response. Asynchronous packets carry signalling information for distributed multimedia systems, along with blocks of non-realtime data, and traffic generally associated with TCP/IP-based applications.

Unlike the fixed bandwidth guarantees afforded nodes with isochronous traffic, nodes with asynchronous traffic are given fair access to the bus. The access policy is not based on packet size, as is the case with isochronous transactions. Instead, during periods of contention, each node can transmit one asynchronous packet, therefore, granting media-access to contending nodes in a round-robin fashion.

A set of three basic transaction types fall into the asynchronous category: These transactions are:

- *Read* - A source node can obtain a data block from the target node's address space;
- *Write* - A source node can write a data block to the target node's address space;
- *Lock* - A source node can first read a data block from the target node's address space, and then use the lock operation to modify the data block from the previously read state.

All three operations rely on an addressing mode which identifies: the node, the bus on which the node resides, and a memory offset within the node. Therefore, an address is composed of the following tuple <*bus, node, offset*>; this is reflected above in Figure 2.4. Asynchronous transactions provide a mechanism for reading or modifying blocks of memory that are addressed using the tuple shown above. The addressing components are as follows:

- *Bus* - The bus portion of the address consumes 10 bits and indentifies the bus on which the node resides. The 1394 Standard provides for muliple busses to be inter-connected by bridges;
- *Node* - The node portion of the address consumes 6 bits, therefore, allowing a maximum of 63 nodes per bus - node 63 has been reserved as a broadcast address;
- *Offset* - The offset portion of the address is also referred to as the *destination offset*. It consumes 48 bits and addresses a specific memory location within the node.

As an example, the *read* transaction allows the contents of a memory block to be returned

to the requesting node. Using the above addressing method, the request specifies the memory location of interest.

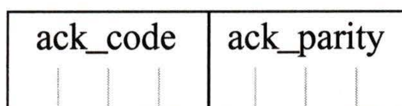


Figure 2.7: Acknowledgement Packet

Unlike isochronous packets, which have no guarantee of delivery, 1394 defines a mechanism for retransmitting lost, discarded, or corrupt asynchronous packets. This capability is enabled by the exchange of an Acknowledgement (ACK) packet (shown above in Figure 2.7) upon the reception of an asynchronous packet. ACK packets may not be received by the source node due to the loss of the original packet or the ACK packet. However, an asynchronous packet may also be discarded by the target node due to buffer overflow or a checksum error. In either case, within the ack_code field of the ACK packet, the relevant status code is returned by the target node. The accompanying ack_parity field provides a error-detection mechanism. As asynchronous packets do not include sequence numbers, the ACK packet is returned by the target immediately after the reception of the asynchronous packet.

The asynchronous protocol can be decomposed into two phases; each phase normally yields one asynchronous packet and one ACK packet. During the first phase, the source node assembles and sends a *request* packet which describes the desired operation with the associated addresses and data payload. For instance, a request packet for a read transaction would indicate the read transaction type, along with the address and size of the desired block of memory. Once received by the target node, the response phase of the transaction is performed. In the case of the read transaction, the requested data block is retrieved and placed within a response packet which is then returned to the source node.

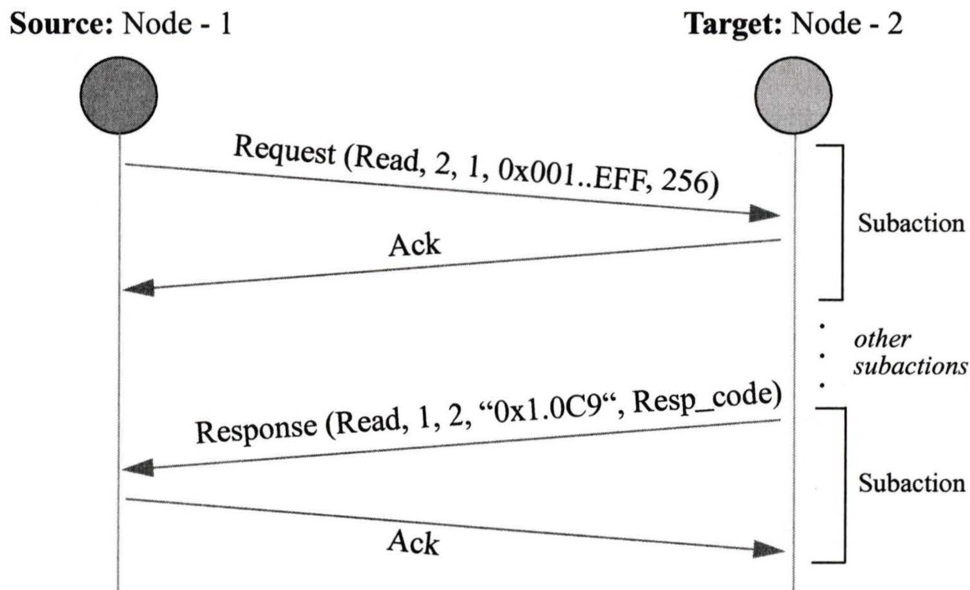


Figure 2.8: Request and Response Phases

The *split* transaction model is illustrated above in Figure 2.8. Here we see a request packet with several fields including: the read transaction type, the destination node address, the source node address, the destination offset, and the size of the data block.

An acknowledgement (ACK) packet is returned to a node upon receiving either a request or a response packet. The ACK packet indicates if the reception of the original packet was successful. Alternatively, the asynchronous packet may not be accepted due to a full buffer at the target node, an invalid destination offset, or data corruption; the ACK packet would indicate each of these error conditions. If an initial transmission yields no ACK packet in return, or the ACK packet indicates an error condition, the source node retransmits the original packet in a subsequent subaction.

Finally, the response packet is returned to the source node. Fields within the packet include: the read transaction type, the destination node address, the source node address, the requested data-block, and a response code which indicates the success of the operation.

For further reference, asynchronous packet formats are illustrated in Appendix A.

2.7 Isochronous Transactions

Traffic sources that are sensitive to delay and delay-variation are well suited to the isochronous services provided by 1394. A cycle time of 125 μs exists on the 1394 bus, during each cycle, a 1394 node is guaranteed a time-slot whose duration is defined prior to commencement of the traffic source.

In the 1394 communications model, a single traffic source is assigned an isochronous channel. A channel has one traffic source known as a *talker,* and multiple *listeners.* Therefore, all packets from a traffic source are carried over a single channel and nodes listen to that channel to receive the stream of packets.

Support for high-bandwidth multimedia streams and the ability to acquire bandwidth guarantees are the quintessential characteristics of 1394. By providing such guarantees for bandwidth, along with bounds on delay, and delay variation, 1394 is well suited for supporting consumer electronics devices which have traditionally relied on analog media for their interconnectivity.

2.7.1 Isochronous Packets and Channels

Media streams must be segmented and packetized to be transmitted across the 1394 bus. The 1394 isochronous packet encapsulates a segment of the stream data destined for a particular channel.

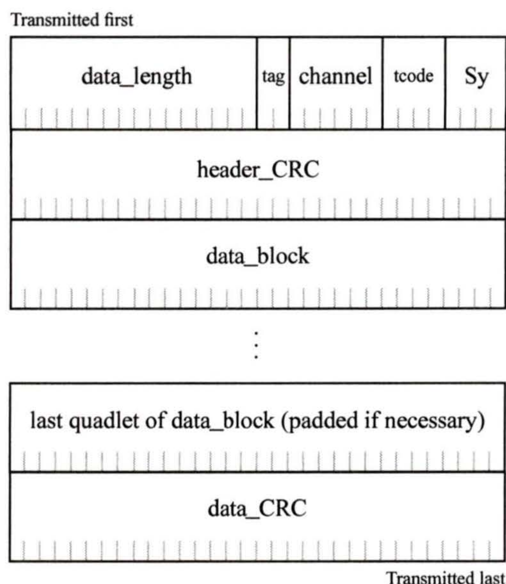


Figure 2.9: Isochronous Packet Format

An illustration of an isochronous packet is shown above in Figure 2.9. The channel field specifies the isochronous channel to which the packet is associated. The data_block fields contains the payload of the packet. When the packet is received from the bus, the link layer examines the channel field and either forwards the contents of the packet up to the application layer or simply discards it. The data_block field contains zero or more quadlets of data. Padding may be used if the payload data length is not a multiple of four bytes. An explanation of the individual fields follows in Table 2.1.

Table 2.1: Isochronous Packet Field Contents

Abbreviation	Field Name	Description
data_length	Data Length	The value can range from 0000h to FFFFh, indicating the number of bytes in the data_block. Possible values are multiples of 4, hence padding the last quadlet may be necessary.
tag	Data Format Tag	The value of 00b indicates that the data_block is unformatted. All other values are reserved.

Table 2.1: Isochronous Packet Field Contents

Abbreviation	Field Name	Description
channel	Channel Number	Value indicates the isochronous channel with which the packet is associated.
tcode	Transaction Code	The value for an isochronous packet is Ah.
Sy	Synchronization Code	The value is application specific.
header_CRC	Header CRC	The CRC value for the header.
data_block	Data Payload	Contains the data to be transferred.
data_CRC	Data Block CRC	The CRC value for the data block.

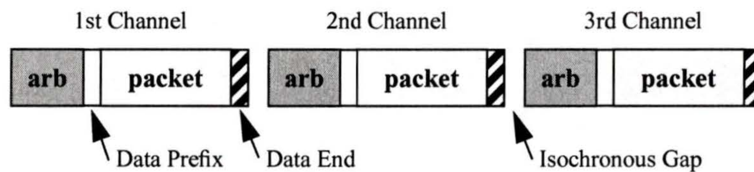


Figure 2.10: Non-concatenated Isochronous Transactions

The process for arbitration is explained in Section 2.5. We note that once arbitration for an isochronous transaction has been won, a node can transmit at most one packet for a specific channel. This illustration of bus activity is shown above in Figure 2.10. Here we see the arbitration phase followed by a data-prefix signal, and then the transmission of a packet. This is followed by a data-end signal and an isochronous gap. This sequence is repeated for other nodes which are ready to transmit a packet. Hence, at most one packet for each channel is transmitted per isochronous cycle.

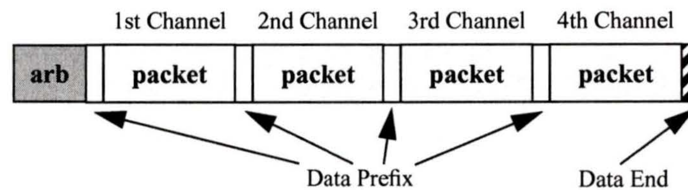


Figure 2.11: Concatenated Isochronous Transactions

If a node is a source for more than one isochronous channel, a node may yield several packets during a single cycle period. The process of concatenating isochronous transactions is illustrated above in Figure 2.11. After gaining access to the bus during the arbitration phase, a data-prefix signal is propagated across the bus, followed by the transmission of the first packet. If packets from other node-resident channels are waiting, the node immediately yields another data-prefix signal, followed by another isochronous packet. This process can repeat itself until a waiting packet from each channel is transmitted. After the last packet has been transmitted, a data-end signal is sent over the bus, after which, further transactions by other nodes can proceed.

2.7.2 Isochronous Resource Management

A node reserves resources on behalf of all the applications that it hosts. The resources include channel IDs, and bandwidth from the 1394 network, and local buffer space.

Channels IDs provide an addressing mechanism for a uni-directional multicast session. Each channel has only one "talker", therefore, the 64 available channels must be carefully allocated, Thereby preventing multiple sources over an identical channel. A register *channels_available*, hosted by the IRM, specifies the channels which have not yet been acquired. The register contains 4 quadlets, where each bit represents the utilization of each channel. A register *bandwidth_available*, also hosted by the IRM, reflects the amount of bandwidth which is not utilized.

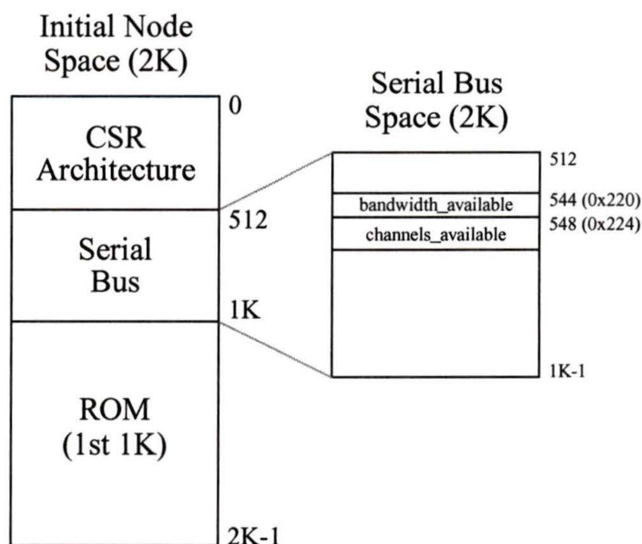


Figure 2.12: Location of Resource Registers in Memory Space

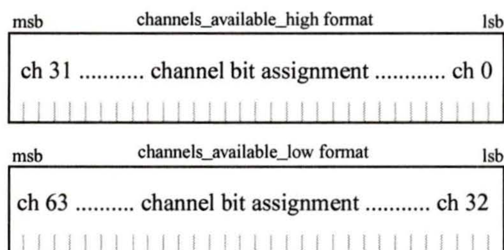


Figure 2.13: Channels Available Register

The *channels_available* register is shown above in Figure 2.13; its relative location within the memory space is illustrated above in Figure 2.12. The register resides at the offset 224h from the Serial-Bus dependent address space. All bits in the register are intialially set to ones. Bits which represent utilized channels are set to zero.

The register is accessible by each node using a combination of read and lock transactions. The read transaction provides the requesting node a list of the available channels, and the lock transaction allow the requesting node to update the register with a new value which reflects the channels which it has acquired.

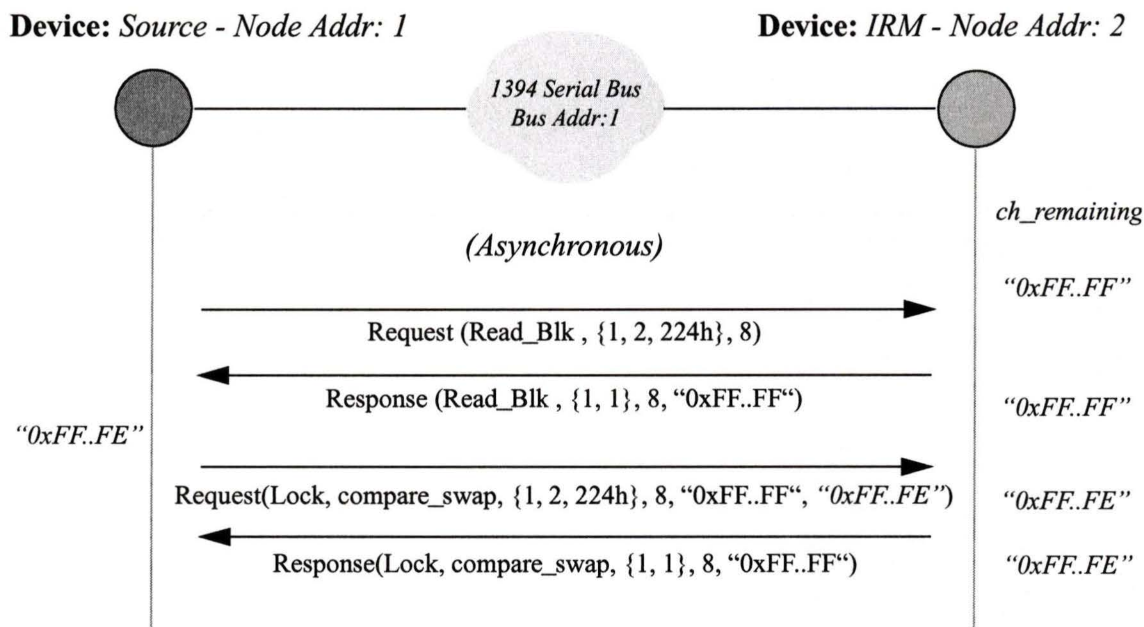
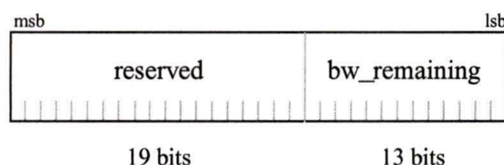


Figure 2.14: Acquisition of Isochronous Channels

The process of channel ID acquisition is shown above in Figure 2.14. The steps involved within this process are as follows:

1. Read the contents of the *ch_remaining* register - The source node uses a read transaction to fetch the contents of the register. The request indicates the {bus=1, node=2, and offset=224h} address of the register, along with the register size=8. If the read transaction is successful, the desired data is received in the response phase.
2. Calculate a new register value - Having retrieved the register value, the source node can flip a bit from 1 to 0 in the position which represents the desired channel.
3. Modify the contents of the *ch_remaining* register - The source node must now write the modified value back to the register. Because the register may have been updated by another node since the read performed in step 1, a lock transaction is used to detect such an intermittent update. The specific type of lock request used is known as the *compare-swap*. The request indicates the same target address {bus=1, node=2, and offset=224h} and register size=8. Both the original and updated register values are passed in the request. The old value is used by the target node to determine if an intermittent update has occurred. A response packet contains the pre-swap value of the register, therefore, indicating if an intermittent update was made. If the pre-swap value is not equal to the original value passed in the request, the IRM does not commit to the update and the transaction fails. If this transaction failure occurs, the source node recalculates a new value and then reattempt the lock-swap.



Maximum possible value = 6144 AUs
- at least 25 μs (1229 AUs) removed at start-up as for asynchronous transactions
- the remaining 100 μs (4915) is allocated to isochronous transactions

Figure 2.15: Bandwidth Available Register

The *bandwidth_available* register is shown above in Figure 2.15. The register resides at the offset 220h from the Serial-Bus dependent address space as illustrated in Figure 7.12. The length of time which a node has access to the bus, for the transmission of an isochronous packet, is measured by the Allocation Unit (AU). One AU is based on the amount of time (i.e., 20.345 ns) that is required to transmit one quadlet over a 1573.864 Mbps bus (commonly referred to as a S1600 bus).

The *bw_remaining* field of the bandwith_available register indicates the number of remaining allocation units. Upon restart, the register contains 6144[1] AUs, the maximum possible number of AUs. However, the Bus Manager is responsible for claiming a portion of the bus bandwidth for asynchronous transactions. Therefore, a minimum of 1229 AUs, or at least 20% of the total bus bandwidth is reserved.

---

1. Note: 20.345 ns * 6144 = 125μs

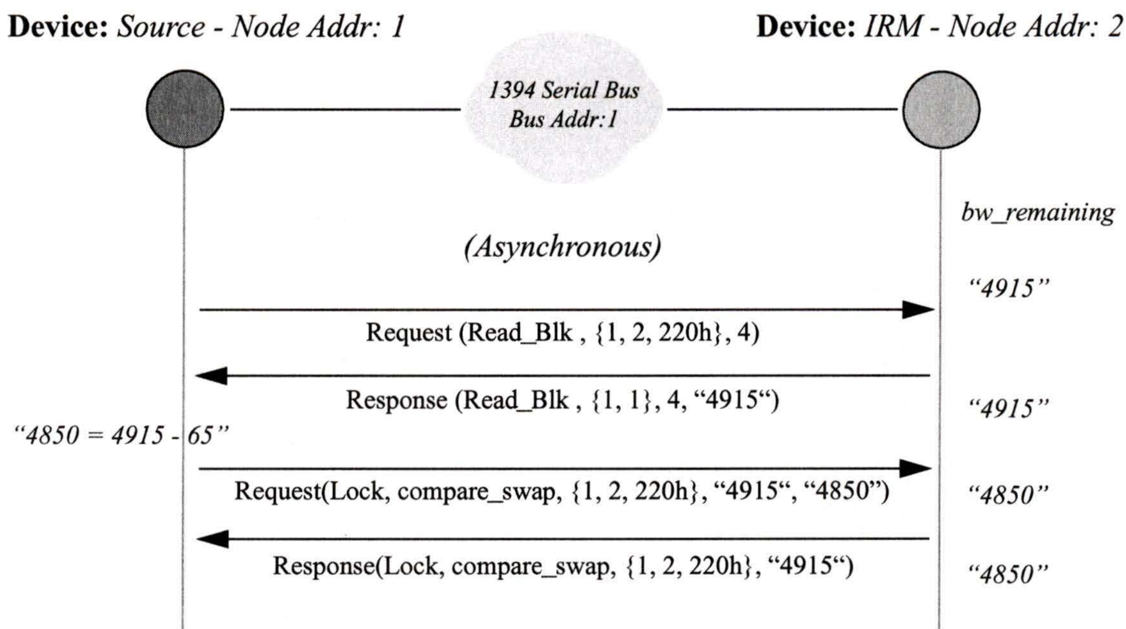


Figure 2.16: Acquisition of Bandwidth Allocation Units

The process for acquiring bandwidth is similar to the above procedure for channel allocation. The bandwidth allocation process is illustrated above in Figure 2.16. The steps are as follows:

1. Read the contents *bw_remaining* register - The source node uses a read transaction to fetch the contents of the register. The request indicates the {bus=1, node=2, and offset=220h} address of the register, along with the register size=4. If the read transaction is successful, the desired data is received in the response phase.
2. Calculate a new register value - Having retrieved the register value, the source node deducts its desired allocation in AUs, indicating the value of the remaining bandwidth.
3. Modify the contents of the *bw_remaining* register - The source node must now write the modified value back to the register. Because the register may have been updated by another node since the read performed in step 1, a lock transaction is used to detect such an intermittent update. The specific type of lock request used in this example is known as the *compare-swap*. The request indicates the same target address {bus=1, node=2, and offset=220h} and register size=4. Both the original and updated register values are passed in the request. The old value is used by the target node to determine if an intermittent update has occurred. A response packet contains the pre-swap value of the register, therefore, indicating if an intermittent update was made. If the pre-swap value is not equal to the original value passed in the request, the IRM does not commit to the update and the transaction fails. If this transaction failure occurs, the source node recalculates a new value and then reattempt the lock-swap.

3 The Utility Model

Our goal is to ensure the most efficient use of network bandwidth and channel identifiers when managing the limited resources of a 1394 Serial bus. We assume that a user's need for resources is represented by the users's willingness to pay for a specific QoS. We assume that "the most efficient use" means an allocation which maximizes a function called Utility, often defined as total revenue summed over all users. Given the requirements for a revenue based management system we must consider how to derive resource allocations based in the inputs of user requirements, charge rates, and the underlying resource availability. Furthermore, applications must be capable of adapting to the allocated resource level (possibly by refusing to run).

We have chosen the Utility Model [8] as a framework for making resource management decisions in an adaptive environment. Previously, Chen [7] applied the Utility Model to the problem of controlling layer-coded video sources over RTP/RTCP [14] within a local area network.

In this chapter we present the Utility Model: a framework which addresses our management requirements.

3.1 The Adaptive Multimedia System

The Utility Model addresses the problem of allocating multiple resources among multiple users who are running sessions which process and transport multiple types of media data.

Within an application, multiple media types such as audio, video, and graphics can be

composed and defined at several different qualities. Hence, a specific multimedia quality is composed of specific media qualities. For instance, a medium quality digital TV session may be composed of medium quality video and high quality audio along with a web-based overlay, whereas a low quality TV session could be composed of low quality video and medium quality audio and no overlay. The application defines each of these session qualities, also known as *operating qualities*, as shown in equation (3.1); in this instance a specific audio, video quality defines a session quality.

$$\mathbf{q_i} = (q_{ia}, q_{iv}) \tag{3.1}$$

A set of operating qualities for a specific session defines a *quality profile*, shown below in equation (3.2). Hence, the profile defines at each quality level the constituent media qualities for session *i*. In the profile below, q_{i1} and q_{in} would represent the lowest and highest qualities, respectively.

$$\mathbf{P_i} = (q_{i1}, q_{i2}, q_{i3}, \dots, q_{in}) \tag{3.2}$$

Each supported operating quality likely requires different amounts of resources, as each constituent media type has its own set of requirements. In equation (3.3), the resources CPU (c), memory (m), and network bandwidth (b) are defined in terms of the requirements for each media type.

$$\begin{aligned} c_i &= c(q_i) = c(q_{ia}) + c(q_{iv}) \\ m_i &= m(q_i) = m(q_{ia}) + m(q_{iv}) \\ b_i &= b(q_i) = b(q_{ia}) + b(q_{iv}) \end{aligned} \tag{3.3}$$

Hence, the sum of the individual resource requirements for each media type provide the resource requirements for the specific operating quality. For instance, low quality digital TV is composed of low quality video and medium quality audio; both media

correspond to several resource requirements.

Each operating quality must map to known resource requirements. This quality-resource mapping is defined in equation (3.4). Such a mapping is challenging to quantify due to the variability in requirements for media streams through time. For this discussion, we assume that the peak requirements are specified.

(3.4)

$$r_i = (c_i, m_i, b_i) = r(q_i)$$

Now that we have a profile of operating qualities (3.2) and a mapping from each quality to the resource requirements shown in equation (3.4), we now must consider how to allocate scarce resources while also maximising system utility.

Utility is a function of a specific operating quality. The definition of utility is based on the goals of the system and is the essence of the management policy. Utility may represent the revenue yielded for a specific operating quality of a session. Or utility may reflect the relative importance of the various operating qualities of the sessions supported by the multimedia system.

The system utility defined in equation (3.5) is simply the sum of all utilities, where each utility corresponds to a chosen operating quality for each active session within the system.

(3.5)

$$U = \sum_{i=1}^{n} u_i(q_i)$$

Revenue maximization is the common example of a utility objective. Alternatively, the system utility may reflect other policies for which maximization is appropriate. For instance, a weighted scheme may give preference to work-related applications, over those which provide entertainment. Or, it could give preference to sessions initiated by parents to those initiated by adolescent children, especially sessions containing Heavy Metal

content.

Table 3.1: Session Quality Profile with Resource and Utility Mappings

Aggregate Quality	Video Quality	Audio Quality	CPU Rqmt	RAM Rqmt	BW Rqmt	Utility
Gold: q_3	high: q_{3v}	high: q_{3a}	$c(q_{3a})+c(q_{3v})$	$m(q_{3a})+m(q_{3v})$	$b(q_{3a})+b(q_{3v})$	$u(q_3)$
Silver: q_2	medium: q_{2v}	high: q_{2a}	$c(q_{2a})+c(q_{2v})$	$m(q_{2a})+m(q_{2v})$	$b(q_{2a})+b(q_{2v})$	$u(q_2)$
Bronze: q1	low: q_{1v}	medium: q_{1a}	$c(q_{1a})+c(q_{1v})$	$m(q_{1a})+m(q_{1v})$	$b(q_{1a})+b(q_{1v})$	$u(q_1)$

In Table 3.1 above, a session profile is illustrated. In this example, the constituent media qualities are given, along with the resource requirements for each operating quality.

Based on the information in the above table, the goal of the management system is to select an operating quality for each session, such that the system utility is maximised while not exceeding the available resources. This constraint is shown below in equation (3.6).

$$\sum_{i=1}^{n} r(q_i) \le R \tag{3.6}$$

The resource vector **R** = (*C, M, B*) represents the resource capacity of the system. For our *n* sessions, the sum of the requirements for the chosen operating qualities must not exceed **R**. That is, we must not allocate more resources than we have. This is a necessary condition to guarantee a level of QoS.

3.2 The Adaptive Multimedia Problem

.

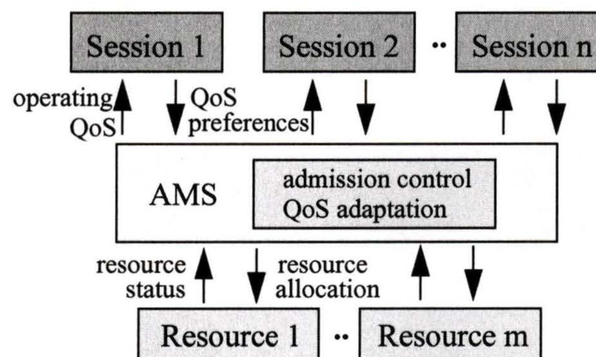


Figure 3.1: Adaptive Multimedia System Requirements

The issues and requirements of the adaptive multimedia system (AMS) are depicted above in Figure 3.1. Starting at the top, the individual sessions provide the AMS with information detailing the utility and resource requirements for each operating quality within their quality profile. Based on the request, the AMS determines if there are sufficient resources to admit the session. And in return, the AMS indicates to the session the operating quality which has been supported. The AMS may redistribute resources within the duration of a session, thus, indicating to the affected sessions their new operating qualities.

Given the above framework for defining the requirements and dynamics of multi-quality, multi-resource sessions, Khan [8] states the Adaptive Multimedia Problem (AMP) as follows:

1. Each user specifies a quality profile which defines a set of acceptable operating qualities for the session.
2. A session's operating qualities are assumed to be mapped to required resources.
3. The system is subject to the system resource constrains.
4. A session's operating qualities are assumed to be mapped uniquely to session utility.
5. The system utility is the sum of all session utilities.

The Utility model gives us a unified approach for admission control and quality adaptation. The definition of Utility also defines an admission control policy. Admission of a new session occurs if:

1. A solution exists for existing sessions and the session under consideration; and
2. The system utility including the new session is greater than the previous system utility.

3.3 Mapping the AMP to the MMKP

The goal is to find an allocation which maximises system utility. This problem involves multiple sessions, multiple operating qualities and multiple resource constraints. Khan mapped the AMP to a multi-dimensional, multiple-choice 0-1 knapsack problem (MMKP), a generalization of the classical 0-1 knapsack problem.

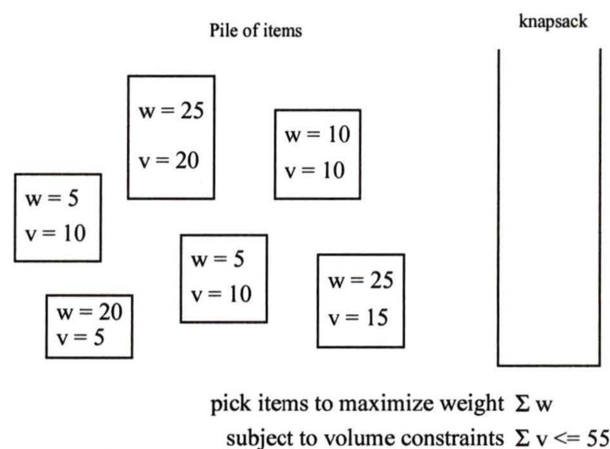


Figure 3.2: The Knapsack Problem

A simple example of the knapsack problem is illustrated above in Figure 3.2. The goal is to maximise the weight that is packed into the knapsack, while not exceeding the volume constraints of the knapsack. In this example, we are picking items from one pile, within the conditions of one constraint. The MMKP problem considers multiple constraints (i.e., multi-dimensional) and multiple piles (i.e., multi-choice), where only one item can be chosen from each pile.

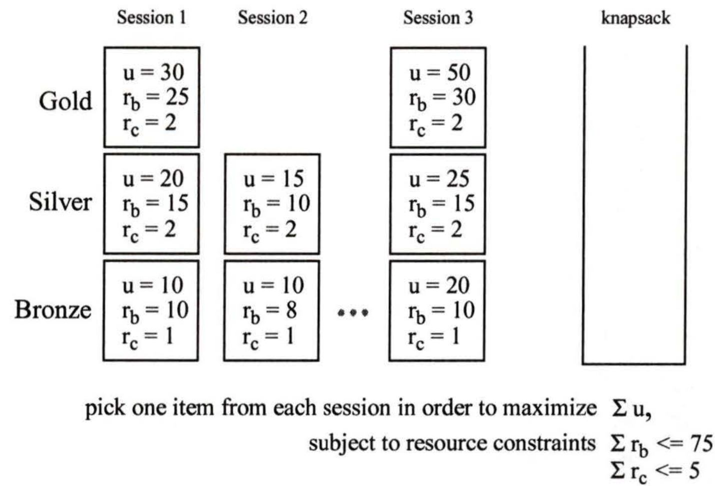


Figure 3.3: Mapping the AMP to the MMKP

The formulation of the AMP as an MMKP is illustrated above in Figure 3.3. For each session there exists one or more operating qualities. Each operating quality has a set of bandwidth and channel requirements (i.e., r_b, r_c) and a corresponding utility value (i.e., u). Hence, the goal of the MMKP algorithm is to select one operating quality from each sessions's quality profile, such that system utility is maximized while not violating the resource constraints. Note that each profile contains a null operating quality with zero resources required and zero utility. Selection of the null quality means rejecting the session.

3.4 The HEU Algorithm

In [8] Khan initially implemented the MMKP using a branch and bound algorithm, with the bound calculation based on a linear programming technique known as the Simplex Method. Although this approach provides an optimal solution to the MMKP, its exponential complexity made it an unacceptable choice, given the realtime constraints for managing a set of multimedia sessions. To address the realtime requirements of the management computation, Khan developed a heuristic (HEU) algorithm which provides

near-optimal solutions for the MMKP. Its complexity was shown to be $O(mn^2(l - 1)^2)$, for a system with n sessions, m resources, and l operating qualities for each session.

The behaviour of the HEU algorithm can be summarized as follows:

1. Each session is initially allocated its minimum operating quality.
2. An upgrade is allocated to the session which incurs the largest aggregate resource[1] savings.
3. If an aggregate resource savings is not achievable, the session upgrade that is selected incurs the largest increase in utility to the smallest increase in aggregate resource.
4. Steps 2 and 3 are repeated until: a resource constraint is violated, or until no further upgrades can be made.

3.4.1 A Utility Function for a Residential Environment

In the previous sections we outlined the issues which have been addressed by the Utility Model. The model's goal is the management of a finite set of resources while maximizing some function of the system's state, called the utility function.

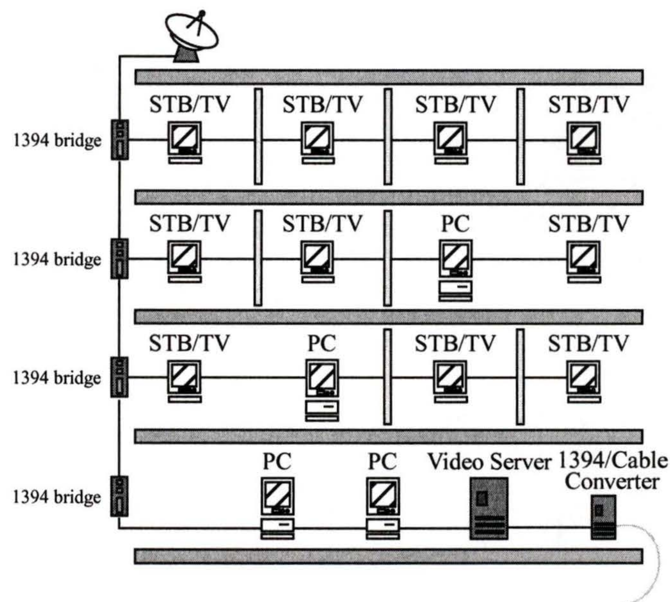


Figure 3.4: Residential IEEE 1394 Installation

1. Aggregate Resource [16] is a projection of the required resource vector onto the current resource usage vector. It provides a penalty for a item which has not yet been picked, based on current resource utilization. This penalty is greater if the candidate item (i.e., the upgrade) makes use of scarcer resources. Alternatively, candidate items which make use of lesser used resources incur a smaller penalty. The aggregate resource savings for a session is the difference between the aggregate resource of the current and upgrade operating qualities.

In Figure 3.4 above, the 1394 network is placed in a residential context. Whether in a hotel or condominium, 1394 can provide a scalable network which supports applications ranging from security systems to interactive TV and web browsing though a set-top-box (STB). The P1394.1 draft 0.08 [3] specifies a standard for bridging local buses, thus allowing the composition of larger networks. Such a network likely supports hundreds of broadcast channels which are available regardless of viewership. The residual capacity on the network can then be utilized on a pay-for-use basis. Hence, we assume a revenue maximization policy as depicted below in equation (3.7).

$$Rev = \sum_{i=1}^{n} u_i(q_i) \tag{3.7}$$

The heuristic selects a set of operating qualities $\{q_1, q_2, .. q_n\}$ such that *Rev* is maximized, while not violating the resource constraints.

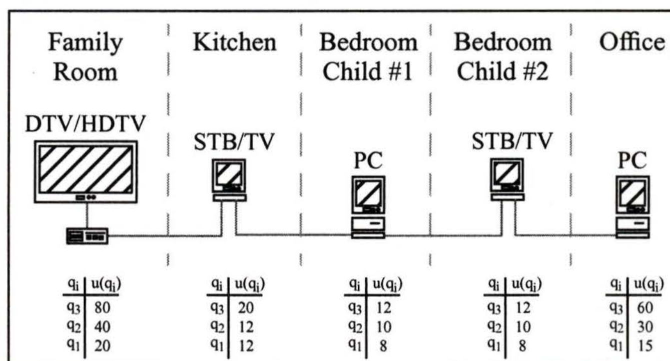


Figure 3.5: Using the Utility Model in a 1394 Home Installation

Our approach to resource management is also applicable to a networked home environment; a reality in the not so distant future. Devices throughout the home which yield or receive digital multimedia streams compete for bus resources. Although a profit motive may not be a relevant goal in allocating resources amongst the individual users within the home, we can still reuse the concepts of the Utility Model to ensure meaningful utilization. The parameters for management are the same, but their meanings are different.

A simple management scheme within the home environment requires a means to prioritize the relative importance of locations, users, or specific applications. In our example, illustrated above in Figure 3.5, such prioritization is based on the different locations throughout the home; a quality to utility mapping is shown for each location. When contention for bus resources occurs, the magnitude of each utility influences how the quality of the sessions are admitted, upgraded, or downgraded. However, since HEU considers the resource costs of a specific operating quality, utilities in this context are more that simple priority levels. Hence, sessions with lower resource requirements are given preference over other sessions with similar utilities and larger resource requirements.

4 Enhanced Management of the 1394 Serial Bus

Within the standard IEEE 1394 resource acquisition model, applications grab resources without regard to the requirements and priorities of other applications on the serial bus.

Our proposed contribution to resource management considers the requirements and priorities of other applications when rendering allocation decisions. We have defined a middleware, termed the Resource Management Client (RMC), which supports applications that require bus resources. A single node hosting a process, called the Resource Management Server (RMS), provides a centralized management component. In our management scheme, RMCs submit requests to the RMS and the RMS yields and distributes a set of allocations back to the participating RMCs.

In this chapter we consider: the resources which need to be managed, how they are managed, and how our enhanced management scheme coexists with legacy devices.

4.1 Application Resource Requirements

Sessions require access to local and shared resources before the transfer of media data can commence. Within the home environment, sessions carry out a variety of data transfers, and therefore impose a variety of resource requirements. For instance, the transfer of a 60 minute MPEG-2 stream from a DVD player to a DTV requires sufficient isochronous resources, and adequate buffer space at source and sink. Alternatively, the transfer of asynchronous data originating from interactive web activities also requires local buffer space, but access to the bus is regulated by asynchronous arbitration policies.

The aforementioned resources must be apportioned and allocated in a manner which conforms to our management goals. We now consider each of these resources and their inherent relationships.

4.1.1 Isochronous Resources

Time-sensitive data sources which require access to the bus at regular intervals are well suited to the isochronous services defined by IEEE 1394. Both bandwidth and channel identifiers must be acquired before such transmissions begins.

A quantity of isochronous bandwidth resource represents the maximum timeslice in which a node can access the bus during a single cycle of 125 μs. As described in Section 2.7.2, the node accesses the *bandwidth_available* register on the IRM node and then subtracts the required bandwidth. Bandwidth is measured in *allocation units;* one unit represents the amount of time in which one quadlet (4 bytes) can be transmitted over a 1572.864 Mbps bus (commonly referred to as a S1600 bus). As described in Section 2.7.2, the *bandwidth_available* register reports the unallocated units which are available for isochronous transactions. The sending node must acquire enough units such that sufficient isochronous data can be transmitted within a single timeslice during each cycle.

An isochronous channel represents a 1:n flow of isochronous data. The sending node must first access the *channels_available* register on the IRM node. As described in Section 2.7.2, the register is modified by the sending node, after which, the register specify which of the 64 channels identifiers have not been acquired. Once acquired, the source node tags outgoing isochronous packets with the acquired channel identifier.

4.1.2 Local Buffer Space

Applications which are directly accessing the isochronous services of an underlying driver [11] are responsible for provisioning adequate local buffer space. The buffer space required depends on the consumption and production rates of the listening and source applications, respectively. Since this memory is local to each node, it is not included in our management scheme.

4.2 Partitioning IEEE 1394 Resources

In the above section we considered the relationship between each resource and the supported applications. We now consider how IEEE 1394 resources can be partitioned between legacy devices and our enhanced management scheme.

4.2.1 Bandwidth

Normally, standard 1394 resident application directly acquires a quantity of bandwidth for its own purposes. Our management scheme assumes the management of a large proportion of the total bandwidth; however, it does not deduct the bandwidth allocation units from the IRM registers. Upon making a request to the RMS, an RMC receives permission to use some quantity of bandwidth; the application associated with the RMC then accesses the IRM and claim the prescribed amount.

Such a scheme is preferable to the RMS directly acquiring bandwidth from the IRM, and then allocating it to the RMCs. An application's additional accesses to the IRM are costly in terms of time, however, it prevents system inconsistencies. For instance, when transitions between operating qualities occur (i.e., adaptation), a temporary over-consumption of resources may result; this would violate the 1394 Standard and is therefore considered a system failure.

The distribution of bandwidth occurs when the bus is initially powered-on, or when the bus topology was changed; both conditions trigger a bus reset. At the bus reset, the Bus Manager sets aside some portion of bandwidth for asynchronous traffic. This desired amount can be directly deducted from the *bw_remaining* register. After which, at most 80% of the allocation units remain in the register. After the standard one second reset phase, an additional period of one second is provided for legacy devices to re-acquire their pre-reset amount of bandwidth.

For an additional third second, we assume that any new acquisitions by legacy devices are made, therefore, any residual bandwidth at the IRM can then be managed by the RMS. However, the RMS does not modify the contents of the IRM, it simply assumes that it has control over this bandwidth. This distribution of isochronous bandwidth is shown below in Figure 4.1.

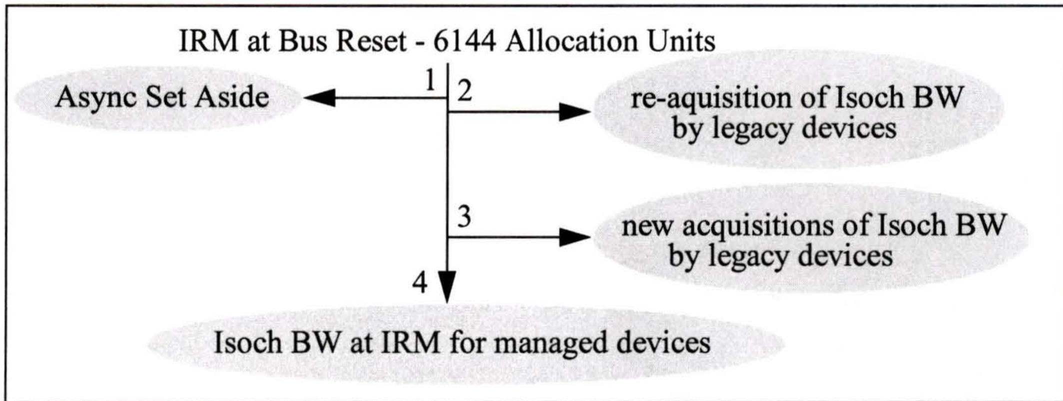


Figure 4.1: Distribution of Bandwidth

4.2.2 Channels

Channel identifiers must be shared between legacy nodes and those nodes supported by our enhanced management scheme. Therefore, the RMS assumes control over a portion of the identifiers from the *channels_remaining* register at the IRM.

At bus reset, the *channels_remaining* register specifies that all 64 channels are available. As with the re-acquisition bandwidth, during a one second reset phase which follows the reset phase, legacy nodes may re-acquire channels which had been acquired prior to the reset. As with the acquisition of bandwidth, we assume that legacy nodes make new acquisitions of channels during the third second after a bus reset. After completion of the reset phase and two additional seconds, the RMS assumes control of the residual channels at the IRM.

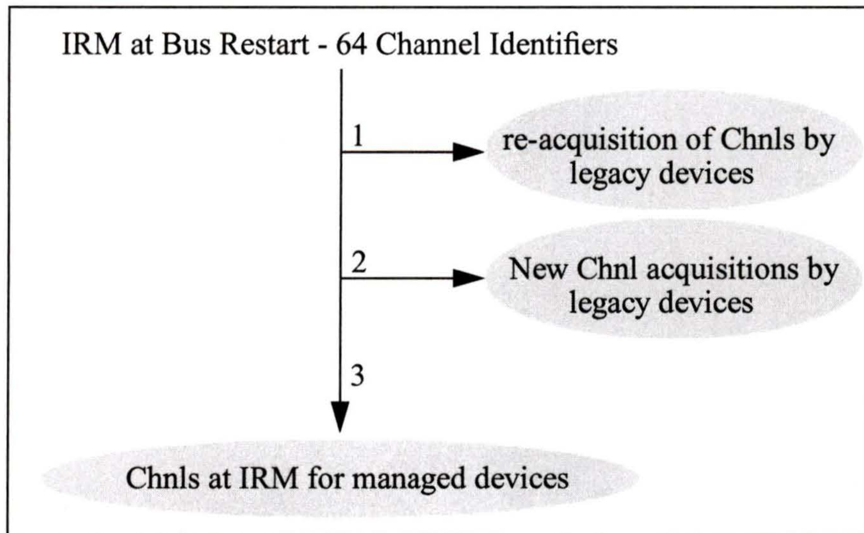


Figure 4.2: Distribution of Channel Identifiers

This distribution of the identifiers is shown above in Figure 4.2. Channel requirements can be specified by stating the number of channels needed. Requests for specific channel numbers, as required by some legacy devices such as the Sony DV camera, are possible through the standard channel acquisition methods. However, our management scheme does not accommodate specific hard-coded channel IDs; our scheme simply allocates a quantity of channel identifiers.

4.3 Reservation Semantics

The concepts of *operating quality* and *quality profile* were covered in Chapter 3. An application generates a quality profile that defines an ordered sequence of operating qualities. A *resource mapping* specifies the resource requirements for each of the operating qualities within the profile. A request from the application to its local RMC includes this resource mapping, along with parameters which are specific to the implemented management policy. The RMC forwards the mapping and other parameters to the RMS.

Table 4.1: Specification of Resource Requirements

Resource	Units
• Isochronous Bandwidth	• *i* Allocation Units
• Isochronous Channels	• *j* Channels

The resources we requests are listed above in Table 4.1. This includes bandwidth requirements, and the required number of channel IDs for isochronous transactions. We have not included local buffer requirements as they are not a resource which is shared amongst multiple nodes.

4.4 Resource Management Architecture

The resource management scheme we propose is based on the IEEE 1394-1995 Standard and is transparent to legacy devices which use standard resource acquisition methods. Our resource management architecture is based on a centralized server model. Multiple nodes, supported by Resource Management Client (RMC) middleware, interact with a single node which provides the services of the Resource Management Server (RMS). However, passive replicas of the RMS are supported to address the fallibility of the centralized model. The signalling protocol between the RMS and RMC components employs the underlying asynchronous transactions for the reliable transport of command and configuration information.

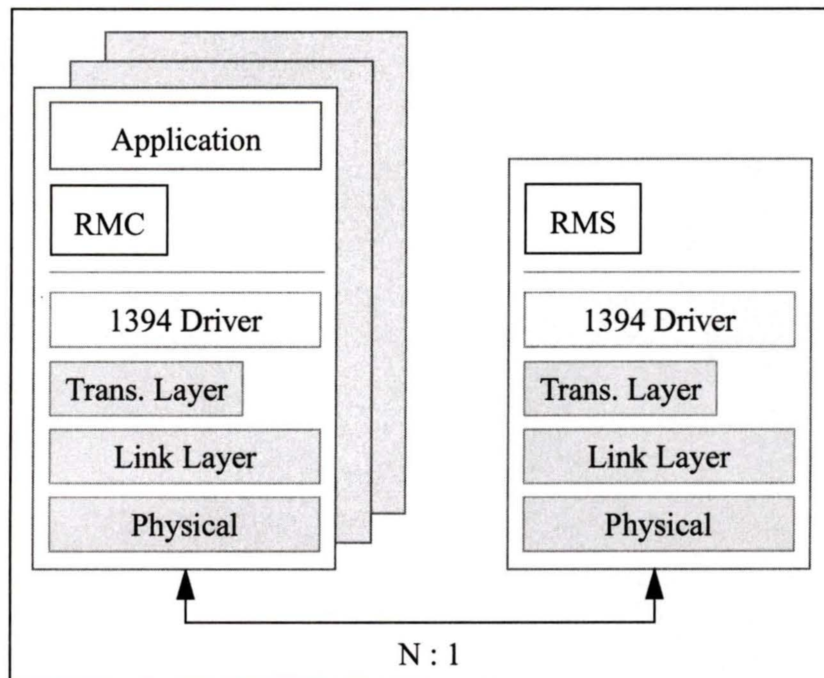


Figure 4.3: RMC and RMS Association

The topology of our management system is illustrated above in Figure 4.3. In this depiction we have separated the RMS and RMC components, however it is likely that RMC capable nodes are also RMS capable. As several RMS capable nodes are likely to exist on the same serial bus, we have defined an election protocol.

In the following two sections we present the architecture of both RMS and RMC components.

4.5 Resource Management Server

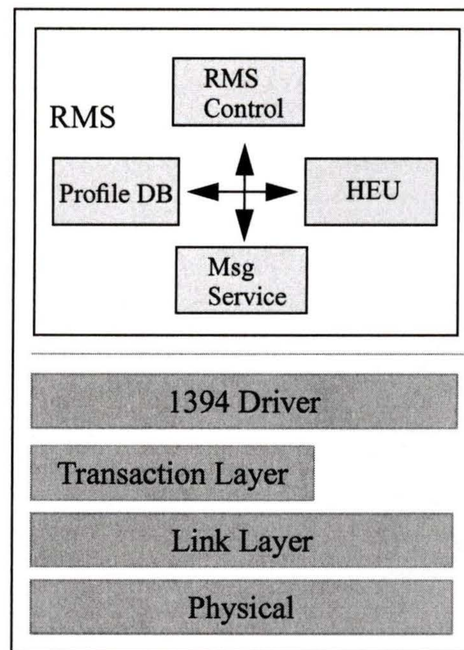


Figure 4.4: RMS Architecture

The RMS component relies on the underlying transaction layer to distribute and receive management data. Although the illustration shown above in Figure 4.4 only depicts the RMS component, the same node may also perform non-management roles, such as those which require isochronous transactions and therefore an RMC component. We now present the overall architecture of the RMS and the APIs associated with each module.

The RMS is composed of three main modules:

- *RMS Control* - orchestrates the steps in fulfilling requests.
- *MSG Service* - provides a source and sink for signalling messages, and controls the RMS election process. The RMS component relies on the MSG Service (MSGS) for the packaging and extraction of signalling messages from payloads passed over the serial bus. MSGS also facilitates the election process, thereby, determining the address of the elected RMS node.
- *Profile DB* - maintains a table of information about resource requirements and the

resulting resource allocations for all of its clients. The Profile DB module (PDB) for the RMS component maintains a single table of information including resource requirements and resulting allocations for all applications hosted by the RMCs within the network.

- *HEU* - implements the allocation policies within the system. The primary purpose of our management scheme is to allocate resources to multiple clients within the context of a specific allocation policy. The HEU process implements the heuristic [8] which was developed to obtain the optimal system utility for a set of quality-adaptive applications.

Further details about the design and operation of the RMS can be found in Appendix D.

4.6 Resource Management Client

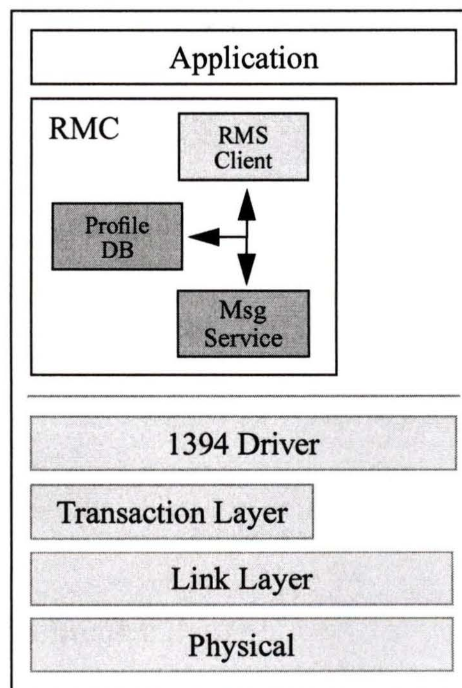


Figure 4.5: Resource Management Client

The RMC exchanges resource request messages and allocation messages over the underlying transaction layer. The RMC supports the requirements of the application by

accepting requests and forwarding them to the RMS node. Upon receiving an allocation, the RMC directs the behaviour of the application, hence controlling resource consumption.

A node which hosts an RMC component is likely be RMS capable, however, they operate in isolation of each other. Now the overall architecture of the RMC component and the APIs associated with each module is presented below.

The RMC is composed of two main modules:

- *RMC Client* - interfaces with the supported applications, and orchestrates the client-side requests and the resulting allocations.
- *MSG Service* - provides a source and sink for signalling messages, and listens to RMS election decisions. The Message Service (MSGS) module for the RMC component is similar in function to the MSGS within the RMS. The MSGS packs and extract the signalling messages from the payloads of control packets passed over the serial bus;
- *Profile DB* - maintains a table of information, where each record represents a local client and contains information about the resource requirements and the resulting allocation. The RMC component can support multiple clients. For instance, multiple applications and multiple sessions for each application may need to be managed. Profile information can be resubmitted to a newly elected RMS, after a bus reset, transparently from the supported clients.

Further details about the design and operation of the RMC can be found in Appendix C.

5 Enhanced Resource Management Protocols

Our enhanced resource management scheme provides several protocols which support resource allocation and the initial configuration of RMS and RMC modules. In this chapter we present these protocols and illustrate how they are supported by the underlying mechanisms of the IEEE 1394 bus.

5.1 RMS/RMC Start-up and Re-configuration

When the 1394 serial bus is initially powered on, a series of configuration steps occur during the start-up phase. These steps include: tree identification, self identification, and resource acquisition. These steps, part of a bus reset phase, are addressed in Section 2.4 of Chapter 2. After these standard configuration steps have concluded, our application level management components must be configured in preparation for the management services. Once configured, RMC components are free to submit requests.

If the bus topology is changed at some later time, a re-configuration phase occurs. Initially, a bus reset is signalled, and then the standard bus configuration steps are repeated. This is followed by the re-configuration of our management components.We now examine both of these scenarios.

5.1.1 RMS Initial Start-up Procedure

The individual phases which occur when the serial bus is initially started are illustrated below in Figure 5.1. The prerequisite phases for following phases are depicted by arrows

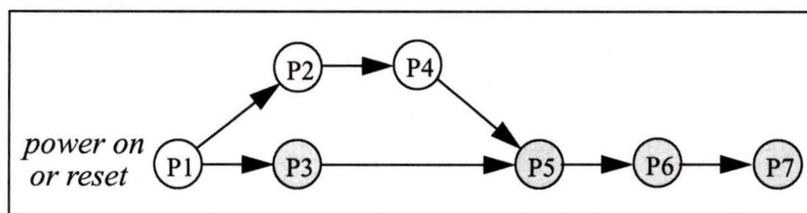


Figure 5.1: Serial Bus and RMS Setup Sequence

Each of the phases (the shaded phases are associated with our enhanced management) are defined as follows:

- Phase 1: Bus initialization and configuration - Upon bus power-up, a bus reset occurs, resulting in initialization and configuration. During the first 1000ms phase after a bus reset signal, the topology of the tree is identified, the root node is determined, and each node is assigned an address. Additionally, the bus manager and the isochronous resource manager are elected.
- Phase 2: Isochronous resources are re-acquired from the IRM - During the second 1000ms after a bus reset signal, legacy nodes may re-acquire resources from the IRM; these resources had originally been obtained prior to the bus reset.
- Phase 3: RMS election - This phase can occur concurrently with Phase 2. Multiple RMS candidates attempt to write their own node number in a well known register location on the node which hosts the IRM. The first node to succeed is elected to provide RMS services.
- Phase 4: Legacy nodes acquire new isochronous resources - We assume that legacy devices acquire new resources during the third second after a bus reset signal.
- Phase 5: RMS assumes control of the residual resources at the IRM - Once the election phase has completed and legacy nodes have acquired the necessary resources, the RMS component assumes control of the remaining resources. However, the RMS does not acquire the resources from the *bw_remaining* and *chnl_remaining* registers at the IRM; the specific managed nodes which make use of the resources acquire them directly. The RMS simply dictates the amounts of resources which can be acquired.

- Phase 6: RMCs begin requests - Once the RMS has been elected, RMC components can begin to submit allocation requests to the RMS. This process requires that the RMC specifies its resource requirements, along with other policy-specific information.
- Phase 7: RMS allocations are distributed - After acquiring bandwidth from the IRM and after receiving requests from RMC components, the RMS distributes allocations to RMC components.

5.1.2 RMS Restart Procedure

After a node has been removed or added to the bus, a bus reset is triggered resulting in the above initialization and configuration steps (shown in Figure 5.1) being repeated. Hence, the latency associated with restarting our management scheme is similar to the latency of the original start-up.

In the event of an RMS failure, a bus reset is triggered and the above steps are also repeated. We assume fail-stop behaviour for the node that provides the RMS services, along with fail-stop behaviour of the actual RMS services at that node.

5.2 The RMC to RMS Signalling Protocol

In this section we define several signalling protocols which support communication between the RMC and RMS modules. The first protocol provides a means for an RMS to be elected amongst several candidate nodes. The second protocol provides a means for RMCs to request resources and allows the RMS to convey the resource allocation levels to the individual nodes. The formats of the messages which support these protocols are presented in Appendix B, and the formats of the underlying asynchronous packets are illustrated in Appendix A.

5.2.1 RMS Election Protocol

Our management scheme relies on the selection of a single node to provide the services of the RMS module. After each instance of a bus reset, our election protocol is employed to select the next node which provides RMS services.

Our protocol relies on the underlying services of the *compare_swap* variation of the asynchronous lock operation, as defined by IEEE 1394. During the period of the election, each candidate node attempts to update, with its own node address, a known register location at the IRM node. The address of the IRM node is known by all nodes on the bus as a result of the self identification phase after the bus reset. The first node to accomplish this lock transaction wins the election. Other nodes, which subsequently attempt the update operation fail - a normal outcome of the compare_swap operation.

We are employing an unused quadlet, at 0xFFFF00234, in the CSR register space for our register. The format for this register is defined below in Figure 5.2.

```
typedef struct {
      USHORT     nNode;             /* RMS Node number*/
      USHORT     nGen;              /* Generation*/
} RMSNODE, *PRMSNODE;
```

Figure 5.2: RMS Node Register Format

An invalid node address within the register could be used to indicate that no candidate node has updated the register. However, because the register may reside on a legacy node,

we cannot assume that we would have the ability to initialize the register to 0x3F (the identifier for a broadcast address, therefore, an invalid address for a specific node on the bus) upon bus reset. However, we can assume that the register is initialized to zero upon bus reset.

Given that 0 is a valid node address, we use our RMS Node Register Format (illustrated above in Figure 5.2) to differentiate between a valid node address and an unmodified register. For instance, a register value of *nNode* = 0, and *nGen* = 0 would indicate that no election has yet occurred. A register value where 0 <= *nNode* < 62 and *nGen* = 1 would indicate that an election has occurred and *nNode* is the address of the elected RMS.

We illustrate a victorious election attempt by Node 1 in Figure 5.3 below. The candidate knows that the post-reset value of the IRM register is 0x0000000. It uses a compare_swap operation to update the value of the register from 0x0000000 to 0x00010001. These two values are passed as argument and data values within the lock request packet, respectively. In this example, the request specifies: the operation type, the destination address, the source address, the type of lock operation, the destination address for the register, the argument value, the data value, and the data size of both argument and data values. A full explanation of the fields within a lock request packet is given in Section A.2 of Appendix A.

At the destination, upon receiving the request, the node compares the current value of the destination offset with the argument data value. If they are equivalent, the data value is written to the destination offset. Otherwise, the value at destination offset remains unchanged.

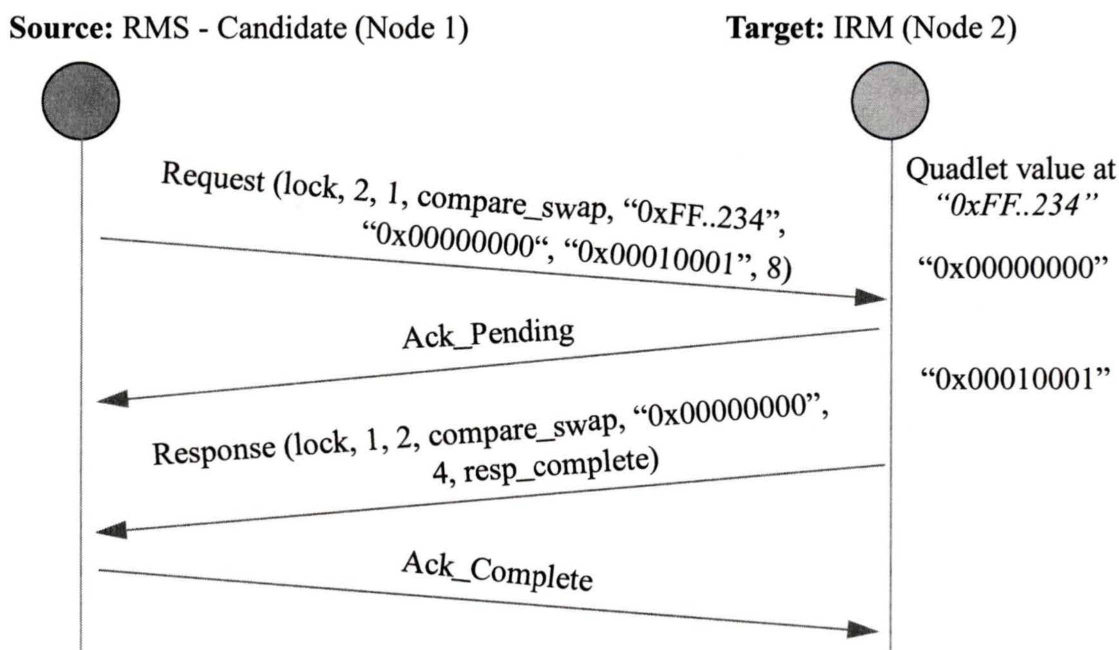


Figure 5.3: RMS Election Protocol: Election Victory

In either case, the value at the destination offset before the update attempt is returned in the lock response packet. In this example, the response specifies: the operation type, the destination address, the source address, the type of lock operation, the old data value, the data value, the size of the returned data value, and a response code indicating a completed transaction. A full explanation of the fields within a lock request packet are given in Section A.2 of Appendix 2.

Upon receiving the response, Node 1 can determine the success of the update by comparing the old value within the response packet, with the original argument value, as passed within the request packet. If they are equal, the update and the election was successful.

In Figure 5.4 below, we illustrate the case of a unsuccessful election attempt. In this example, the method is used to pass a request packet to the IRM node. The only difference is the data value, as it specifies a different candidate node - 3.

An Ack packet is passed back to the source node, followed by a response packet. Because the old value field in the response packet does not equal the argument field in the

request packet, we know that the register at the IRM was not updated, hence the election was lost.

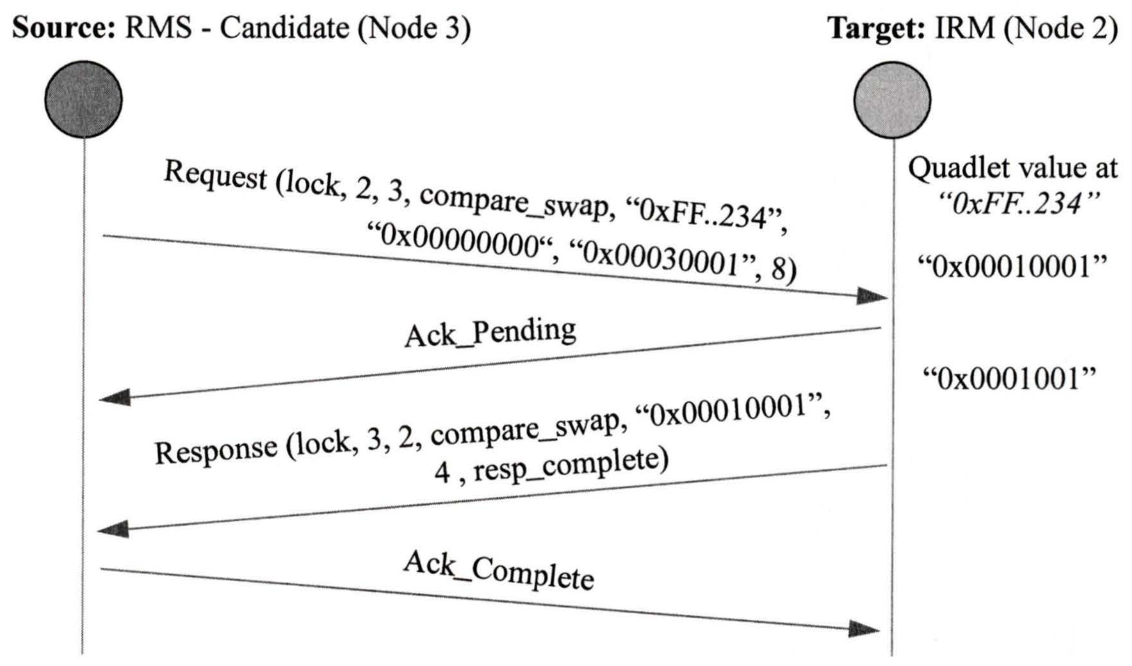


Figure 5.4: RMS Election Protocol: Election Defeat

5.2.2 RMS Address Acquisition

As outlined above, we have defined a way for an RMS node to be elected amongst multiple candidates. After the election, the resulting RMS node address must be known by the RMC modules on the bus. To acquire this address, the RMC module utilizes the underlying services of the asynchronous read operation, as defined by IEEE 1394.

The RMC module reads the register at the IRM node. If the resulting value (i.e., 0x00000000) indicates that the election has not yet been completed, a time-out occurs, followed by a re-attempt to read the register.

We illustrate this protocol in Figure 5.5 below. Node 4 attempts to acquire the RMS address from Node 2. In this example, the read request specifies: the operation type, the destination address, the source address, the destination address for the register, and the size[1] of the data block to be read. And finally, an Ack packet is returned which indicates the successful reception of the request packet at Node 2.

The second half of the read transaction now occurs; the value of the register is carried in the read response packet. In this example, the read request specifies: the operation type, the destination address, the source address, the requested data block, and the size of the data block to be read. A full explanation of the fields within read request and response packets is given in Section A.1 of Appendix A. And finally, an Ack packet is returned which indicates the successful reception of the response packet at Node 4.

In our example, the value of 0x00000000 is returned, therefore, the election has not yet been completed. After a second attempt, using the same read request/response framework, the RMC determines that the Node 1 has been elected as the RMS node.

---

1. Because a quadlet of data is being read, quadlet versions of both request and response packet formats are used. These are more efficient, as they no not include data_length or payload CRC fields - the quadlet payload is covered by the header CRC field. Both read and write transactions provide special packet formats for transferring a single quadlet of data. These quadlet formats are illustrated in Sections A.1 and A.3 of Appendix A.

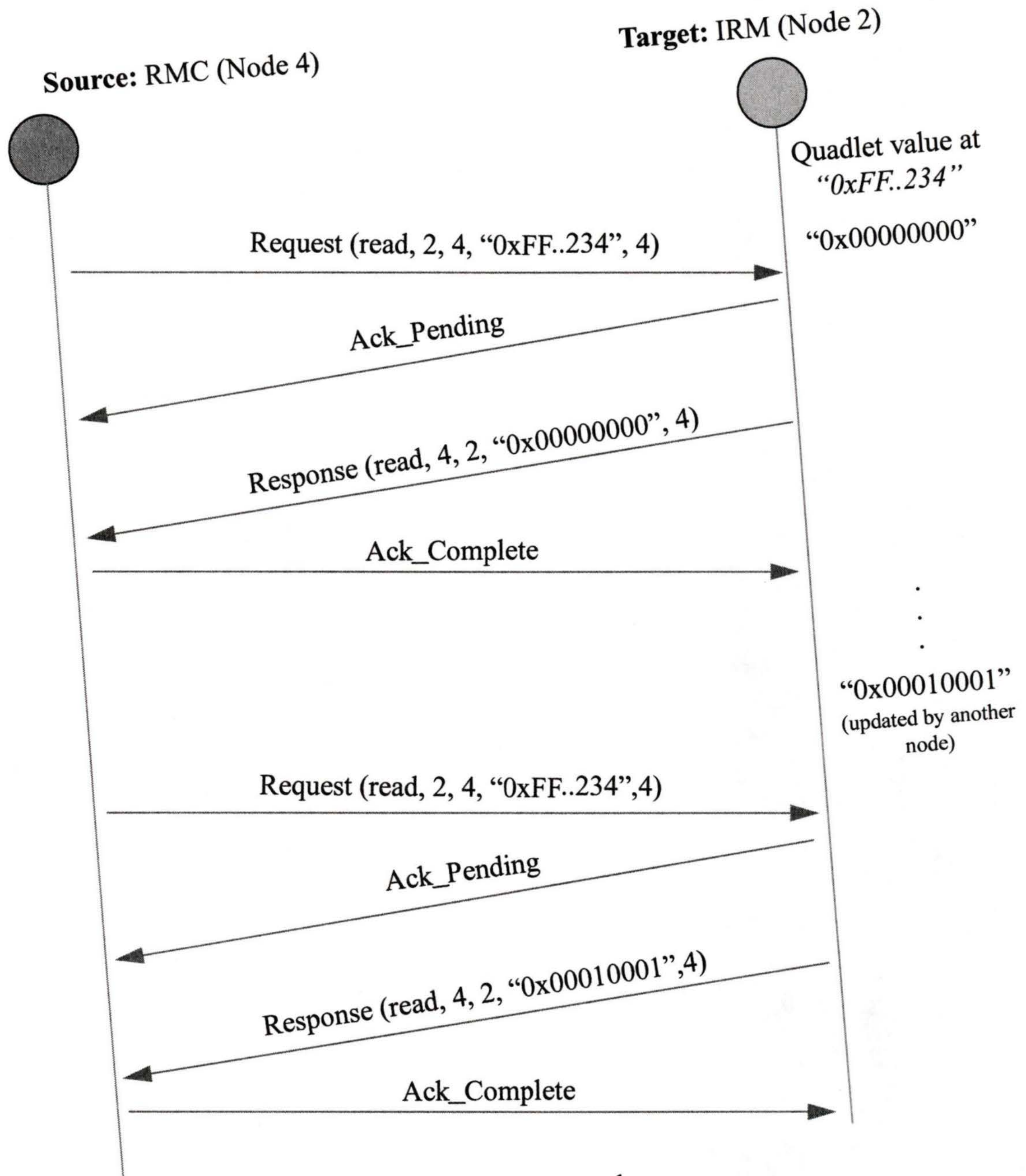


Figure 5.5: RMS Address Acquisition Protocol

5.3 Request and Allocation Protocol

The packetization and transport of resource request and allocation messages is provided by our resource management protocol. We rely on the underlying support of the transaction layer to send our messages between the nodes.

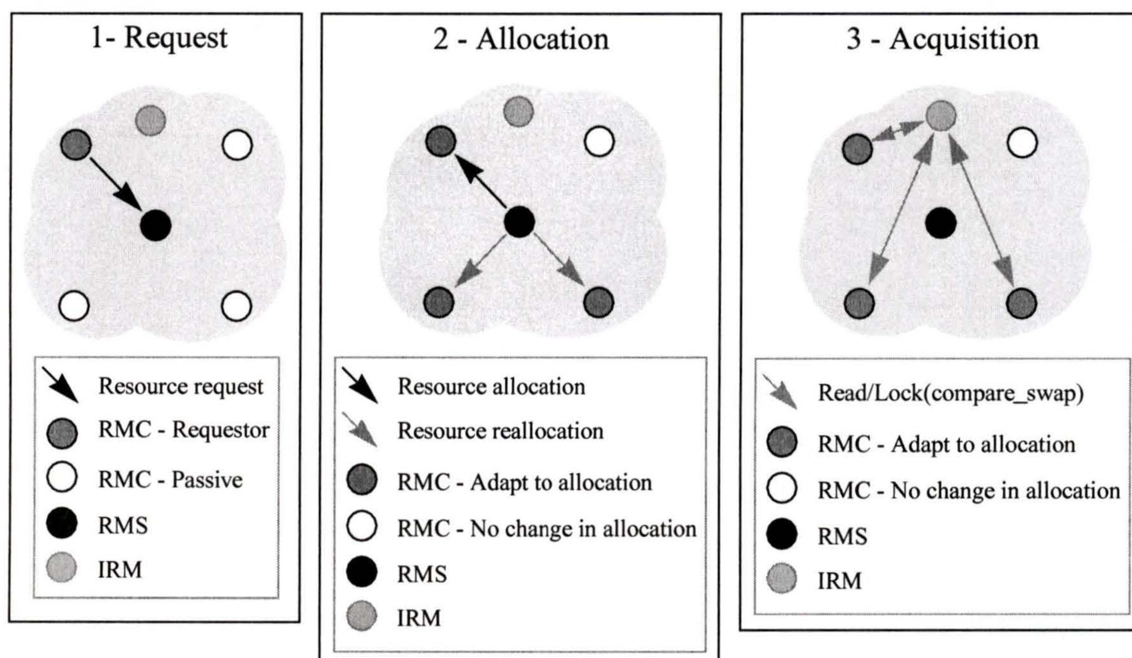


Figure 5.6: RMS and RMC - Request and Allocation Protocol Phases

Our protocol, as shown above in Figure 5.6, consists of two phases: in the first phase, the RMC component of the requesting node sends a request message to the RMS component; in the second phase, the RMS component sends an allocation message to the requesting node, and reallocation messages are sent to nodes whose current resource utilization must change.

A third phase is also illustrated. This phase is not part of our protocol; it is the IEEE 1394 Standard method for acquiring a quantity of resources. The applications that are supported by their local RMCs contact the IRM and acquire some quantity of resources as dictated in the (re)allocation messages.

The signalling between the RMC and RMS components occurs with the support of the reliable transport provided by the transaction layer. Our signalling messages generally

consist of allocation requests and allocation decisions from the RMC and RMS components, respectively. In both cases, the signalling data is sent within an asynchronous write operation, as defined by IEEE 1394.

Because we are sending multiple write messages in the allocation phase, a broadcast version of the write transaction could have been used, therefore, requiring only one subaction to disseminate the allocation information to all nodes. However, the broadcast version does not support acknowledgement. For reliability, we have chosen to send multiple write messages, where the reception of a block-write request packet is confirmed by the return of an ACK packet.

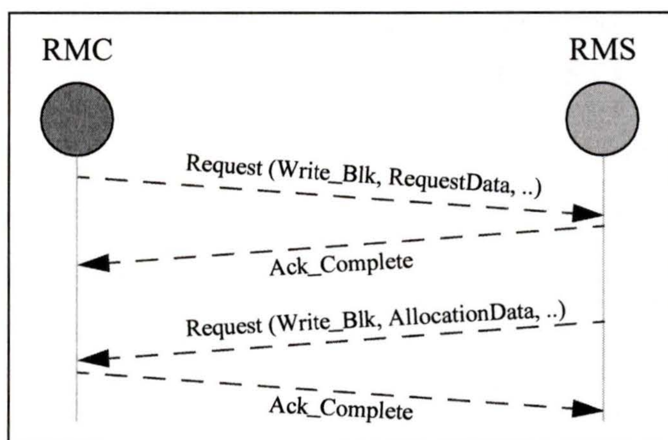


Figure 5.7: RMC Request - Successful Transaction

A request for resources is encapsulated in a write request packet and sent from the RMC to the RMS. If the write transaction was completed successfully, the transaction layer of the node which supports the RMS, does not return a response packet - success of the write transaction is conveyed in the Acknowledgement packet; a feature provided by the IEEE 1394 standard. The RMS then returns an allocation decision within a write request; similarly, no response packet is needed if the write transaction was successfully completed. Both phases are depicted above in Figure 5.7.

There are two sources of failure when transporting signalling information between the RMC and RMS components. For instance, a write request packet may become lost or corrupt. But, more likely, the request packet is not accepted by the responder node as a

result of limited buffer space. If the packet was lost, no ACK is returned by the responder. If the responder did not accept the packet due to corruption or limited buffer space, an ACK packet,which indicates the specific cause of rejection, is returned.

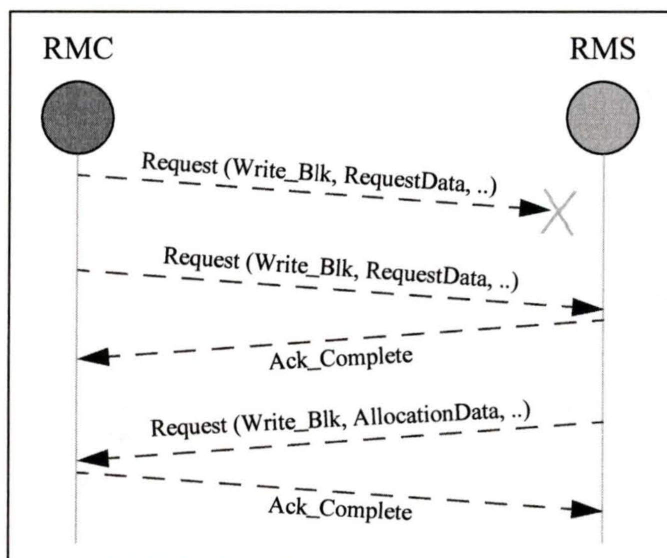


Figure 5.8: RMC Request - Lost Request Packet

A lost request packet is detected by a time-out at the requesting node; an ACK packet will not have been received from the responder within the sub-action. This case is depicted above in Figure 5.8. The request packet is resent until an ACK is received.

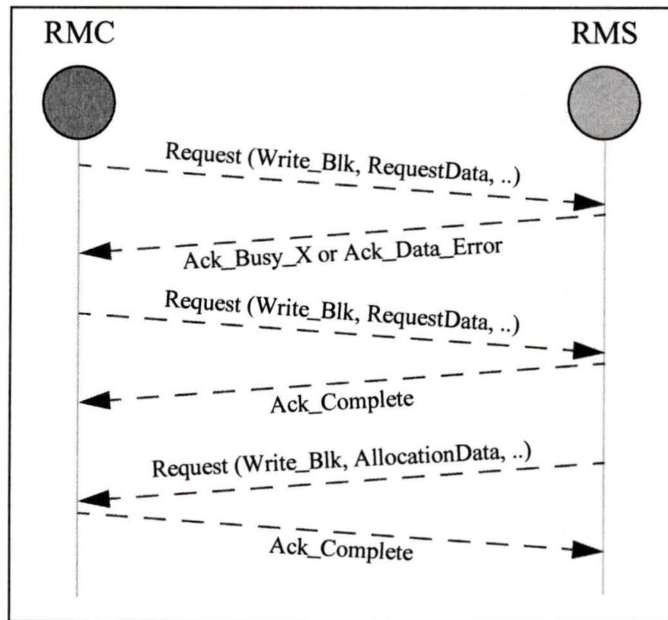


Figure 5.9: RMC Request - Rejected Request Packet

The write request may be received and rejected by the responder node. This case is shown above in Figure 5.9. The ACK packet which is returned to the requester indicates the specific fault. The request packet is then re-transmitted until an ACK is received.

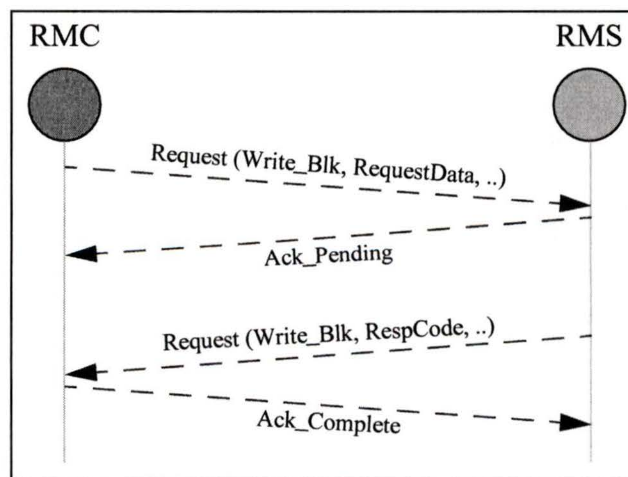


Figure 5.10: RMC Request - Failed Write Transaction

The above three error conditions simply involve problems with receiving the write request packet; these error conditions are handled by the link and transaction layers of requester and responder nodes. However, once the packet is received by the responder, the transaction request may be discarded at the 1394 application layer. For instance, if the write transaction is attempting to access a read-only area of memory, the transaction fails and a response packet describing the error condition is returned. Such access control is likely to be managed by the driver software on the responder machine.

Above in Figure 5.10, we illustrate this scenario. The request is received and accepted, and an acknowledgement packet is returned by the transaction layer of the RMS. However, the write transaction is not completed, and therefore, a response code indicated the error condition. From the perspective of the 1394 standard, this error condition was triggered by the application layer at the responder, and it must be handled by the application layer at the requester. Hence, our signalling protocol must be aware of such situations. Our protocol must define specific memory locations into which our messages can be written, hence avoiding this type of error from occurring.

In this chapter we have given an overview of the protocols which are necessary for application resource management over IEEE 1394. Further details of these protocols can be found in the Appendices. The data structures and message formats specific to our protocols can be found in Appendix B. And in Appendix A, illustrations are given of the IEEE 1394 packet formats which support our protocols.

6 IEEE 1394 Simulator

As a means of conducting a performance analysis of our resource management architecture, a simulation of a 3-port cable-environment[1] IEEE 1394 device was developed within the OPNET 7.0.B Modeler environment. Its design and implementation was a large proportion of the Thesis work and provides the only[2] comprehensive model of IEEE 1394 under the OPNET environment.

The model allows for a variety of serial-bus topologies to be arranged for the purposes of experimenting with higher-level protocols and applications which rely on an underlying IEEE 1394 architecture.

The 1394 device was developed using the fundamental tools provided by the OPNET environment. The bulk of our work involved the implementation of the single 1394 node which included the constituent layers as defined in the standard. Within the scope of this development, we relied on OPNET's support for: packet definition and handling, process definition and interrupt management, statistical collection, physical link definition, along with various other libraries of functions - called by the C programming language. The Network Editor provided by OPNET allows for a variety of topologies to be defined and simulated using our 3-port node. Other OPNET services provided for the management of multiple simulation runs, along with the collection and analysis of performance and related timing data.

1. The IEEE 1394 Standard specifies physical layers for both a backplane and cable topology. The cable environment is the format commonly used for consumer electronics devices.
2. A much simpler model by Walles [17] was implemented under OPNET to measure bus utilization.

In this chapter we explain the structure of our 1394 node model and we present the APIs for each of the constituent layers. We also give an overview of several processes within the model.

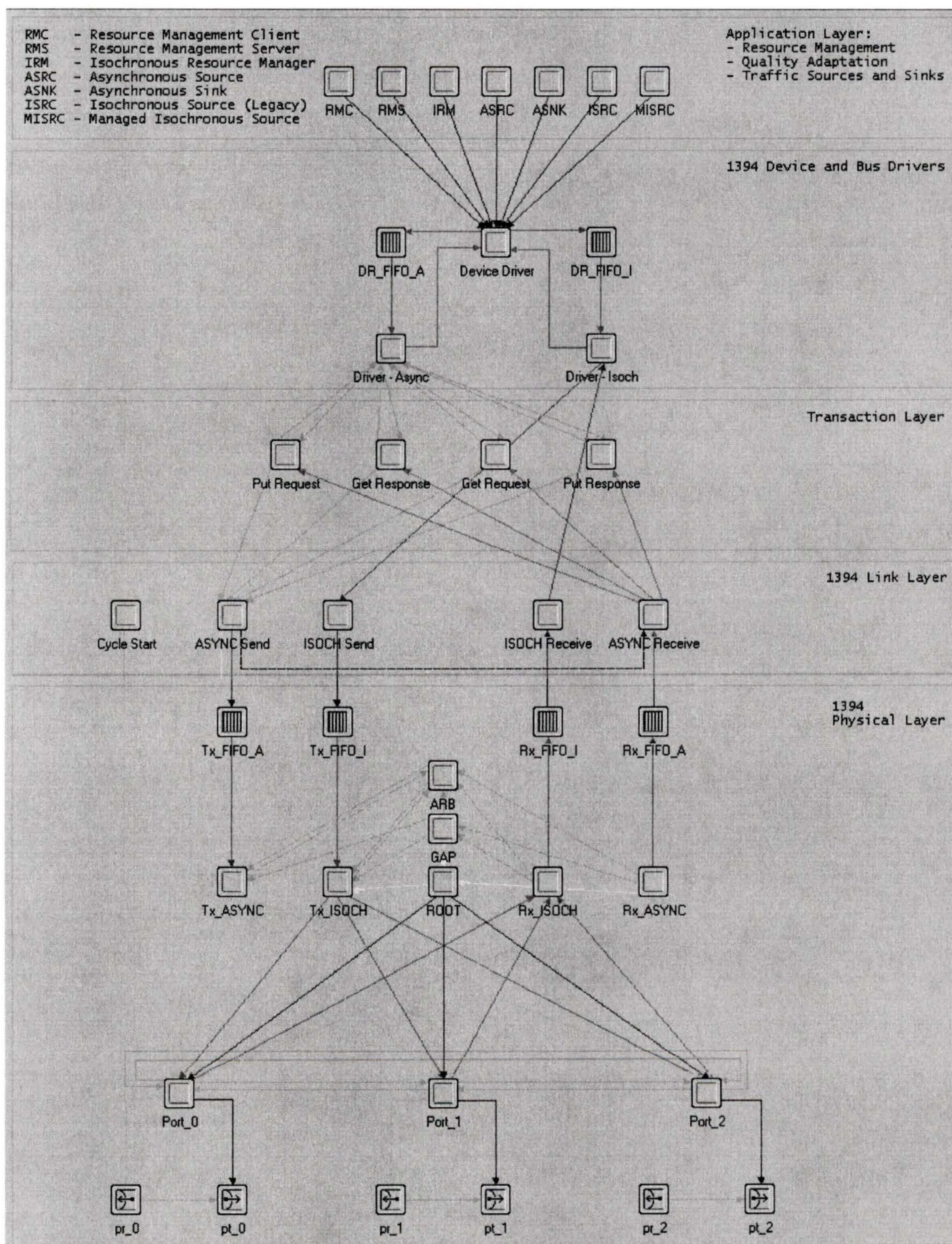


Figure 6.1: IEEE 1394 3-Port Node Model (Node Domain)

The 3-Port node model shown above in Figure 6.1 contains many inter-connected

processes, represented by the boxes. OPNET defines this view of the processes and their inter-connections as the *node domain*. Each process is defined by a finite state machine and related C code. This intra-process level of abstraction is termed the *process domain* (not shown).

Within the node domain, processes can communicate through two methods. The first method is the *packet stream*. These are illustrated by the solid unidirectional lines between processes. A process sends a packet onto the stream; the destination process then receives the packet. The second method is the *statistic wire*. They are similar in appearance to the packet stream, except for their dashed lines. Instead of carrying packets they pass integer data. This is particularly useful for passing control information between processes.

The square boxes at the bottom of the node model, labelled pr_*n* and pt_*n* are not processes; they are individual input/output ports defined by the OPNET package. These ports provide an interface between nodes within the *network domain* of the model. The 9 node network shown below in Figure 6.2 illustrates this domain.

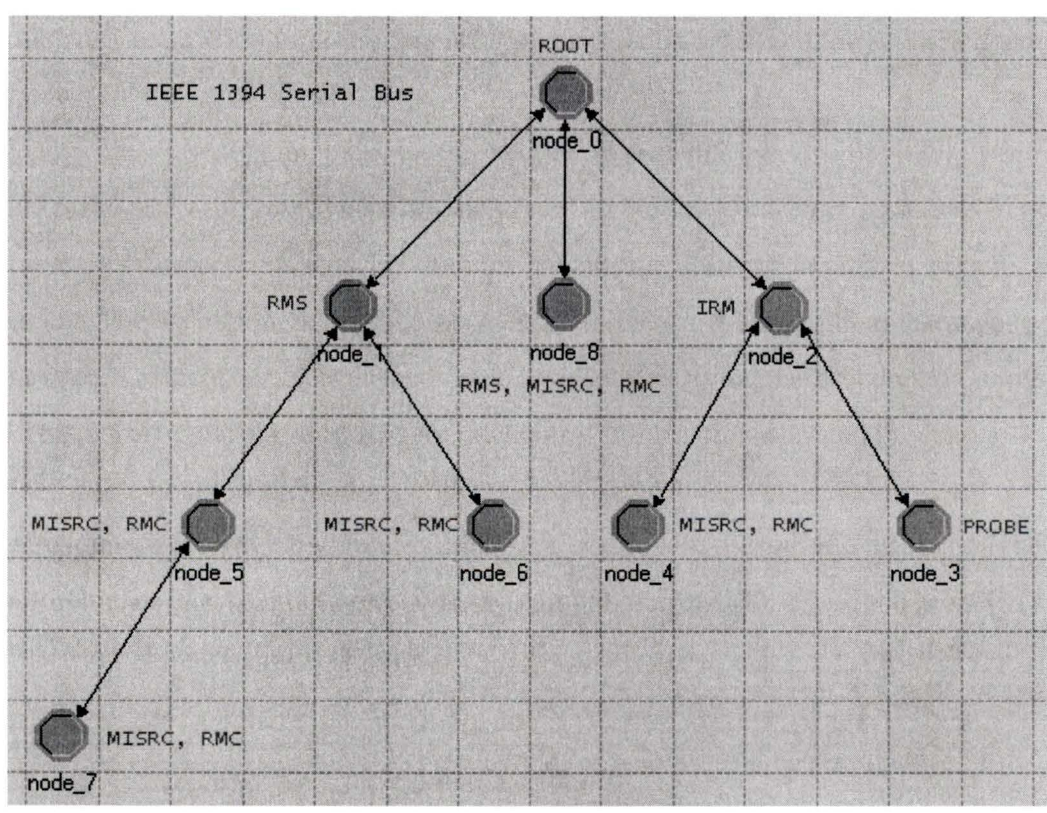


Figure 6.2: 1394 Model Network Model (Network Domain)

Our node model delineates the processes into each of the fundamental layers of the 1394 architecture. Each of these layers is illustrated in the following sections.

6.1 Physical Layer

Our implementation of the 1394 Physical layer provides a means for inter-node communication throughout our simulation. Specifically, our implementation of the physical layer provides the following functions:

- Automatic port and connection configuration;
- Cycle start management;
- Isochronous gap, subaction gap, and arbitration-reset gap detection;
- Isochronous and asynchronous arbitration; and
- Port repeater services.

Unlike higher layers, we have not developed an explicit API for the physical layer. The link layer directly communicates with the physical layer by an exchange of packets over the defined *packet streams*. These functions are addressed in the following subsections.

6.1.1 Automatic Port and Connection Configuration

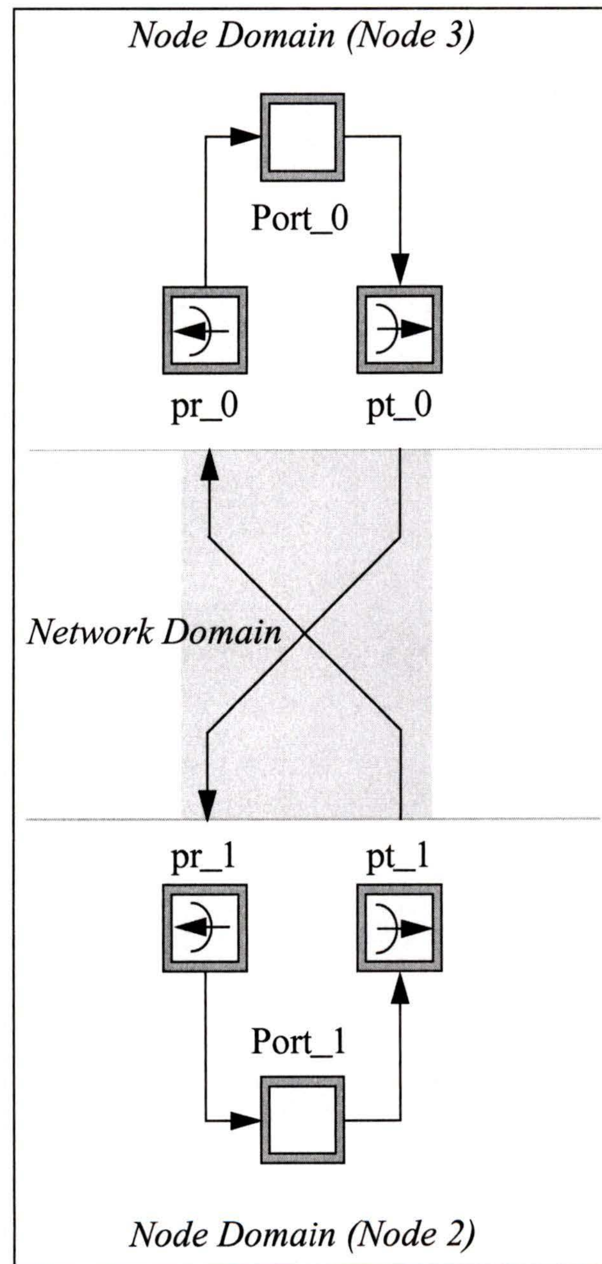


Figure 6.3: A Connection between Ports

Within a topology, a node may be directly connected to at most 3 other nodes. Each connection between nodes is defined by matched receive and transmit ports. For instance, a connection between Port_0 of Node 3 and Port 1of Node 2 would be defined by pairing pr_0 of Port_0 - Node_3 with pt_1 of Port_1 - Node_2 and pt_0 of Port_0 - Node_3 with

pr_1 of Port_1 - Node_2. These pairings, illustrated above in Figure 6.3, are defined within the attributes of the connection. These attributes are illustrated below in Figure 6.4.

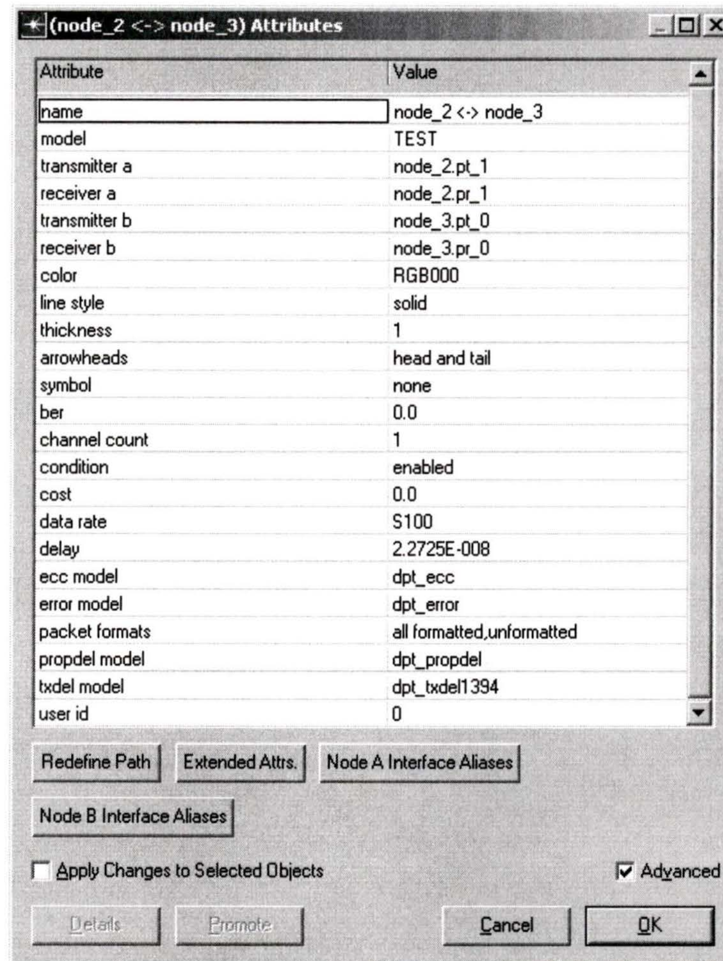


Figure 6.4: Connection Model Attributes

When the model initially starts, the topology of the network is unknown. Within our node model, port processes Port_1 .. Port_3 start by discovering the existence of each other. They also test for active connections on their pr_n/pt_n ports, and they determine their parent or child relationship within the connection.

Within the 1394 standard, the parent/child states of each port, along with the node numbers are discovered with the tree discovery and self-id phases, respectively. For simplicity, our model does not implement these phases, however, the node number and parent/child states are defined within the node attributes. This is illustrated below in Figure 6.5.

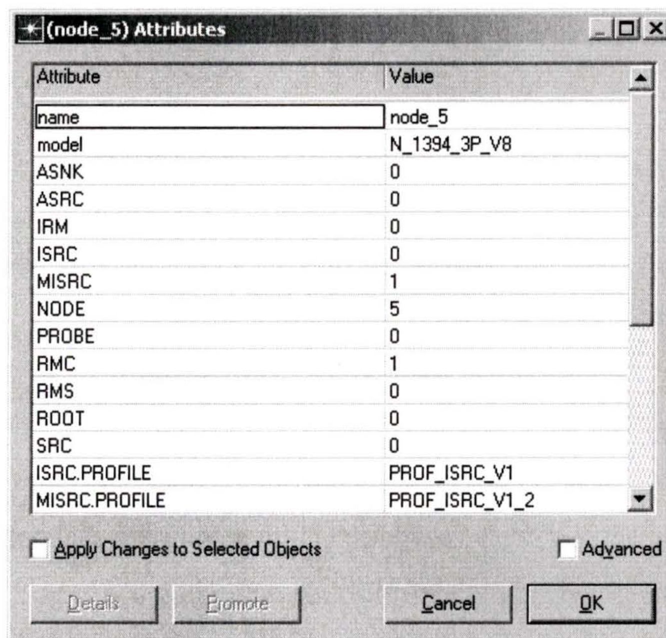


Figure 6.5: Node Model Attributes

The attribute *name* defines the label for the node within the network domain. More important is the *NODE* attribute; it defines the node's number within the network. Other attributes from *ASNK* to *SRC* are boolean values, representing the availability of specific application level functions.

6.1.2 Cycle Start Management

The operation of the serial bus is reliant on the transmission of a Cycle Start Packet (CSP) to mark the beginning of every 125 μs period. This function is performed by the Cycle Start and Tx_ASYNC processes at the Link and Physical layers, respectively. A single node within our network model is defined as the Root; the Cycle Start and ROOT processes on other nodes remain inactive.

Upon expiration of the 125 μs timer, the Cycle Start process yields a CSP and forward it to the Tx_ASYNC process. At the earliest opportunity, the Tx_ASYNC process arbitrates and gains access to the bus. The CSP is then be forwarded from the Tx_ASYNC process to the port processes, where it is then propagated across the bus.

6.1.3 Port Repeater Services

A 1394 device may have one or more ports. For instance, a notebook computer may be equipped with a single port, therefore, providing connectivity with a video camera or a larger network. Alternatively, an AV device may have 2 ports, enabling it to be daisy-chained amongst several other devices. We have designed our model to support 3 ports; this is a common format for 1394 network interface cards that are available for Intel-based personal computers. At the base of Figure 6.1, processes labelled Port_0, Port_1, and Port_2 encapsulate the functionality of these three ports. Below theses ports are the standard OPNET receive and transmit ports (e.g., pr_*n* and pt_*n*) to which our ports communicate. A connection between our individual nodes pair the pr_*n* and pt_*n* ports of Port_*N* with the pr_*m* and pt_*m* ports of Port_*M*.

As explained in Chapter 2, the topology of a 1394 serial bus forms a tree. In supporting multiple ports we must address the propagation of data packets and control signals over each of the neighbouring ports, hence propagating the packets and signals to other parts of the tree. For instance, a data packet received on Port_1, must be duplicated and forwarded over Port_0 and Port_2. The port processes in the physical layer of our implementation manage how the packets and signals are propagated across the topology.

6.1.4 Gap Detection

During the span of 125 μs cycle, idle time, or gaps must also be detected by the individual nodes so that arbitration can begin. For instance, upon detecting an isochronous gap, a node which has an isochronous packet to send will enter into an arbitration phase.

The GAP process within our physical layer listens to activity on the bus, as delivered by the Rx_ISOCH and Rx_ASYNC processes. Upon detecting a particular gap length, the GAP process signals neighbouring processes. For instance, when the GAP process detects

a subaction gap, it signals the Tx_ASYNC process. If an asynchronous packet is waiting to be transmitted, upon receiving the gap signal the arbitration phase will commence.

6.1.5 Arbitration Services

Our model follows the arbitration scheme defined within the IEEE 1394 Standard. After detecting an appropriate gap, a node requests access to the bus for transmission. This request is propagated through the network to the root node. The first request to be received by the ROOT process will be granted. However, by the standard, the inter-node transmission of arbitration request and grant messages is done through a signalling scheme over 4 data lines, not through the transmission of control packets. As OPNET does not easily support this granularity of data transmission, we have implemented the signalling by using a set of small packets.

In our model, the GAP process triggers the arbitration process by signalling the Tx_ASYNC process in the event of a subaction gap, or by signalling the Tx_ISOCH process in the event of an isochronous gap. If the Tx_ASYNC/Tx_ISOCH process has a packet waiting to be transmitted, the arbitration request *tx_req* is transmitted. As specified in the standard, a request is propagated to the root node. The node which wins the arbitration receives a packet which encapsulates a *tx_grant* message. Other nodes receive a packet which encapsulates a *data_prefix* message; this prefixes a data packet from the winning node. In either case, the ARB process informs the Tx_ASYNC/Tx_ISOCH process of the outcome. The port processes play a central role in forwarding these arbitration messages amongst the internal processes and out over connections to neighbouring nodes.

6.2 Link Layer

The link layer portion of our model provides the functions as found in the 1394 Standard. It provides a means for transmitting and receiving asynchronous and isochronous packets over the 1394 bus.

6.2.1 Link Layer Functionality

In our model, the Link layer provides the following services:

- Cycle start generation - A cycle start event is initiated at the link layer and forwarded to the physical layer every 125 μs;
- Asynchronous requests, responses, and acknowledgements - The link layer provides the transaction layer services for supporting asynchronous transactions;
- Isochronous transactions - The link layer is a source of incoming and outgoing isochronous data; and
- Support for concatenated isochronous transactions - If a node is a source for more than one channel, it may transmit one packet for each channel during any one isochronous period. Our link layer yields a concatenated packet, containing at most, one packet for each channel.

6.2.2 Link API

The Link API (shown below in Table 6.2) provides services to the Application layer and Transaction layer. The API provides a set of LK_DATA calls which are used to support the processing of asynchronous requests. These are used by the transaction layer for the transmission and acknowledgement of both request and response packet types. The LK_ISO set of calls are used by the application layer (which include the driver) to control and initiate the transmission and reception of isochronous data.

Table 6.2: Link Layer API

API Call	Description
LK_DATA_request	LK_DATA_request packetizes and sends asynchronous request and response messages to the physical layer.
LK_DATA_confirmation	LK_DATA_confirmation conveys the success of an outstanding link data request to the transaction layer.
LK_DATA_indication	LK_DATA_indication passes the contents of a received request or response packet to the transaction layer.

Table 6.2: Link Layer API

API Call	Description
LK_DATA_response	LK_DATA_response passes an acknowledgement to the physical layer to indicate the status of a received request or response packet.
LK_BUS_indication	LK_BUS_indication informs the transaction layer that an ARB_REST_GAP event has been detected.
LK_ISO_CONTROL_request	LK_ISO_CONTROL_request enables the application layer to specify the channels which it requires to receive.
LK_ISO_request	LK_ISO_request is used by the application layer to yield an isochronous packet and initiate the transmission process.
LK_ISO_indication	LK_ISO_indication forwards an incoming isochronous packet to the application layer if the channel number has been requested by a local client process.
LK_CYCLE_indication	LK_CYCLE_indication informs the application layer that a cycle start packet has been received.

6.3 Transaction Layer

Our transaction layer implements a split transaction where request and response packets may be sent during different subactions. Therefore, our implementation is composed of 4 processes, shown in Figure 6.1; each process is responsible for sending or receiving a request or response packet. This separation allows multiple outstanding transactions; ultimately, the driver reconciles the received responses with an outstanding requests.

6.3.1 Transaction Layer Functionality

In our model, the Transaction layer provides the following services:

- Asynchronous write, read, compare_swap, and fetch_add transactions - These standard transaction types operate on the memory local to each node;
- Quadlet and block operations supported - For read and write transactions, both block and quadlet packet formats are defined. As in the standard, only the block

format is provided for the lock transaction; and

- Single phase transaction retry implemented - The single phase transaction retry is the simplest retransmission model specified within the standard; the request or response packet is resent upon an acknowledgement timeout, or when an acknowledgement indicates a problem with transmission or reception of a packet.

6.3.2 Transaction Layer API

Our Transaction layer API (shown below in Table 6.3) provides the means for sending, receiving, and verifying the success of asynchronous transactions.

Table 6.3: Transaction Layer API

API Call	Description
TR_DATA_request	TR_DATA_request initiates an asynchronous transactions by sending a request packet to the physical layer based on parameters from the application layer.
TR_DATA_confirmation	TR_DATA_confirmation informs the application layer about the completion status of the asynchronous transaction from the transaction layer.
TR_DATA_indication	TR_DATA_indication forwards the parameters of an incoming asynchronous request to the application layer of the effected node.
TR_DATA_response	TR_DATA_response initiates the response phase of the transaction after the application layer has received and processed the data request.

6.4 Driver Layer

The IEEE 1394 Standard specifies that the application layer is above the transaction layer, therefore, our driver is considered part of the application layer. However, for illustrative purposes, we consider the driver as separate from other processes at the application layer.

The driver layer of our model provides the user applications with access to the underlying 1394 services. For both asynchronous and isochronous transactions the driver adds functionality not specified by the 1394 standard. This functionality includes:

- Asynchronous Transactions - The *read*, *write* and *lock* transactions are provided as specified in the Standard.
- Memory space access protection and callback services - Asynchronous transactions which are made against a particular range of memory are controlled by the driver. For instance, permissions can be set to make an area of memory read-only. Callback services are provided so that an application can be informed when a specific area of memory has been accessed or modified. This is a useful feature if a specific area of memory is used as a command register to which other nodes can write application level commands.
- Isochronous resource acquisition operations - The standard process of acquiring isochronous resources involves a sequence of read and lock transactions. The driver provides convenient services so that the application need not be burdened by such details when acquiring and releasing resources.
- Cycle-relative timing services with callback - For the purposes of synchronizing processes across the network, the cycle start packet (CSP) can be exploited. As each CSP includes clock information from the Root node, other nodes which receive the CSPs can update their own cycle-relative clocks. These clocks are managed by the driver. A mechanism for registering callbacks is available, therefore interupting a calling process upon timer completion.
- Buffer Management - Buffers are filled with incoming isochronous payloads, or outgoing payloads from a user application; and
- Isochronous Data Management - Services are provided for (un)binding input and output channels to buffers.

6.4.1 Driver Layer API

The Driver API (shown below in Table 6.4) provides services for the application to access the underlying functions of the 1394 models. These services are modelled after those found in the Windows 2000 1394 Device Driver Kit [11]. A detailed specification of the

Driver Layer API can be found in Appendix E.

Table 6.4: Driver Layer API

Call	Description
DR_GetNodeID	The node ID is not dynamically determined within the simulation as there are no tree identification or self ID phases. The "NODE" attribute is manually set within the network domain of the OPNET simulation. This routine returns the node ID attribute value.
DR_GetBusID	For purposes of the current simulation this call always returns '1'. Our OPNET simulation does not require support for multiple buses.
DR_GetIOAddress	DR_GetIOAddress returns the structure defined with nBus, nNode, and nOffset parameters. Only implements 32 bits of the 48bit offset. The high order 16 bits are assumed to be 0x0000.
DR_AsyncRead	DR_AsyncRead reads a block of memory from a device on the 1394 bus. The requested block of memory must be within an available range of addresses as specified by the target node.
DR_AsyncWrite	DR_AsyncWrite writes a block of memory to a device on the 1394 bus. The block of memory that will be written into must be within the available range of addresses as specified by the target node.
DR_AsyncLock	DR_AsyncLock performs several lock related operations on a block of memory on the 1394 bus. Both compare_swap and fetch_add operations are supported for a variety of operand lengths. The memory block to be updated must fall within the available range of addresses as specified by the target node.
DR_IsochAllocateBandwidth	DR_IsochAllocateBandwidth reserves isochronous allocation units based on the bus speed and the number of bytes per frame. After the successful completion of this transaction, the application must not transmit a frame which exceeds the specified maximum. If the application is the source of more than one channel, this routine can be called more than once. The sum of all requisitions is the maximum size of the resulting concatenated packets.

Table 6.4: Driver Layer API

Call	Description
DR_IsochAllocateChannel	DR_IsochAllocateChannel reserves a channel for the transmission of isochronous packets. After the successful completion of the transaction, the calling node is the only source for the allocated channel.
DR_IsochFreeBandwidth	DR_IsochFreeBandwidth releases isochronous bandwidth, which was previously allocated by the DR_IsochAllocateBandwidth call.
DR_IsochFreeChannel	DR_IsochFreeChannel releases an isochronous channel, which was previously allocated by the DR_IsochAllocateChannel call.
DR_AllocateBuffers	DR_AllocateBuffers allocates a ring of buffers. Applications retrieve and send isochronous data through these buffers.
DR_DeallocateBuffers	DR_IsochAllocateBuffers releases the resources associated with the buffers.
DR_IsochListen	DR_IsochListen registers a request to receive isochronous packets of a specific channel. Once received by the driver, packet payloads are inserted into the specified buffer. The application is interrupted each time a new payload is placed in the buffer by the driver.
DR_IsochTalk	DR_IsochTalk registers a channel number and buffer with the driver. The application subsequently writes frames to the buffer, causing the driver to extract each frame and encapsulate it in an isochronous data request.
DR_IsochStop	DR_IsochStop terminates the reception or transmission of isochronous data associated with a specific channel number and buffer.
DR_SetCallBack	DR_SetCallBack allows a client to register a callback for various incoming asynchronous requests. Access permissions for local address can also be specified through this routine.

6.5 Application Level Processes

The device driver, as shown above, is used by application level processes to provide real-

time media streaming and inter-device control. At this level, the traffic characteristics of the devices within our network are defined.

For the purpose of modelling multimedia systems, based on a 1394 infrastructure, we focused on the following requirements:

- Payload source generators - Our sources model high bit-rate multimedia streams. We must model both the arrival times, and sizes of these transport packets;
- Control mechanisms over isochronous sources - Depending upon the desired quality which we wish to yield, our source streams must be adaptive to yield a range of bit rates;
- Inter-node command and control - Application level control within a multimedia system requires a means for exchanging commands between devices. Once received, the commands must be interpreted and processed. Our resource management system requires inter-node control.

The implementations of the resource management system and the data sources address these above requirements. Their implementations are described in Chapter 7.

7 IEEE 1394 Management Implementation

The 1394 Simulator, presented in the previous chapter, provides a platform on which to test our resource management modules. The simulated nodes can take on a variety of behaviours. For instance, one node can provide the functionality of the Isochronous Resource Manager (IRM), while another node yields a steady stream of asynchronous traffic, and other nodes provide the management functionality that we are considering. Hence, we specialize the behaviour of the individual 1394 nodes by augmenting their functionality at the application layer.

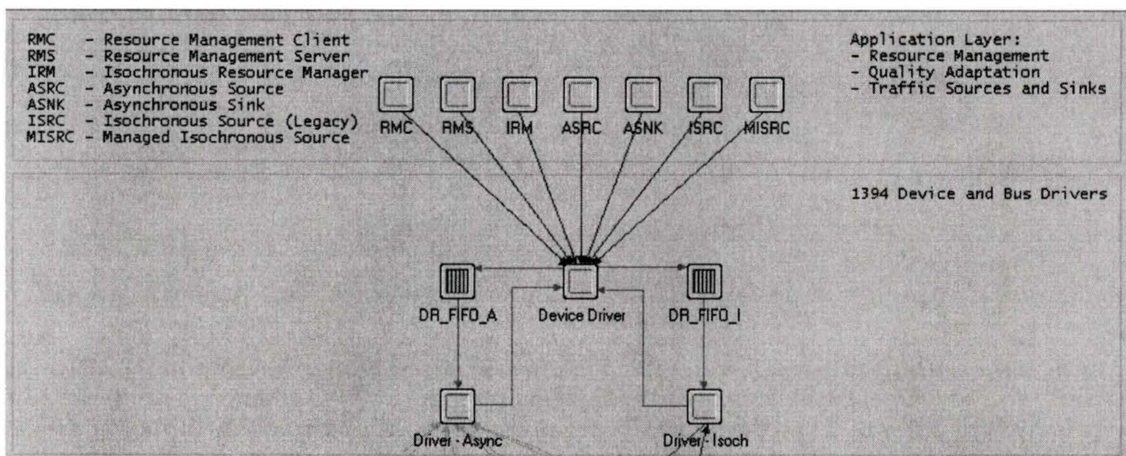


Figure 7.1: Application Layer of 1394 OPNET Simulation

In Figure 7.1 we illustrate the various processes that are provided at the application layer. Each of them (e.g., IRM, ASNK, etc.) is defined within its own module; each module is interconnected to the underlying driver. Every node within our topology

contains these modules, however, we can selectively toggle their existence via a set of parameters, local to each node.

In this chapter we illustrate the implementation of these modules. We first show those modules which provide enhanced resource management functionality, followed by those that provide the behaviour of a standard 1394 topology.

7.1 Management Modules

Our management scheme for IEEE 1394 addresses the issues of resource allocation and control over real-time multimedia sources. The resource allocation requirements are implemented by the Resource Management Server (RMS) and the Resource Management Client (RMC). The control over resource utilization is addressed by the Managed Isochronous Source (MISRC).

Within our simulation environment, the RMS, RMC, and MISRC modules are implemented as individual OPNET processes [12]. The OPNET processes are defined by a Finite State Machine (FSM). However, OPNET's FSM model should not be confused with the conceptual FSM used to model the properties of a communications protocol. We use the OPNET FSM model as general-purpose construct for the definition of a single process. In OPNET, the implementation of a single protocol may require several FSMs.

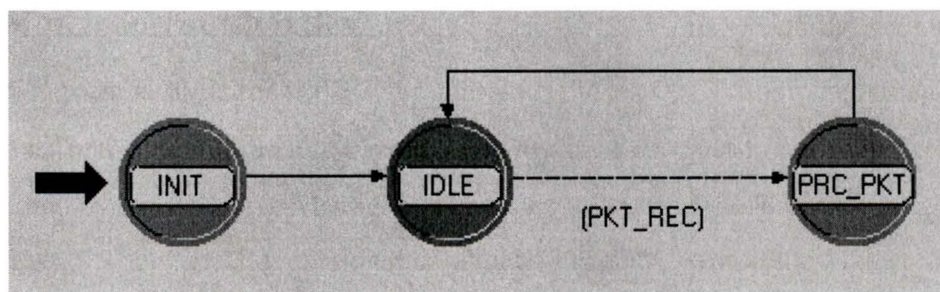


Figure 7.2: An OPNET FSM

As an example, a three-state FSM shown above in Figure 7.2; the process repeatedly receives and then processes packets. When the process is first started, the C code within the INIT state is executed, then the process remains in the IDLE state until an interrupt occurs: specifically, an interrupt due to the reception of a packet from another process. The

PKT_REC transaction condition is essentially a C macro that defines a boolean expression for the conditions of this packet reception. Once received, the packet is processed by executing the C-code in the PRC_PKT state, then a transition occurs back to the IDLE state. The C code within the states and transitions of the FSM calls a collection of proprietary OPNET libraries [13].

7.1.1 Resource Management Server

The RMS module is active on top of one node within our topology. The process which implements this model, provides for the initial election phase and the subsequent management phases. Hence, the RMS module must able to: receive requests from RMC modules, generate a system utility, and then inform the RMC modules of their individual allocations. These responsibilities are highlighted below.

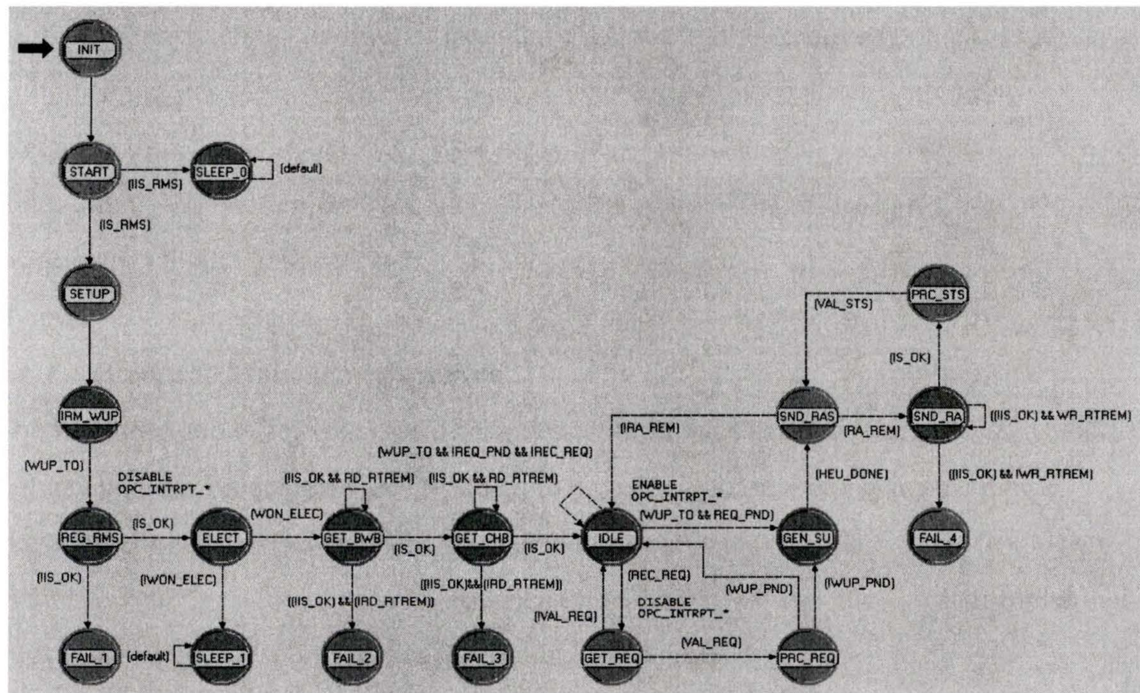


Figure 7.3: Resource Management Server FSM

The RMS module is illustrated in the FSM shown above in Figure 7.3. The responsibilities of the individual states are outlined below in Table 7.1.

Table 7.1: Resource Management Server FSM States

State	Description
1. INIT	Miscellaneous process initialization occurs.
2. START	The process determines if RMS functionality is defined for this specific node.
3. SLEEP_0	RMS functionality was not defined for this node, therefore, this process remains in the SLEEP_0 state.
4. IRM_WUP	Because legacy nodes may coexist with our managed nodes, we allow for a warm-up-period so that legacy devices can acquire resources from the IRM. We assume that the resources remaining after this acquisition period can be managed by our system. This state provides a waiting period.
5. REG_RMS	The RMS process communicates with the RMC using asynchronous write transactions. In this state the RMS process will setup a command register, into which incoming management messages can be written. The RMS calls the DR_SetCallBack() procedure to define the boundaries of this register. After this registration has completed, and once on the IDLE state, the RMS process receives a callback message at every instance of a message being written to the register area.
6. ELECT	A node that provides RMS services must be elected from all those which can support such capabilities. An election between candidates is won by the node that is first to write its address into a well-known register location on the IRM node. This state calls the DR_AsyncLock() driver procedure to accomplish this task.
7. SLEEP_1	If this process did not win the election in the previous phase, it remains in the SLEEP_1 state.
8. GET_BWB	If this state is reached, this node has been elected to provide RMS functionality. The DR_AsyncRead() driver procedure is used to read the bandwidth available register of the IRM. Therefore, the RMS can determine the residual bandwidth available to be managed.
9. GET_CHB	In this state, the DR_AsyncRead() driver procedure is used to read the channels available register of the IRM. Therefore, the RMS can determine the residual channels available to be managed.

Table 7.1: Resource Management Server FSM States

State	Description
10. IDLE	The RMS waits in an IDLE state until a management message has been received from a remote RMC. After receiving a valid message by way of a callback, a transition occurs into the GET_REQ state.
11. GET_REQ	The command message, sent from an RMC using an asynchronous write transaction, is interpreted in this state. If the message is received shortly after a bus-reset, a warm-up-period occurs so that other RMCs can submit requests before a system utility is rendered by our implementation of the utility model.
12. PRC_REQ	The allocation request provided by the previous phase is added to the local profile database within this state.
13. GEN_SU	When this state is entered, the system utility for the resident sessions are rendered. Then, those nodes whose operating qualities have been modified are selected from the session database. Finally, the affected sessions are sorted by their aggregate resource savings.
14. SND_RAS	The previous state provides a sorted list of sessions whose session quality has changed. This state oversees the distribution of command messages which convey the new resource allocations to the affected nodes.
15. SND_RA	This state is responsible for sending a single (re)allocation message to the client. The DR_AsyncWrite() driver procedure is used to send the command message.
16. PRC_STS	The success of transporting the allocation message is considered in this state. The SND_RAS state is then be revisited to determine if there are outstanding messages to be sent.

7.1.2 Resource Management Client

The RMC is the local representative to any constituent processes on the node which require the use of the managed resources. The RMC receives request messages from MISRC processes, and subsequently submit requests to the RMS. Allocation messages received from the RMS are subsequently forwarded to the respective MISRC process. In this sub-section we illustrate the operation RMC module from the perspective of the FSM.

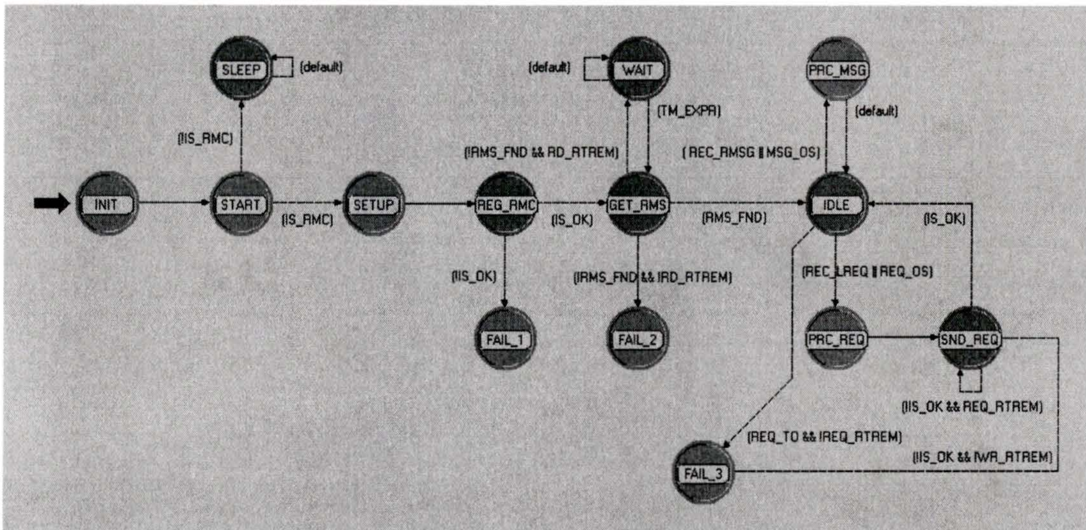


Figure 7.4: Resource Management Client

Our RMC module has been implemented as a OPNET process. The FSM for the MISRC process is illustrated above in Figure 7.4. We highlight the operations of the RMC module in a state-by-state manner in Table 7.2 below.

Table 7.2: Resource Management Client FSM States

State	Description
1. INIT	Miscellaneous process initialization occurs.
2. START	The process determines if RMC functionality is defined for this specific node.
3. SLEEP	RMC functionality is not offered by this node, therefore, this process remains in the SLEEP state.
4. SETUP	RMC functionality is offered by this node. In this state, miscellaneous RMC process initialization occurs.

Table 7.2: Resource Management Client FSM States

State	Description
5. REG_RMC	The RMC process communicates with the RMS using asynchronous write transactions. Each RMC will setup a command register into which incoming management messages can be written. Within this state the RMC calls the DR_SetCallBack() procedure to define the boundaries of this register. After this registration has completed, and the process is in the IDLE state, the RMC process receives a callback message at every instance of a message being written to the register area.
6. GET_RMS	After the a node has been elected to provide RMS services, the node's address must be determined by the RMC before resource requests can be submitted. In this state, the DR_AsyncRead() driver procedure is called to read the contents of a well-known register location on the IRM node.
7. WAIT	If the RMS election has not yet completed, the previous GET_RMS state fails. Therefore, a transition to this state provides a timeout period before the register access is retried.
8. IDLE	The RMC waits in an IDLE state until a management message has been received from either a local MISRC process, or the remote RMS. Depending on the origin of the message, from the RMS or the MISRC, a transition is made to either the PRC_MSG, or PRC_REQ states, respectively.
9. PRC_MSG	A callback has been received due to a command being written to the underlying register. This command message, from the RMS, is likely an allocation message. In this state, it is interpreted and any appropriate messages are forwarded to the MISRCs.
10. PRC_REQ	A command message has been received from the MISRC. This is likely to be an allocation or a deallocation request. In this state, the request is prepared for transmission.
11. SND_REQ	An allocation or a deallocation request from the MISRC results in an command message being set to the RMS. This is accomplished through a call to the DR_AsyncWrite() driver procedure.

7.1.3 Managed Isochronous Source

A MISRC component provides an adaptive isochronous source that is controlled by our resource management scheme. The MISRC communicates with its local RMC by sending control messages which define the multi-level resource requirements, along with other

policy specific information. The RMC sends control messages to the MISRC which ultimately direct the resource consumption of the isochronous source. In this sub-section we illustrate the operation of the MISRC module from the perspective of its FSM.

Figure 7.5: Managed Isochronous Source FSM

Our MISRC module has been implemented as a OPNET process. The FSM for the MISRC process is illustrated above in Figure 7.5. We highlight the operations of the MISRC in a state-by-state manner in Table 7.3 below.

Table 7.3: Managed Isochronous Source FSM States

State	Description
1. INIT	In this state, miscellaneous process initialization occurs.
2. START	In this state, the process determines if MISRC functionality is offered for this specific node.

Table 7.3: Managed Isochronous Source FSM States

State	Description
3. SLEEP	MISRC functionality was not defined for this node, therefore, this process remains in the SLEEP state.
4. SETUP	Arrival at the SETUP state implies that the node provides MISRC functionality. In this state, the session quality profile and resource mappings are read from an external file. Other initializations also take place.
5. WAIT	The process remains suspended in this state until the session is scheduled to begin yielding isochronous traffic.
6. GET_BUF	A buffer for the isochronous traffic is allocated in this state. This is accomplished through the DR_AllocateBuffers() driver procedure.
7. RMC_REQ	A request message is submitted to the local RMC. The request contains the resource requirements for each supported operating quality, along with other policy-specific parameters. The process is suspended in this state until an allocation message has been received back from the local RMC.
8. RMC_MSG	A message has been received from the local RMC and interpreted to determine the resultant allocation from the RMS. The pre-scribed operating quality is then determined.
9. REQ_TO	If the session was not admitted by the RMS, a timeout occurs before another request is attempted. After the timeout, the RMC_REQ state is revisited.
10. REL_BW_0	If isochronous bandwidth has not yet been acquired from the IRM, this state is ignored. If bandwidth has been previously acquired and the new operating quality requires a change in bandwidth requirements, the existing allocation is released by using the DR_IsochFreeBandwidth() driver procedure.
11. GET_BW	The initial or modified bandwidth requirements are acquired using the DR_IsochAllocateBandwidth() driver procedure.
12. GET_CH	A channel is acquired using the driver procedure in this state. We assume that only one channel is required, regardless of the pre-scribed operating quality, hence, an intra-stream scalability scheme is modelled.
13. SET_TALK	Once a channel identifier has been acquired, the identifer is bound to the previously allocated buffer using the DR_IsochTalk() driver procedure.

Table 7.3: Managed Isochronous Source FSM States

State	Description
14. GET_PL	In this state we model the size and inter-arrival time of each isochronous payload. The process is suspended in this state until the next payload has been yielded.
15. SND_PL	Once in this state, the payload is available. It is written to the buffer using the BUFF_WriteTail() procedure. If remaining packets within the session are to be sent, a transition back to the previous GET_PL state occurs.
16. SET_STOP	If no additional isochronous payloads remain to be sent, the binding between the isochronous channel and the buffer is released using the DR_IsochStop() driver procedure.
17. REL_BUF	The buffer is released using the DR_DeallocateBuffers() driver procedure.
18. REL_CH	The channel identifier is released using the DR_IsochFreeChannel() driver procedure.
19. REL_BW_1	The bandwidth is released using the DR_IsochFreeBandwidth() procedure.
20. RMC_REL	A message is sent to the local RMC. This releases reservations associated with this process. Then, a transition back to the WAIT state occurs.

7.2 Legacy Modules

Our management components were described in the previous section. This section illustrates the unmanaged, or legacy, modules at the application layer. These provide the functionality which is provided by the standard 1394 bus. They coexist within our network simulation model to provide background traffic, and to prove that our approach for management is inter-operable with the 1394 Standard.

A typical 1394 topology supports a mixture of isochronous and asynchronous traffic. Each of the following modules provides some aspect of this legacy behaviour.

7.2.1 Isochronous Resource Manager

The Isochronous Resource Manager (IRM) is an elected node which provides several functions to the bus. The primary function being its role in tracking the available bus

resources through a set of registers; nodes read and then update the values of these registers to indicate their reservation requirements. This function is defined within the Serial Bus Management layer of the 1394 Architecture.

We have placed this function at the application layer as it requires the services of our driver. Upon start-up, our IRM declares the address ranges associated with its set of registers. This callback request allows *read* and *lock_swap* operations against the registers from other nodes on the bus. After this initialization, the IRM process is alerted to each request for access to the underlying registers, however, it provides no other active role.

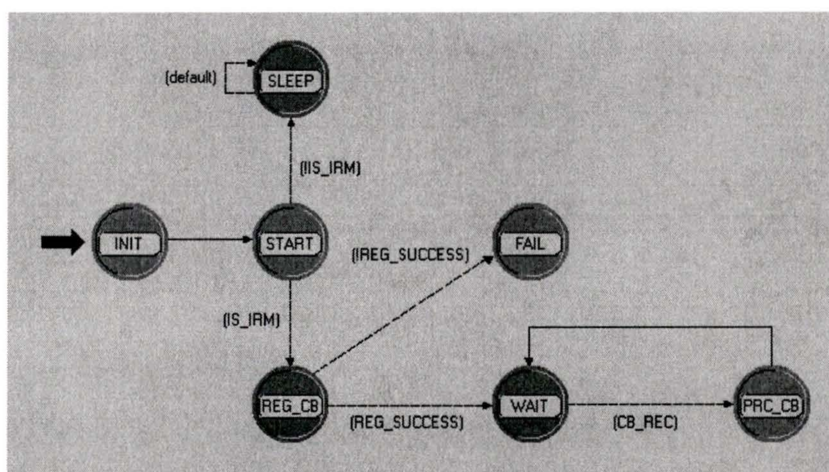


Figure 7.6: Isochronous Resource Manager FSM

Our IRM module has been implemented as a OPNET process. The FSM for this process is illustrated above in Figure 7.6. We highlight the state-by-state operations of the ISRC in Table 7.4 below.

Table 7.4: Isochronous Resource Manager FSM States

State	Description
1. INIT	Miscellaneous process initialization occurs.
2. START	In this state, the process determines if IRM functionality is offered by this specific node.

Table 7.4: Isochronous Resource Manager FSM States

State	Description
3. SLEEP	IRM functionality was not offered by this node, therefore, this process remains in the SLEEP state.
4. REG_CB	In this state, the IRM process defines the register boundaries and access rights using the DR_SetCallBack() procedure. After registering these boundaries, the initial values of these registers are set.
5. FAIL	The callback registration process failed.
6. WAIT	The process remains in this state until an interrupt is received from the driver, indicating an incoming request operation against the specified registers. A service request block (SRB) is received from the driver; this indicates the parameters and operands of the incoming request.
7. PRC_CB	The IRM does not play a management role apart from hosting the registers, therefore, no action is taken when a callback is received. Resources associated with the SRB are released in this state. A transition to the previous WAIT state occurs.

7.2.2 Isochronous Source

Because our management components may likely co-exist with legacy devices which independently acquire bus resources, we have provided the Isochronous Source (ISRC) to model this legacy behaviour. Initially, the ISRC component acquires the bandwidth and channel resources from the IRM. Local buffer space is then configured and the channel identifier is bound to the buffer. Isochronous Payloads are then generated based on a defined distribution.

Our ISRC module has been implemented as a OPNET process. The FSM for this process is identical to that for the MISRC (illustrated above in Figure 7.5); the same module was used to minimise FSM and code maintenance. We highlight the state-by-state operations of the ISRC FSM in Table 7.5 below.

Table 7.5: Isochronous Resource Manager FSM States

State	Description
1. INIT	In this state, miscellaneous process initialization occurs.

State	Description
2. START	In this state, the process determines if ISRC functionality is offered for this specific node.
3. SLEEP	ISRC functionality was not defined for this node, therefore, this process remains in the SLEEP state.
4. SETUP	Arrival at the SETUP state implies that the node provides ISRC functionality. In this state, the session quality profile and resource mappings are read from an external file. Other initializations also take place.
5. WAIT	The process remains suspended in this state until the session is scheduled to begin yielding isochronous traffic.
6. GET_BUF	A buffer for the isochronous traffic is allocated in this state. This is accomplished through the DR_AllocateBuffers() driver procedure.
7. RMC_REQ	This state is bypassed.
8. RMC_MSG	This state is bypassed.
9. REQ_TO	This state is bypassed.
10. REL_BW_0	This state is bypassed.
11. GET_BW	The initial or modified bandwidth requirements are acquired using the DR_IsochAllocateBandwidth() driver procedure.
12. GET_CH	A channel is acquired using the driver procedure in this state. We assume that only one channel is required, regardless of the prescribed operating quality, hence, an intra-stream scalability scheme is modelled.
13. SET_TALK	Once a channel identifier has been acquired, the identifier is bound to the previously allocated buffer using the DR_IsochTalk() driver procedure.
14. GET_PL	In this state we model the size and inter-arrival time of each isochronous payload. The process is suspended in this state until the next payload has been yielded.
15. SND_PL	Once in this state, the payload is available. It is written to the buffer using the BUFF_WriteTail() procedure. If remaining packets within the session are to be sent, a transition back to the previous GET_PL state occurs.
16. SET_STOP	If no additional isochronous payloads remain to be sent, the binding between the isochronous channel and the buffer is released using the DR_IsochStop() driver procedure.

State	Description
17. REL_BUF	The buffer is released using the DR_DeallocateBuffers() driver procedure.
18. REL_CH	The channel identifier is released using the DR_IsochFreeChannel() driver procedure.
19. REL_BW_1	The bandwidth is released using the DR_IsochFreeBandwidth() procedure.
20. RMC_REL	This state is bypassed.

7.2.3 Asynchronous Source

The Asynchronous Source (ASRC) module yields asynchronous requests against other nodes on the bus. This provides background traffic during the later phase of the 125 μs isochronous cycle. Each of the request types (i.e., read, write, lock) is yielded by the ASRC module, and these operations act on operands of varied sizes.

Within each state which is associated with a transaction type, the transaction may not be performed, and a transition may occur to the next state. This allows for a variety of transaction patterns.

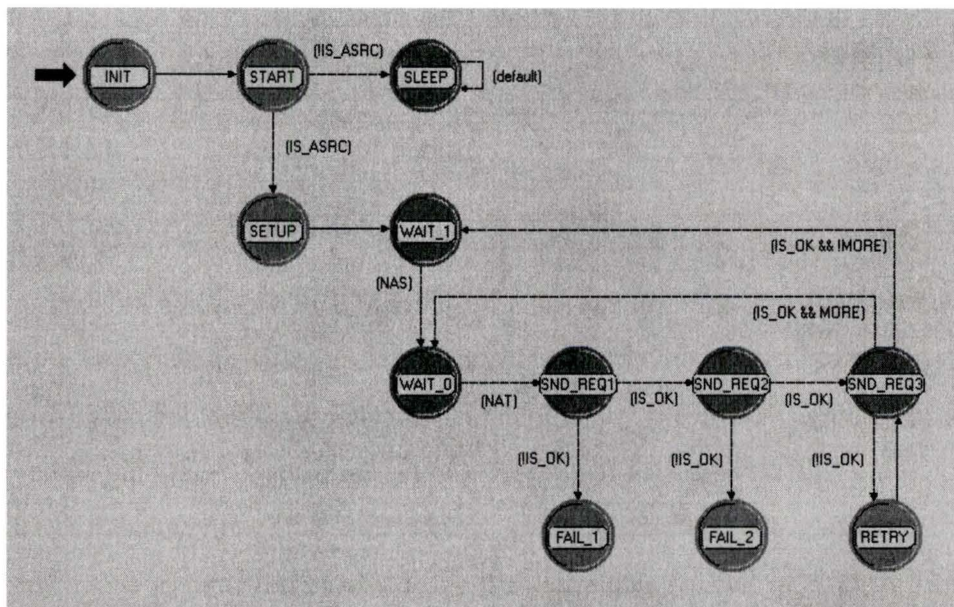


Figure 7.7: Asynchronous Source FSM

Our ASRC module has been implemented as a OPNET process. The FSM for this process is illustrated above in Figure 7.7. We highlight the state-by-state operations of the ASRC FSM in Table 7.4 below.

Table 7.6: Asynchronous Source FSM States

State	Description
1. INIT	Miscellaneous process initialization occurs.
2. START	The process determines if ASRC functionality is offered by this specific node.
3. SLEEP	ASRC functionality was not offered by this node, therefore, this process remains in the SLEEP state.
4. SETUP	Memory is allocated and parameters are defined for the operations.
5. WAIT_0	The process is suspended until the asynchronous transaction occurs. The inter-arrival time between these sets of transactions is defined in this state.
6. SND_REQ_1	A *read* request is performed using the DR_AsyncRead() driver procedure. The size of the block that is read will vary.
7. SND_REQ_2	A *write* request is performed using the DR_AsyncWrite() driver procedure. The size of the block that is written will vary.
8. SND_REQ_3	A *lock_swap* request is performed using the DR_AsyncLock() driver procedure,. An eight byte block is updated with each request. After completion If more transactions are to occur, the next state is WAIT_0, otherwise the process waits for the next session in the WAIT_1 state.
9. WAIT_1	All transactions have occurred.

7.2.4 Asynchronous Sink

The Asynchronous Sink module is a destination for asynchronous requests. The sink initially registers one or more address ranges within which certain operations (i.e., write, lock) can take place. A callback is registered for that specific range. When a operation is made against the memory location, the process is interrupted with a callback, stating the operation and any related operands. This mechanism is useful for processing commands

which are written to particular registers.

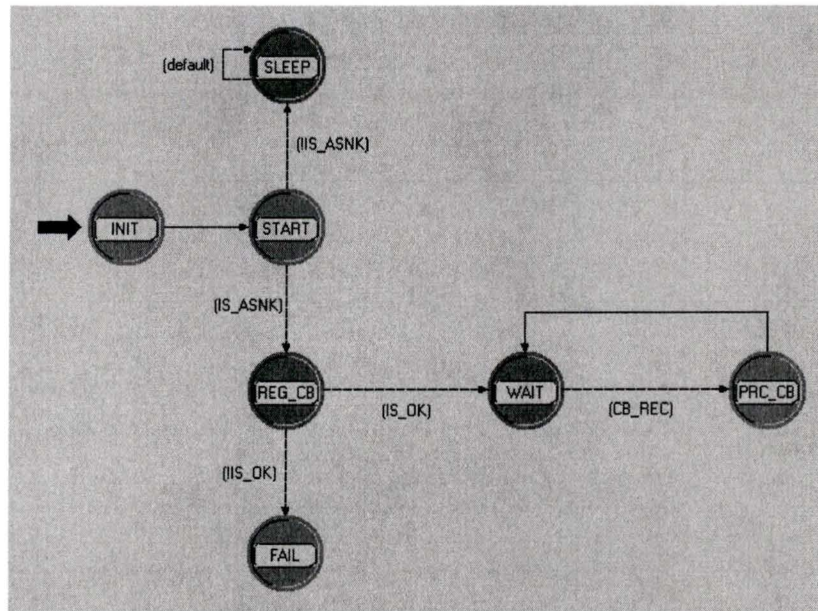


Figure 7.8: Asynchronous Sink FSM

Our ASNK module has been implemented as a OPNET process. The FSM for this process is illustrated above in Figure 7.8. We highlight the state-by-state operations of the ASNK FSM in Table 7.7 below.

Table 7.7: Asynchronous Sink FSM States

State	Description
1. INIT	Miscellaneous process initialization occurs.
2. START	The process determines if ASNK functionality is defined for this specific node.
3. SLEEP	ASNK functionality was not defined for this node, therefore, this process remains in the SLEEP state.
4. REG_CB	The ASNK process defines the address boundaries and access rights using the DR_SetCallBack() driver procedure. These areas are accessible by the received ASRC requests.
5. FAIL	The callback registration process failed.

Table 7.7: Asynchronous Sink FSM States

State	Description
6. WAIT	The process remains in this state until an interrupt is received from the driver, indicating an incoming request operation against the specified registers. A service request block (SRB) is received from the driver; this indicates the parameters and operands of the incoming request.
7. PRC_CB	The ASNK process is the endpoint for the ASRC traffic generation. Resources associated with the SRB are released in this state. A transition to the previous state occurs.

8 Experiments and Results

The components which make up our simulation environment were outlined in the previous two chapters: Chapter 6 illustrated our IEEE 1394 implementation, and Chapter 7 gave an overview of the management and traffic generating modules that are based on our 1394 implementation.

In this chapter we present the experiments and performance results of our resource management modules. We first describe our topology for experimentation. Then, we illustrate the parameters and statistics for the modules. And finally, the individual experiments and their results are summarized.

8.1 Topological Parameters

We have chosen a network topology which is similar in scale and function to a multimedia network within a home or small office setting. The bus bandwidth is limited to 100 Mbps, and it is shared amongst eight nodes. The ninth node on the bus is a passive probe which measures a variety of activities. Our topology under consideration is shown below in Figure 8.1.

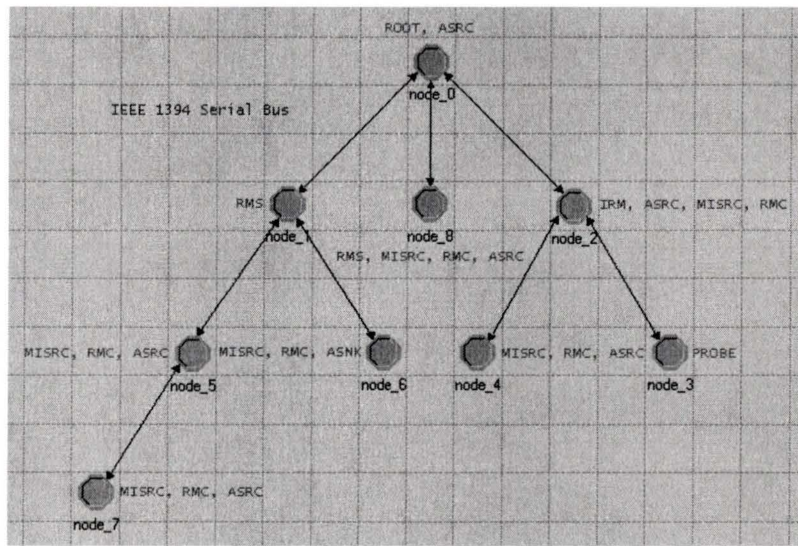


Figure 8.1: Simulation Topology

Although the above Figure depicts a managed network, the same topology is used to model an unmanaged network; the differences are illustrated below in Table 8.1. We note that the naming scheme for the nodes (e.g., Node_1) reflects the order in which the nodes were added to our topology; it does not represent the 1394 node addressing scheme as described within the self identification phase of bus reset.

Table 8.1: Supported Modules in Managed/Unmanaged Topologies

Node	Managed	Unmanaged
Node_0	ROOT, ASRC	ROOT, ASRC
Node_1	RMS	
Node_2	IRM, ASRC, MISRC, RMC	IRM, ASRC, ISRC
Node_3	PROBE	PROBE
Node_4	MISRC, RMC, ASRC	ISRC, ASRC
Node_5	MISRC, RMC, ASRC	ISRC, ASRC
Node_6	MISRC, RMC, ASNK	ISRC, ASNK
Node_7	MISRC, RMC, ASRC	ISRC, ASRC
Node_8	RMS, MISRC, RMC, ASRC	ISRC, ASRC

The listed modules define the functionality of each node. Several of these modules (i.e., MISRC, ISRC, RMC, RMS, ASRC, ASNK) are outlined in Chapter 7. We note that both Node_1 and Node_8 can potentially provide RMS services, however, because we simulate the RMS election process, only one node is elected to provide these services.

The inclusion of the ROOT module indicates that Node_0 has been selected as the root node within the topology. Therefore, it is responsible for the broadcasting of cycle start packets and for rendering arbitration decisions.

The PROBE attribute does not indicate the inclusion of a specific module, instead it indicates the presence of a specific node type. The PROBE node is illustrated below in Figure 8.2.

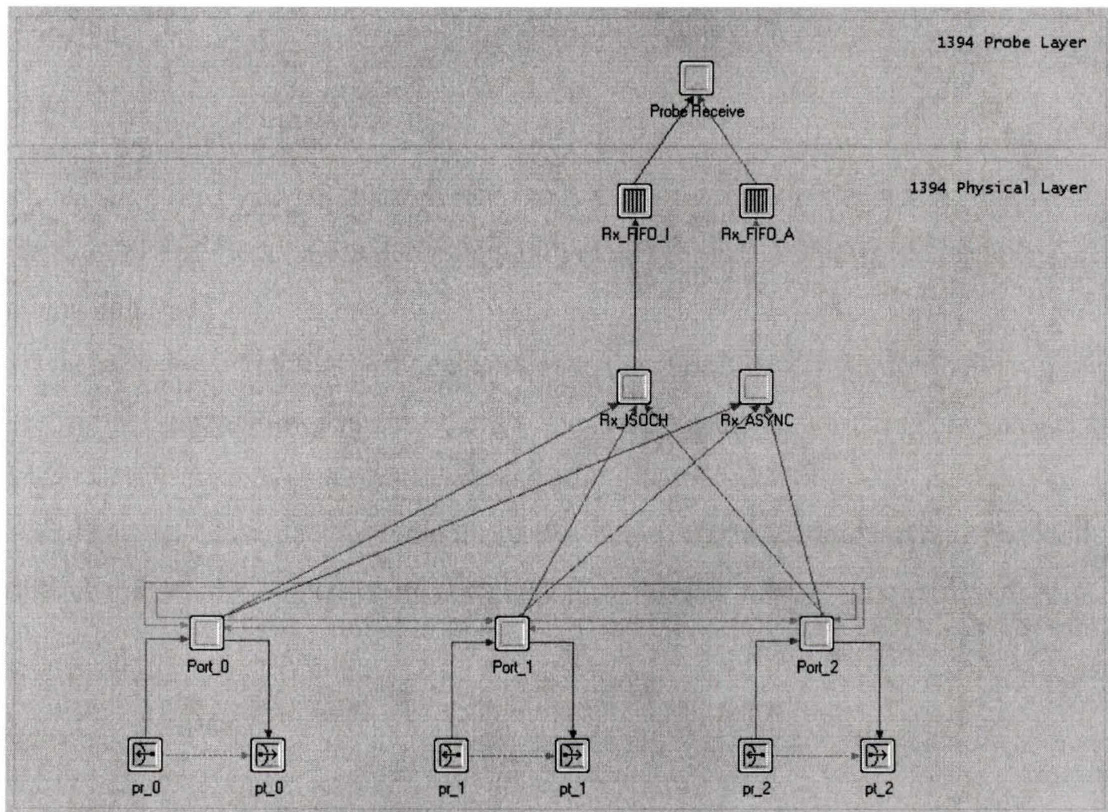


Figure 8.2: Probe Node Architecture

The Probe node is a passive listener to bus traffic. If connected to more than one port, traffic is repeated, however, it is not a source of transactions. A variety of statistics are collected by the *Probe Receive* process. These statistics provide a bus-wide perspective of

traffic and timing. Furthermore, the Probe node provided invaluable data for the purpose of model validation.

8.2 Module Parameters and Statistics

Within our simulation environment, the behaviour of each node is primarily determined by the set of modules which have been selected as active (e.g., MISRC, ASRC). However, the behaviour of each of the selected modules must also be controlled by a set of parameters. Furthermore, for simulation purposes, a set of statistics may also be associated with a module. In this section we define the parameters and statistics associated with the MISRC, ISRC, and ASRC processes.

In the following discussion, we note that the sizes of the illustrated packets are not necessarily indicative of the sizes of their respective payloads. Furthermore, the "..." and the dashed transition lines between the isochronous or asynchronous packets, indicate the possible interleaving of unrelated isochronous periods and asynchronous sub-actions. And finally, the italicised labels represent parameters, whereas the bolded labels represents statistics.

8.2.1 Parameters for Controlling Session Spacing

Although our network has a small number of nodes, we can vary the load by controlling the spacing of sessions from each node. The parameters depicted below in Figure 8.3 are common to all our source types.

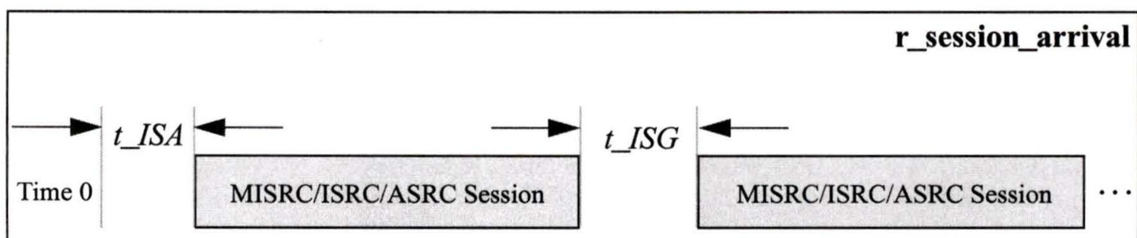


Figure 8.3: Inter-Session Spacing for a Single Source

An explanation of the illustrated parameters and statistics is given in the following tables.

Table 8.2: Parameters for Session Spacing

Parameter	Description
1. t_ISA	Initial Session Arrival: Within the scope of a single source module, the period of time between the simulation startup and the arrival of the first session from a specific source module is specified by *t_ISA* seconds. This is illustrated above in Figure 8.3.
2. t_ISG	Inter-Session Gap: For all of the session types, we have modelled an inter-session gap. For each session source, an idle time of *t_ISG* seconds is inserted between the end of the previous session and the beginning of the next session. This is illustrated above in Figure 8.3.

Table 8.3: Statistics for Session Spacing

Statistic	Description
1. r_session_arrival	Session Arrival Rate: This is illustrated above in Figure 8.3. This statistic is measured in the global context - it is measured across all nodes in the topology.

8.2.2 Managed Isochronous Source

The Managed Isochronous Source (MISRC) process yields both asynchronous and isochronous traffic. The asynchronous traffic is associated with command and control between the MISRC process and the ISRC and IRM processes. The isochronous traffic is associated with the controlled stream of continuous-media data. Both traffic types comprise the phases of an individual session; these phases are illustrated below in Figure 8.4.

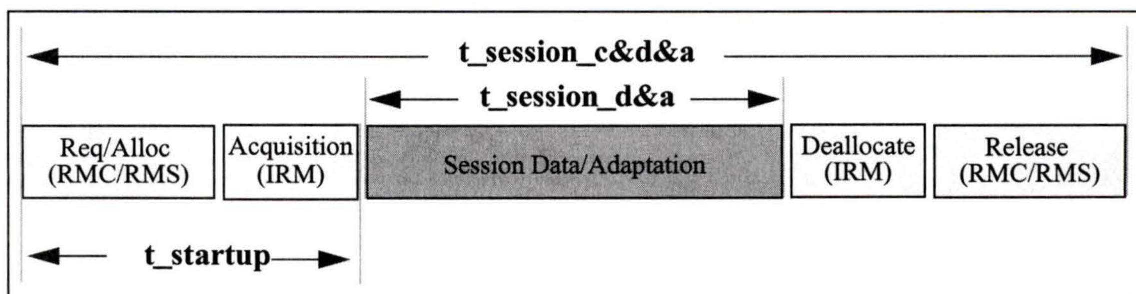


Figure 8.4: Managed Isochronous Session Phases

Each of the phases is explained below in Table 8.4 and illustrated in the following Figures: 8.5, 8.6, 8.7, 8.8, 8.9, and 8.10.

Table 8.4: Managed Isochronous Session Phases

Phase	Description
1. Request / Allocate	This phase yields asynchronous packets which are associated with the messaging between RMC and RMS processes. This traffic is illustrated below in Figure 8.5.
2. Acquire	This phase yields asynchronous packets which are associated with standard acquisition of resources from the IRM. This traffic is illustrated below in Figure 8.6.
3. Session Data / Adaptation	This phase is composed of isochronous data (illustrated below in Figure 8.7), and the asynchronous data which is associated with the adaptation requests from the RMS and the resulting reacquisition of resources from the IRM (illustrated below in Figure 8.8).
4. Deallocate	This phase yields asynchronous packets which are associated with standard deallocation of resources from the IRM. This traffic is illustrated below in Figure 8.9.
5. Release	This phase is composed of a single message which specifies a release request from the RMC to the RMS. This traffic is illustrated below in Figure 8.10.

The following Figures illustrates the asynchronous and isochronous packets which are yielded during the various phases of the management of an isochronous session.

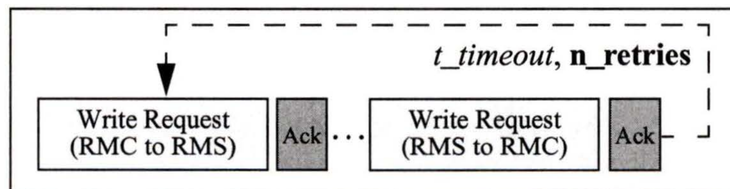


Figure 8.5: Managed Isochronous Source: Request/Allocation Phase

The two asynchronous write request packets depicted above in Figure 8.5 carry the request and allocation messages from the RMC and RMS, respectively. If the requesting session was not admitted, a timeout and retry occurs.

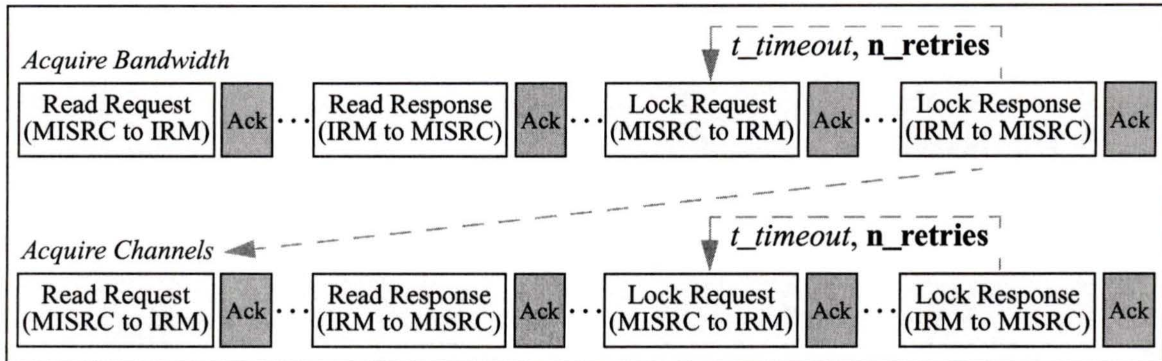


Figure 8.6: Managed Isochronous Source: Acquisition Phase

The traffic associated with the *Acquisition Phase* is shown above in Figure 8.6. During this phase, the MISRC attempts to read and then lock (compare_swap) for the purpose of gaining bandwidth allocation units from the IRM. If the lock transaction was unsuccessful, it is repeated after a timeout. A similar procedure is repeated for the acquisition of a channel identifier.

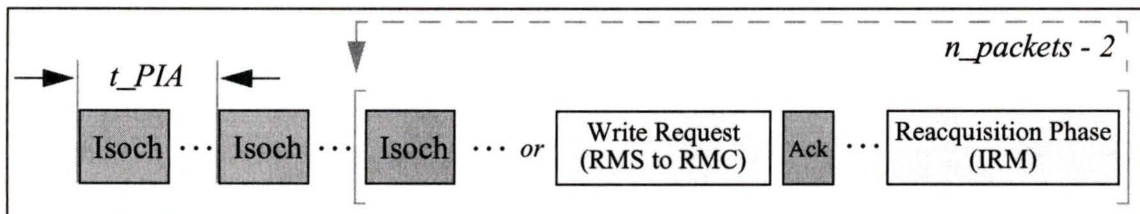


Figure 8.7: Managed Isochronous Source: Session Data/Adaptation Phase

The traffic associated with the *Session Data/Adaptation Phase* is shown above in Figure 8.7. During this phase, the isochronous data is transmitted and adaptation requests are received and processed. An adaptation request is received from the RMS process within a single asynchronous write packet. The dimension *n_payload* of the isochronous packet remains constant until an adaptation request and the reacquisition phase have completed.

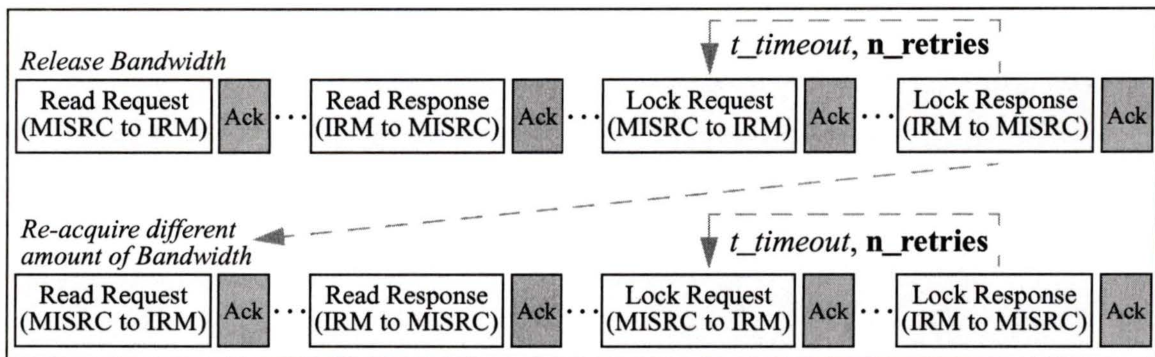


Figure 8.8: Managed Isochronous Source: Reacquisition Phase

After an adaptation has been received from the RMS, via the RMC (shown as a write request packet, above in Figure 8.7), the MISRC must adapt its resource utilization. The MISRC first releases its current bandwidth allocation to the IRM, then it requires a new bandwidth allocation. Both steps use the standard combination of read and then lock (compare_swap) transactions against the IRM. If the lock transaction was unsuccessful, it is repeated after a timeout. This process is illustrated above in Figure 8.8.

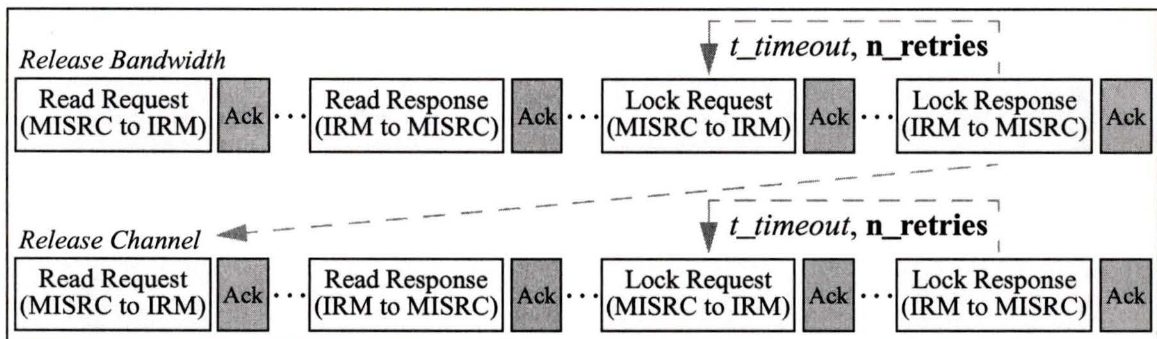


Figure 8.9: Managed Isochronous Source: Deallocation Phase

After all the isochronous data in the session has been sent, the MISRC attempts to release its resources by using the standard combination of read and then lock (compare_swap) transactions against the IRM. If the lock transaction was unsuccessful, it is repeated after a timeout. A similar procedure is repeated for the acquisition of a channel identifier. This process is illustrated above in Figure 8.9.

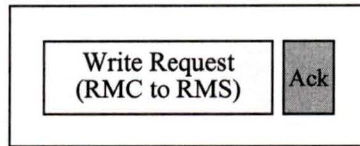


Figure 8.10: Managed Isochronous Source: Session Release Phase

The asynchronous packet depicted above in Figure 8.10 carries the release request from the RMC to the RMS. An explanation of the parameters associated with the MISRC are given in given in the following tables.

Table 8.5: Managed Isochronous Session Parameters

Parameter	Description
1. t_PIA	Packet Inter-arrival Time: Within the scope of a single MISRC module, the period of time between the arrival of successive isochronous packets is specified by *t_PIA*. This parameter is illustrated above in Figure 8.7.
2. n_packets	Number of Packets: Within the scope of a single MISRC module, the number of isochronous packets within a single session is specified by *n_Packets*. This parameter is illustrated above in Figure 8.7.
3. t_timeout	Timeout Period: A timeout/retry occurs when an a request to the RMS did not result in admission, or when an asynchronous compare_swap operation failed. The length of the timeout is specified by *t_timeout* seconds. This parameter is illustrated above in Figures: 8.5, 8.6, 8.8, and 8.9.
4. n_payload	Size of Packet: The number of payload bytes.

Table 8.6: Managed Isochronous Session Statistics

Statistic	Description
1. t_session_c&d&a	Session Duration for Control, Data, and Adaptation: The span of time which encompasses all control and data phases of the session is specified by the **t_session_c&d&a** statistic. This statistic is illustrated above in Figure 8.4.

Statistic	Description
2. t_session_d&a	Session Duration for Data and Adaptation: The span of time which encompasses the data phase of the session is specified by the **t_session_d&a** statistic. This is illustrated above in Figure 8.4.
3. t_session_d	Session Duration for Data: The **t_session_d** statistic is simply the **t_session_d&a** statistic minus any adaptation overhead.
4. t_startup	Startup Time: The span of time for all control before the commencement of the data phase is specified by the **t_startup** statistic. This is illustrated above in Figure 8.4.
5. n_retries	Number of Retires: A timeout/retry occurs when an a request to the RMS did not result in admission, or when an asynchronous compare_swap operation failed. The number of retries until success is recorded in the **n_retries** statistic. This statistic is illustrated above in Figures: 8.5, 8.6, 8.8, and 8.9.

8.2.3 Unmanaged Isochronous Source

The Unmanaged Isochronous Source (ISRC) process yields both asynchronous and isochronous traffic. The asynchronous traffic is due to the acquisition and release of resources between ISRC and IRM processes. The isochronous traffic is associated with the stream of continuous-media data. These phases are illustrated below in Figure 8.11.

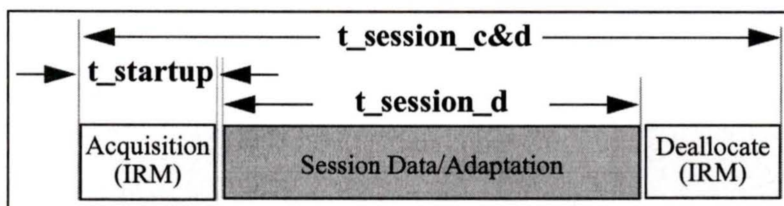


Figure 8.11: Unmanaged Isochronous Source: Session Phases

Each of the phases is explained below in Table 8.7 and illustrated in the following Figures.

Table 8.7: Unmanaged Isochronous Session Phases

Phase	Description
1. Acquisition	This phase yields asynchronous packets which are associated with standard acquisition of resources from the IRM. This traffic is identical to that of the Acquisition phase for the managed source (illustrated above in Figure 8.6).
2. Session Data	This phase is composed of isochronous data This traffic is illustrated below in Figure 8.12.
3. Deallocation	This phase yields asynchronous packets which are associated with standard deallocation of resources from the IRM. This traffic is identical to that of the deallocation phase for the managed source (illustrated above in Figure 8.9).

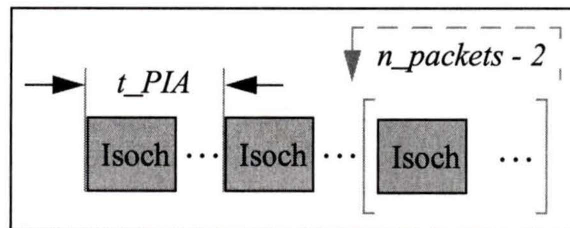


Figure 8.12: Unmanaged Isochronous Source: Session Data Phase

The Session data phase shown above in Figure 8.12 depicts the stream of continuous media. As this stream is yielded by an unmanaged source, its quality (i.e., *n_payload*) remains constant throughout the session.

Table 8.8: Unmanaged Isochronous Session Parameters

Parameter	Description
1. t_PIA	Packet Inter-arrival Time: Within the scope of a single ISRC module, the period of time between the arrival of successive isochronous packets is specified by *t_PIA*. This parameter is illustrated above in Figure 8.12.
2. n_packets	Number of Packets: Within the scope of a single ISRC module, the number of isochronous packets within a single session is specified by *n_Packets*. This parameter is illustrated above in Figure 8.12.

Parameter	Description
3. t_timeout	Timeout Period: A timeout/retry occurs when an asynchronous compare_swap operation failed or insufficient resources exist at the IRM. The length of the timeout is specified by *t_timeout* seconds. This parameter is illustrated above in Figures 8.6, and 8.9.
4. n_payload	Size of Packet: The number of payload bytes.

Table 8.9: Unmanaged Isochronous Session Statistics

Statistic	Description
1. t_session_c&d	Session Duration for Control and Data: The span of time which encompasses all control and data phases of the session is specified by the **t_session_c&d** statistic. This statistic is illustrated above in Figure 8.4.
2. t_session_d	Session Duration for Data: The span of time which encompasses the data phase of the session is specified by the **t_session_d** statistic. This is illustrated above in Figure 8.4.
3. t_startup	Startup Time: The span of time for all control before the commencement of the data phase is specified by the **t_startup** statistic. This is illustrated above in Figure 8.4.
4. n_retries	Number of Retires: A timeout/retry occurs when an asynchronous compare_swap operation fails or when insufficient resources exist at the IRM. The number of retries until success is recorded in the **n_retries** statistic. This statistic is illustrated above in Figures: 8.5, 8.6, 8.8, and 8.9.

8.2.4 Asynchronous Source

The Asynchronous Source (ASRC) process yields a flow of asynchronous packets that provides background traffic across the network. The transactions are targeted towards the Asynchronous Sink (ASNK) process.

A session of asynchronous transactions is composed of read, write and lock transactions. The resultant flow of traffic is depicted below in Figure 8.13.

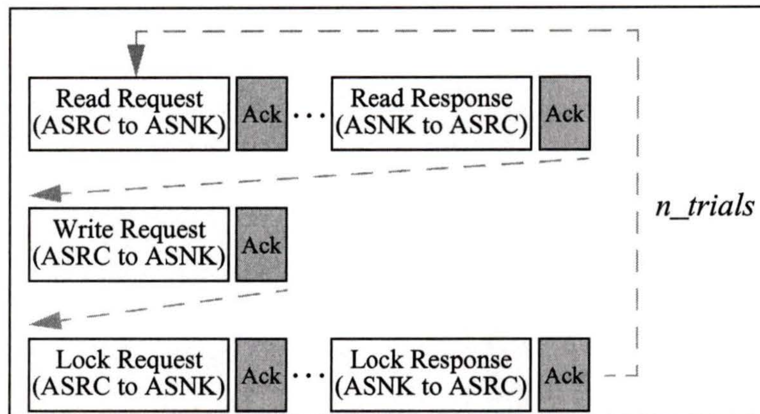


Figure 8.13: Asynchronous Source: Asynchronous Session

Table 8.10: Asynchronous Session Parameters

Parameter	Description
1. n_trials	Number of Trials: Within the scope of a single ASRC module, a sequence of asynchronous transactions is repeated. This parameter is illustrated above in Figure 8.13.
1. t_ITG	Inter-Trial Gap: the spacing between successive trials in a session.
1. n_payload	Size of Packet: Number of payload bytes per transaction.

8.3 Experimental Results

Our simulations were conducted with the intention of determining the following:

1. The effectiveness of our management scheme;
2. The timeliness of the management; and
3. The associated costs of management.

We conducted two sets of simulations; a single variable was varied during each set:

1. *t_ISG*: By varying inter-session gaps between the sessions of both managed and unmanaged isochronous sources, we are in effect changing the traffic load over a series of simulations. There were 34 simulations per set, and each simulation spanned 4 seconds of simulated time. Managed and unmanaged topologies were simulated separately.
2. *n_packets*: By varying the number of isochronous packets within the sessions, of both

managed and unmanaged sources, we are in effect changing the management overhead and the traffic load, over a series of simulations. There were 34 simulations per set and each simulation spanned 6 seconds of simulated time. Managed and unmanaged topologies were simulated separately.

Each of the above trials were conducted for both managed and unmanaged topologies. All parameters for each topology were identical, however, the unmanaged topologies supported the ISRC modules, yet they did not support the RMS, RMC, and MISRC modules. Furthermore, the ISRC module yielded non-adaptive isochronous sessions at the highest operating quality, whereas the MISRC module yielded adaptive sessions. A summary of the simulation parameter values for both trials is shown below in Table 8.11.

Table 8.11: Parameter Values for Simulations

Parameter	Description	trial: t_ISG	trial: n_packets
1. t_PIA (isoch)	A constant inter-arrival rate for the isochronous packets. See Figure 8.7.	0.000125 sec	0.000125 sec
1. t_ITG (async)	A constant inter-trial gap for asynchronous traffic.	0.0001 sec	0.0001 sec
2. t_ISA (isoch)	A random value X of an exponential distribution, with a mean value of 0.005. X defines the delay before the first session at a specific node. See Figure 8.3.	X	X
3. t_ISA (async)	A random value Y of an exponential distribution, with a mean value of 0.001. Y defines the delay before the first session at a specific node. See Figure 8.3.	Y	Y
4. n_trials	The number of times a set of asynchronous transactions is repeated per session. See Figure 8.13.	4	4

Parameter	Description	trial: t_ISG	trial: n_packets
5. t_timeout	The waiting time before a retry. See Figures 8.5, 8.6, 8.8, and 8.9 for examples.	0.001 sec	0.001 sec
6. t_ISG (isoch)	A constant gap between the end and start of isochronous sessions. See Figure 8.3.	[0.00001 .. 1.0] sec	0.01 sec
7. t_ISG (async)	A constant gap between the end and start of asynchronous sessions. See Figure 8.3.	0.0001 sec	0.0001 sec
8. n_packets	The number of packets in an isochronous session. See Figure 8.7.	100	[10 .. 20,000]
9. n_payload (async)	The number of payload bytes per asynchronous transaction (compare_swap has 2*8 bytes).	8	8
10. n_payload (isoch)	The number of payload bytes per isochronous packet, per cycle.	See Table 8.12	See Table 8.12

Table 8.12: MISRC quality profile with requirements and utilities

Operating Quality	Bytes per Cycle	Mbps	BW AUs Required	CH IDs Required	Utility
1.	100	6.4	960	1	20
2.	200	12.8	1360	1	22
3.	300	19.2	1760	1	24
4.	600	38.4	2960	1	26

The above table specifies the number of payload bytes within each source packet, per cycle, for each operating quality. Also specified for each operating quality: the required bandwidth allocation units, and channel identifiers, along with the respective utility.

Each MISRC uses the same profile, therefore, a simple equal-share allocation scheme is enforced. Emphasis has been placed on providing the minimum quality to as many sessions as possible.

8.3.1 Effectiveness of Management

Figures 8.14 and 8.15 below show the **t_startup** delay for both managed and unmanaged isochronous sessions. This delay is representative of the time between the arrival of a new session and the start of the data phase.

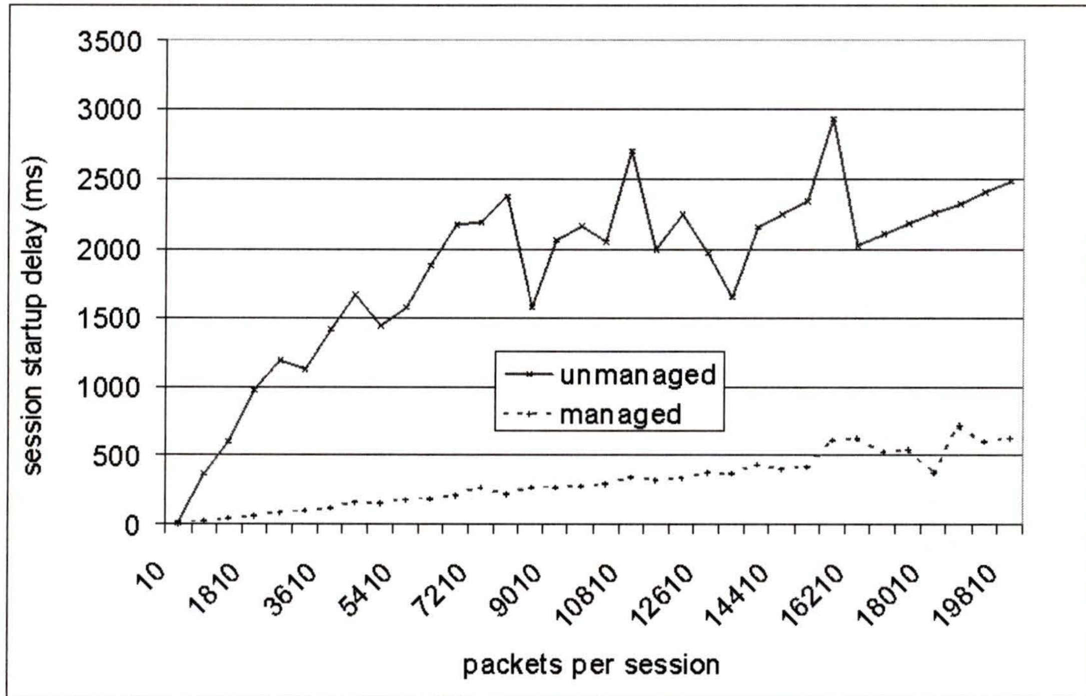


Figure 8.14: Session Startup Delay versus Number of Packets/Session

In Figure 8.14 we show the mean **t_startup** delays for a series of simulations, where the *n_packets* parameter is varied for each simulation. For the unmanaged topology, only one client (see Figure 8.16) session is supported at the highest quality, therefore, other clients must wait until the session has completed. The managed topology can support more clients (see Figure 8.16), and even with larger management overheads (see Figures 8.18), a shorter startup delay is experienced. We note that as the sessions become longer, the time which resources are unavailable also lengthens.

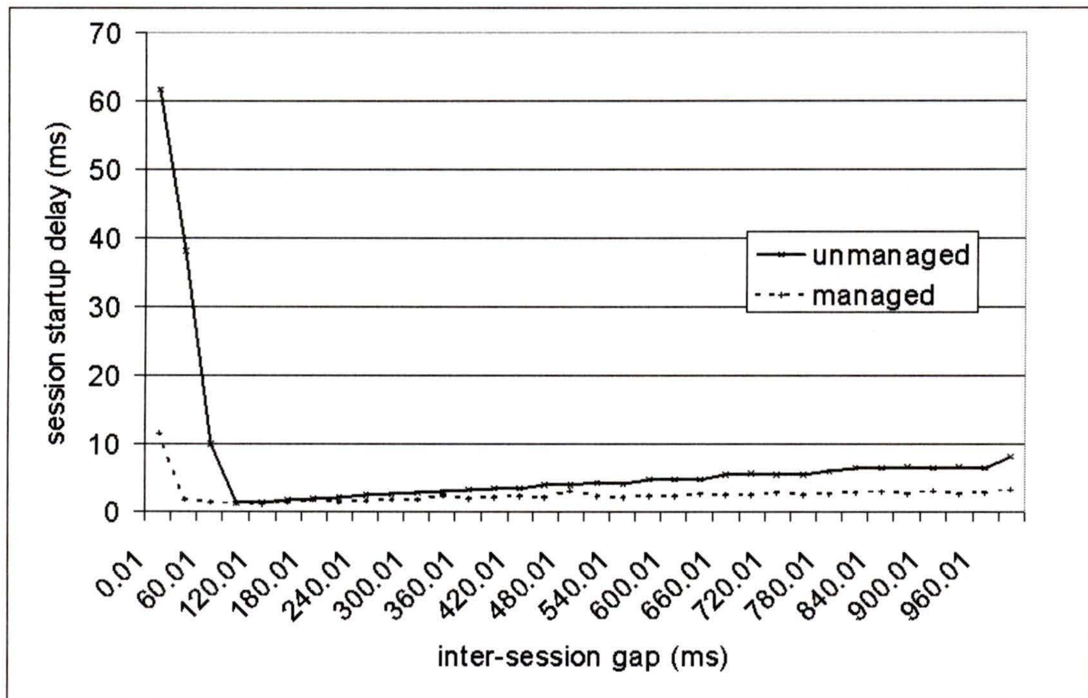


Figure 8.15: Session Startup Delay versus Inter-Session Gap

In Figure 8.15 we show the mean **t_startup** delays for a series of simulations, where the *t_ISG* parameter is varied for each simulation. For the unmanaged topology, only one client (see Figure 8.17) session is supported at the highest quality, therefore, other clients must wait until the session has completed. The managed topology can support more clients (see Figure 8.17), and even with larger management overheads (see Figures 8.18), a shorter startup delay is experienced. We note that as session arrivals become increasingly spread apart, there is less contention and the startup delay for unmanaged sessions decreases. However, we see a gradual increase in startup delay for both session types. This is explained by the increase in asynchronous utilization of the ASRC (see Figures 8.20 and 8.21); messages associated with the ISRC, MISRC, IRM, RMC, and RMS must compete for bus access against the ASRC traffic.

Figures 8.16 and 8.17 below show the mean system capacities for concurrent sessions under the managed and unmanaged isochronous sessions. The capacity is measured as a average, weighted by the time at which a particular capacity is supported.

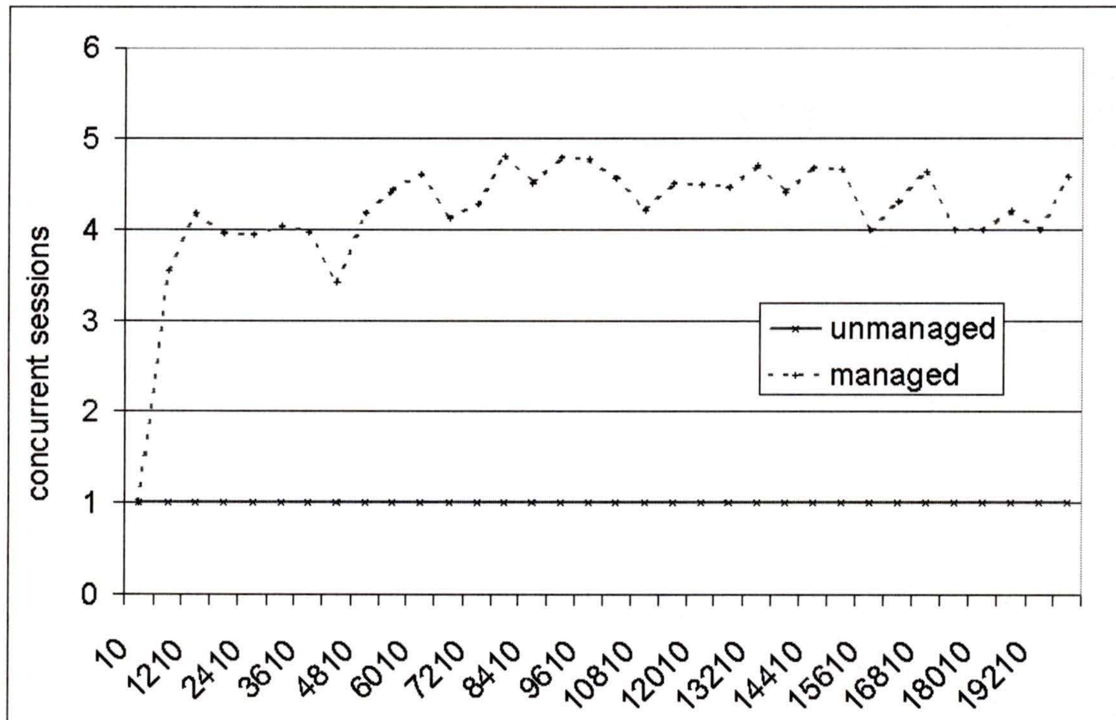


Figure 8.16: Concurrent Sessions versus Number of Packets/Session

In Figure 8.16 we show the mean system capacities for series of simulations, where the *n_packets* parameter is varied for each simulation. For the unmanaged topology, only one client session is supported at the highest quality. The managed topology can support more clients as the sources are adaptive and can operate on lower levels of resources.

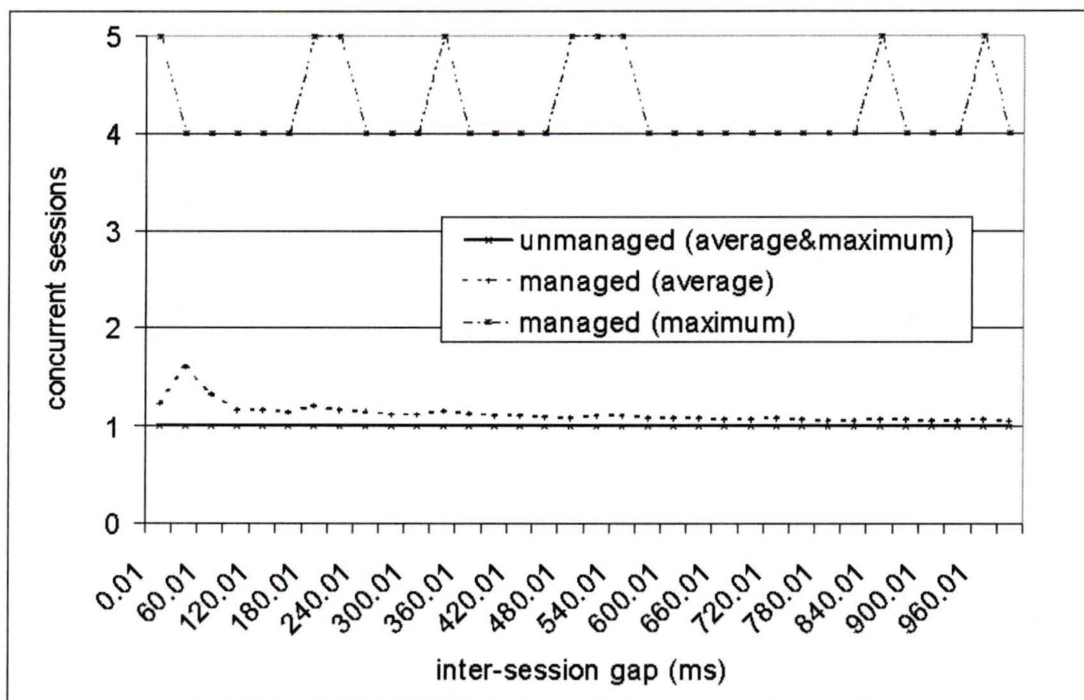


Figure 8.17: Concurrent Sessions versus Inter-Session Gap

In Figure 8.17 we show the average[1] and maximum system capacities for a series of simulations, where the *t_ISG* parameter is varied for each simulation. For the unmanaged topology, only one client session is supported at the highest quality, therefore, other clients must wait, as there are not enough resources to start a second session. The managed topology can support more clients as the sources are adaptive; we show average and maximum measurements.

1. This average is weighted by the time over which each value was valid.

Figures 8.18 and 8.19 below show the mean asynchronous traffic under the managed and unmanaged isochronous sessions. For the managed topology, the traffic is yielded by: the ASRC modules, the transactions between the MISRC and IRM modules, and the messaging between the RMC and RMS modules. For the unmanaged topology, the traffic is yielded by the ASRC modules, and the transactions between the ISRC and IRM modules.

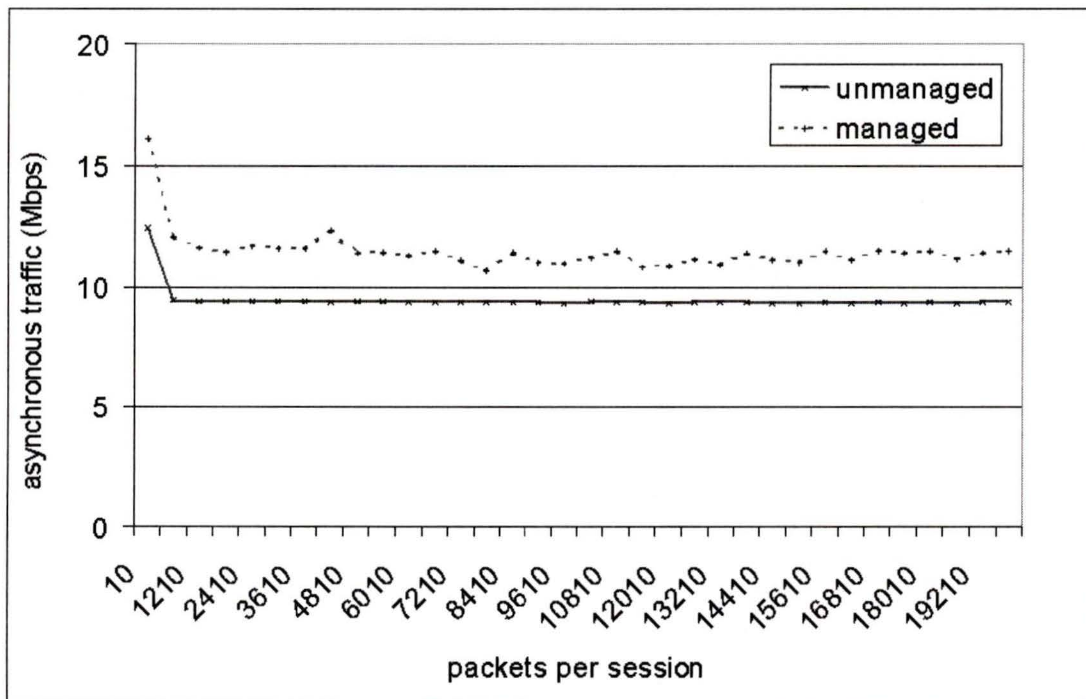


Figure 8.18: Asynchronous Utilization versus Number of Packets/Session

In Figure 8.18 we show the mean asynchronous bus utilization for a series of simulations, where the *n_packets* parameter is varied for each simulation. The difference between these two measurements is introduced from two sources: the extra overhead our management incurs; and the extra opportunities the ASRC has to transmit when the MISRC is not generating traffic, due to adaptation or other management.

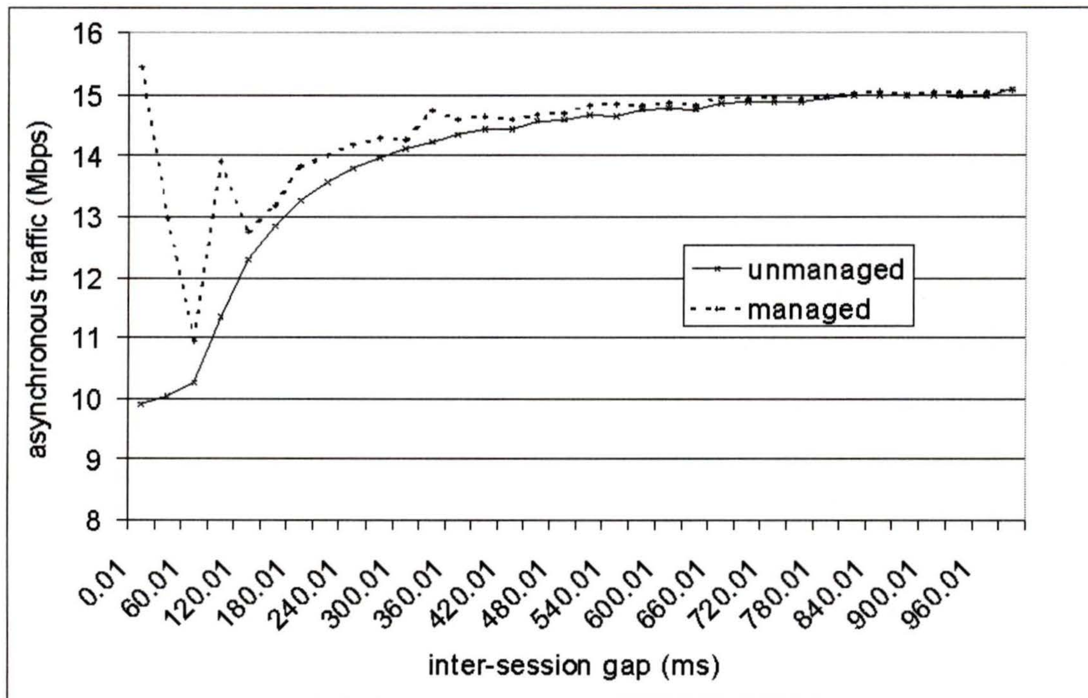


Figure 8.19: Asynchronous Utilization versus Inter-Session Gap

In Figure 8.19 we show the mean asynchronous bus utilization for a series of simulations, where the *t_ISG* parameter is varied for each simulation. For the unmanaged topology, asynchronous utilization gradually increases as the isochronous utilization decreases (see Figure 8.23). This increase is due to the background traffic yielded by the ASRC module. The managed topology yields a utilization which decreases along with the decrease in the arrival rate of new sessions. The variability can be explained by the processing of reallocation requests, and the extra opportunities the ASRC has to transmit when the MISRC is not generating traffic. As with the unmanaged topology: when the isochronous utilization decreases, the asynchronous utilization from the ASRC module increases (see Figure 8.23 for isochronous utilization).

Figures 8.20 and 8.21 below show the components sources of asynchronous traffic for managed and unmanaged topologies, respectively.

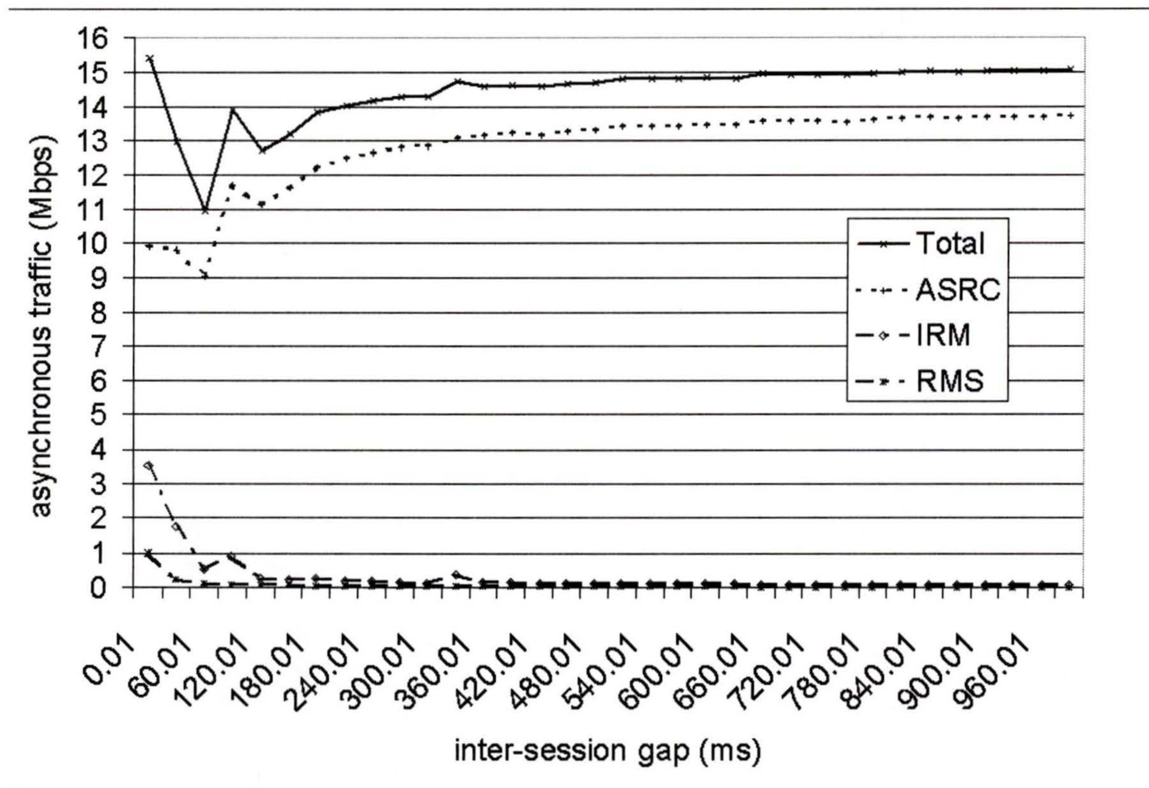


Figure 8.20: Managed Async Utilization versus Inter-Session Gap

In Figure 8.20 we show the mean asynchronous bus utilization, and its components, for a series of simulations, where the *t_ISG* parameter is varied for each simulation. The components of asynchronous traffic for a managed topology include traffic yielded by the ASRC modules (i.e., ASRC), traffic between the MISRC modules and the IRM module (i.e., IRM), and the traffic between the RMC modules and the RMS module (i.e., RMS).

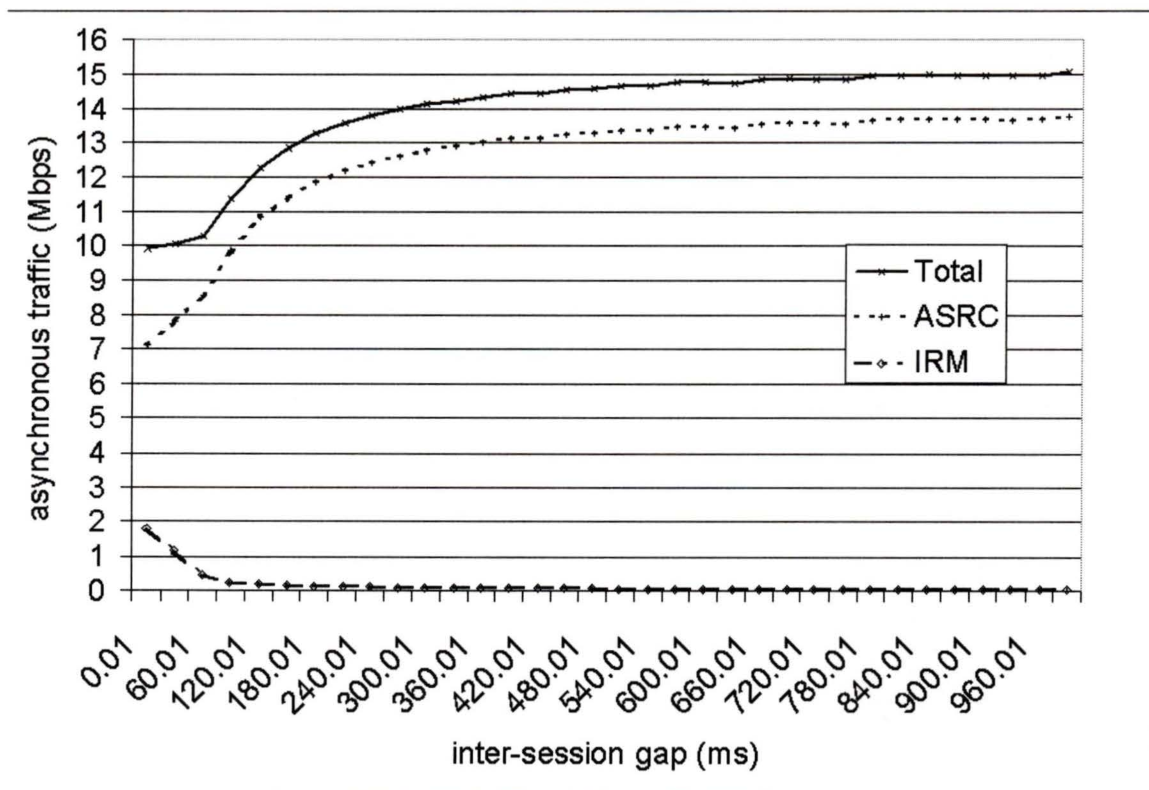


Figure 8.21: Unmanaged Async Utilization versus Inter-Session Gap

In Figure 8.21 we show the mean asynchronous bus utilization, and its components, for a series of simulations, where the *t_ISG* parameter is varied for each simulation. The components of asynchronous traffic for an unmanaged topology include traffic yielded by the ASRC modules (i.e., ASRC), and traffic between the ISRC modules and the IRM module (i.e., IRM).

Figures 8.22 and 8.23 below show the mean isochronous traffic under the managed and unmanaged isochronous sessions. For the managed and unmanaged topologies, the traffic is yielded by the MISRC, and ISRC modules, respectively.

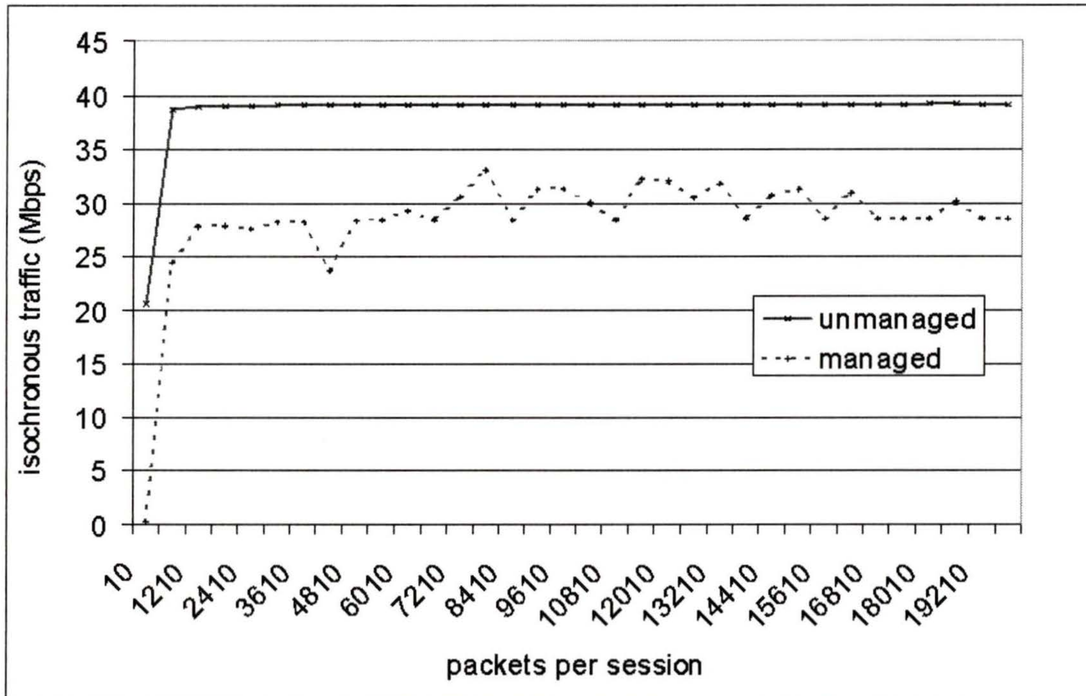


Figure 8.22: Isochronous Utilization versus Number of Packets/Session

In Figure 8.22 we show the mean isochronous bus utilization for a series of simulations, where the *n_packets* parameter is varied for each simulation.

For the unmanaged topology, utilization is constant after the session length dominates over a constant *t_ISG. T*herefore, the idle time, between the increasingly longer sessions, becomes insignificant.

Utilization of the managed topology is lower because the 5 concurrent sessions (see Figure 8.16), at the operating quality 1, yield less traffic than one unmanaged session at full quality (see Table 8.12 for an explaination of the resource requirements). The variability in the number of concurrent sessions and the reallocations (see Figures 8.30 and 8.32) during a session contribute to a slight variability in utilization, as shown above.

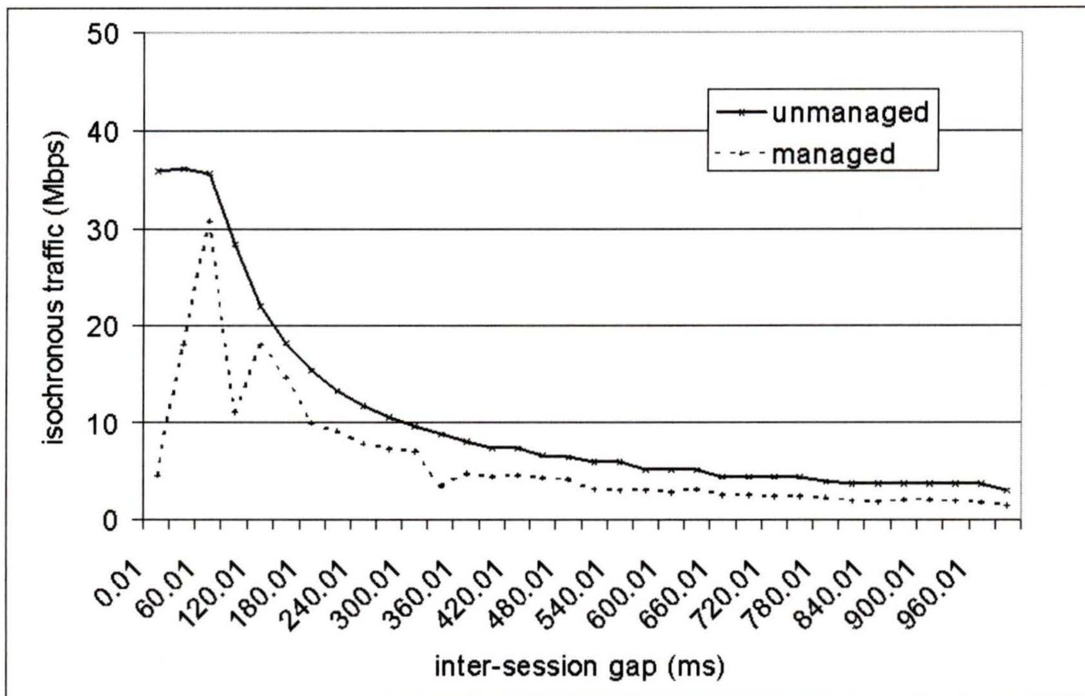


Figure 8.23: Isochronous Utilization versus Inter-Session Gap

In Figure 8.23 we show the mean isochronous bus utilization for a series of simulations, where the *t_ISG* parameter is varied for each simulation. For the unmanaged topology, utilization is constant until session arrivals have sufficiently thinned so that ISRC modules are not continually contending against each other for bus resources.

The managed topology yields a utilization which initially increases due to the decreasing domination of the control phases (i.e., Request/Allocation, Acquisition, Adaptation, Deallocation and Release) over the Session data phase. The later, gradual decrease in utilization is consistent with the fewer session arrivals due to the increasing *t_ISG*.

Figures 8.24 and 8.25 below show the average[1] operating qualities for managed isochronous sessions.

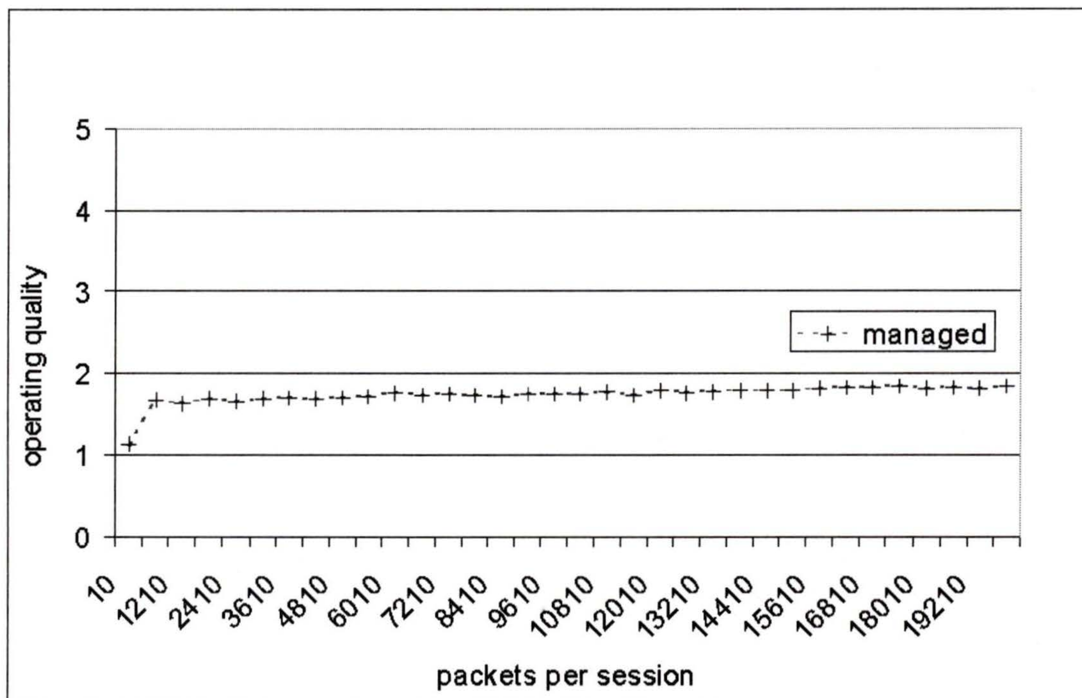


Figure 8.24: Active Operating Qualities versus Number of Packets/Session

In Figure 8.24 we show the operating qualities for a series of simulations, where the *n_packets* parameter is varied for each simulation. The managed topology experiences operating qualities which are consistent with the utilities we assigned to each session (see Table 8.12).

1. This average is weighted by the time over which each value was valid.

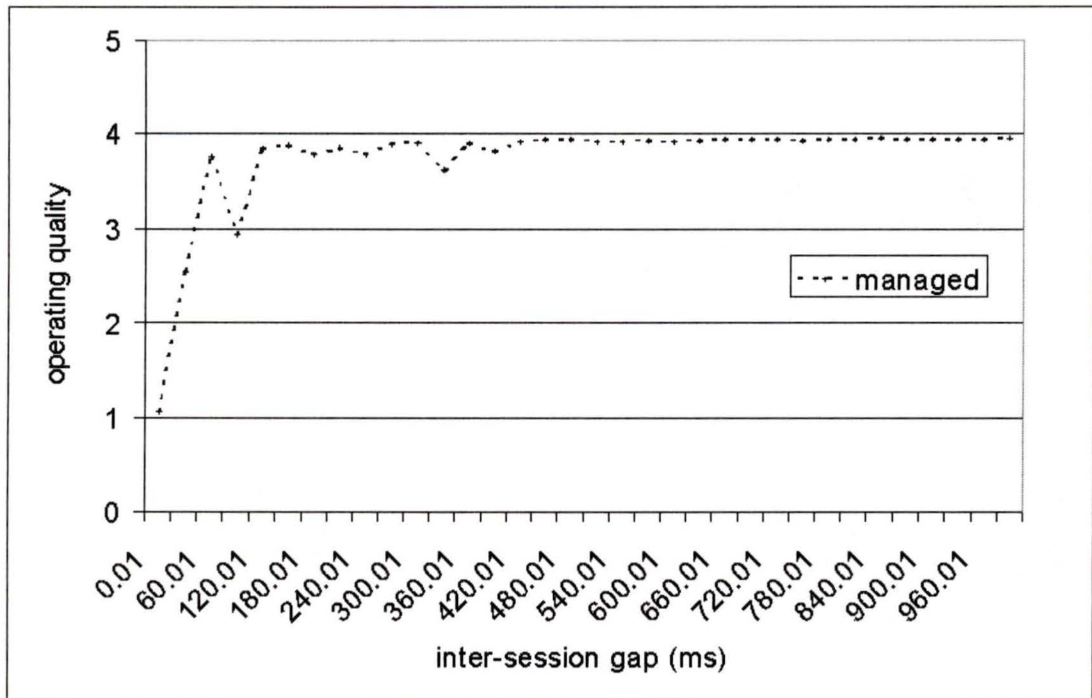


Figure 8.25: Active Operating Qualities versus Inter-Session Gap

In Figure 8.25 we show the operating qualities for a series of simulations, where the *t_ISG* parameter is varied for each simulation. The managed topology experiences operating qualities which increase as session arrival rates decrease. Hence, lesser demand allows fewer sessions to operate at higher qualities.

8.3.2 Timeliness of Management

Figures 8.26 and 8.27 below show the delay attributed to the Request/Allocation phases of the managed isochronous session.

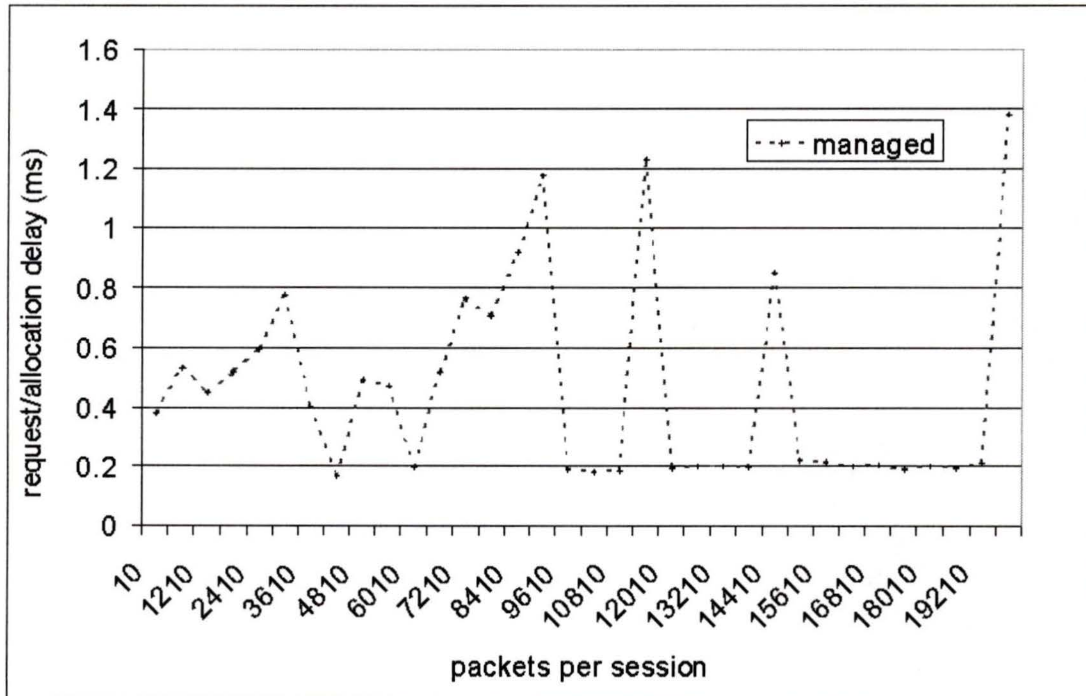


Figure 8.26: Req/Alloc Delay versus Number of Packets/Session

In Figure 8.26 we show the mean delay associated with the messaging latency[1] of the Request/Allocation phase (see Figure 8.5) for the managed session. This latency includes the time for: sending the request; computing the allocation; and finally, receiving the allocation.

---

1. For this measurement we do not include the timeout and retries due to the rejection of admission.

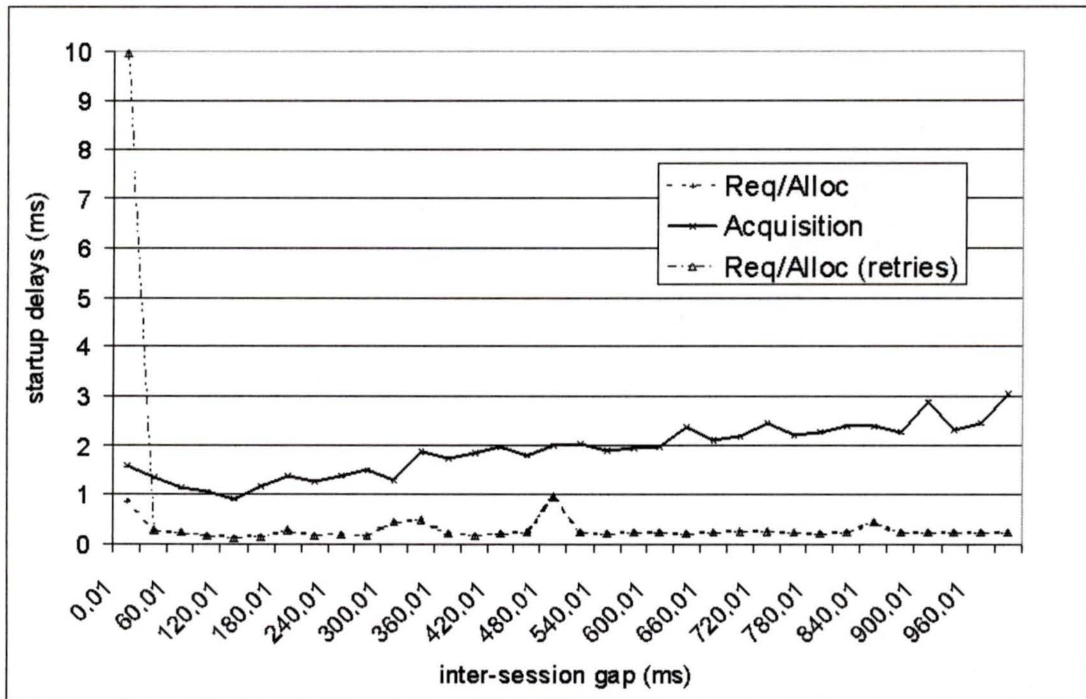


Figure 8.27: Startup delays versus Inter-Session Gap

In Figure 8.27 we show the mean delays associated with the Request/Allocation and Acquisition phases (see Figures 8.5, 8.6) of the managed session. We show three measurements: the Request/Allocation phase (i.e., Req/Alloc (retries)); the total messaging latency[1] of the Request/Allocation phase (i.e., Req/Alloc), and the Acquisition phase (i.e., Acquisition). We note that *Req/Alloc (retries)* is much longer that *Req/Alloc* only at the highest session arrival rates. This is because there are very few admission rejections as session arrival rates decrease.

1. For this measurement we do not include the timeout and retries due to the rejection of admission

8.3.3 Associated Costs of Management

Figures 8.28 and 8.29 below show the percentage of the entire session span (**t_session_c&d&a** for managed, and **t_session_c&d** for unmanaged) that is consumed by the data phase (**session_d**). We are illustrating the session data efficiency.

For the managed source, the phases: Request/Allocation, Acquisition, Deallocation, Release, and Adaptation add to the overhead and therefore reduce the session efficiency. For the managed source, the phases Acquisition, and Deallocation add to the session overhead.

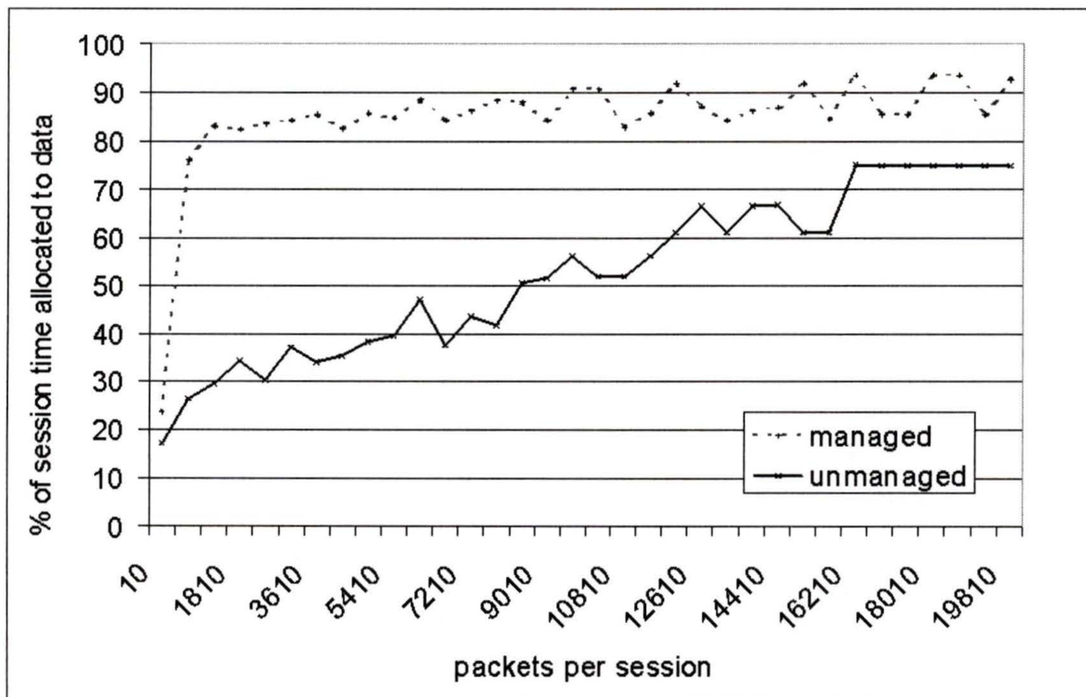


Figure 8.28: Session Data Efficiency versus Number of Packets/Session

In Figure 8.28 we show the efficiency for a series of simulations, where the *n_packets* parameter is varied for each simulation. For both MISRC and ISRC modules, the efficiency increases as the session data phase increases. Because of its shorter startup delay, we note that the managed session is generally more efficient than a unmanaged session. This is because the unmanaged session experiences many more reattempts to gain admission, compared to the managed session.

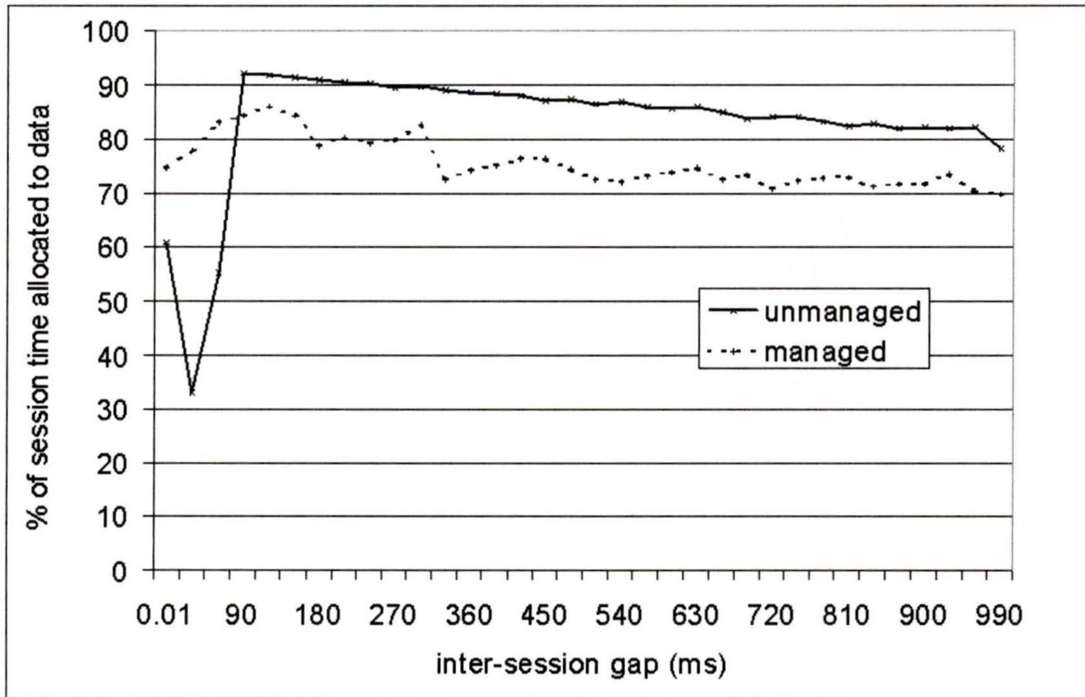


Figure 8.29: Session Data Efficiency versus Inter-Session Gap

In Figure 8.29 we show the efficiency for a series of simulations, where the *n_ISG* parameter is varied for each simulation. For both MISRC and ISRC modules, the efficiency increases as session arrival rates decrease: fewer admission rejections and adaptations (for the MISRC) and fewer timeout&retries when acquiring resources from the IRM (for the MISRC and ISRC) imply lower overheads. Because of its shorter startup delay, we note that the managed session is initially more efficient than the unmanaged session. However, the efficiency of the unmanaged session surpasses the managed session when resource contention on the unmanaged topology is less dominant. As the session arrival rate decrease, the efficiency, of both session types, gradually decreases as messages associated with the ISRC, MISRC, IRM, RMC, and RMS must compete for bus access against ASRC traffic.

The Figures 8.30 and 8.31 below show the mean number of adaptation requests per managed session. The following Figures, 8.32 and 8.33, show the mean delays for these adaptation/reacquisition phases. These phases are illustrated in Figures 8.7 and 8.8.

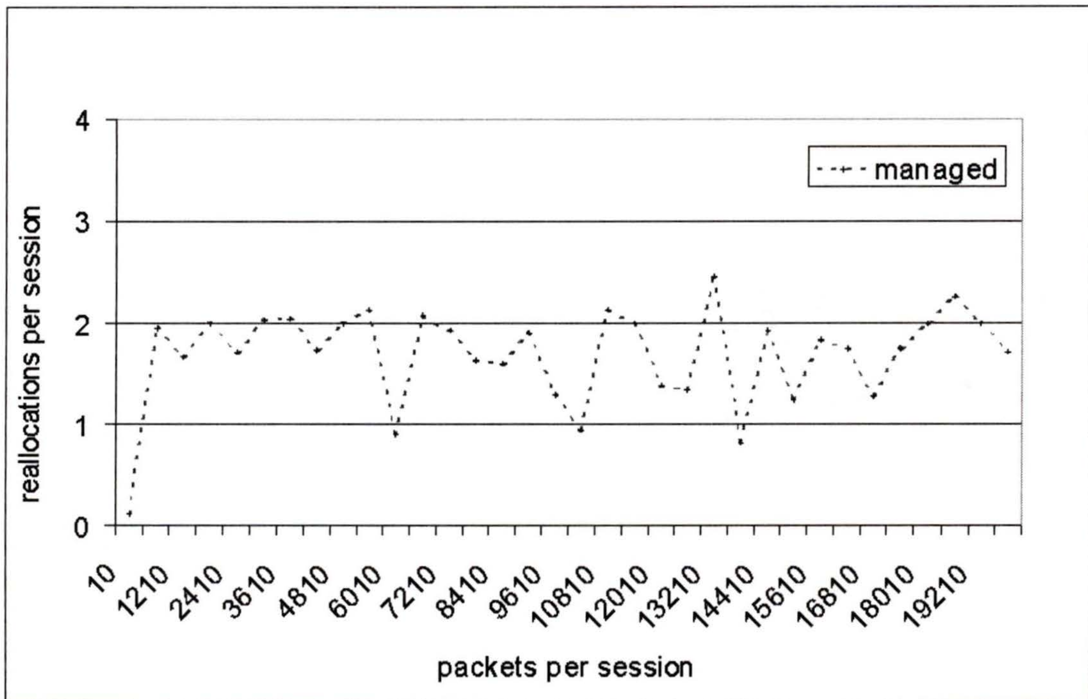


Figure 8.30: Reallocations/session versus Number of Packets/Session

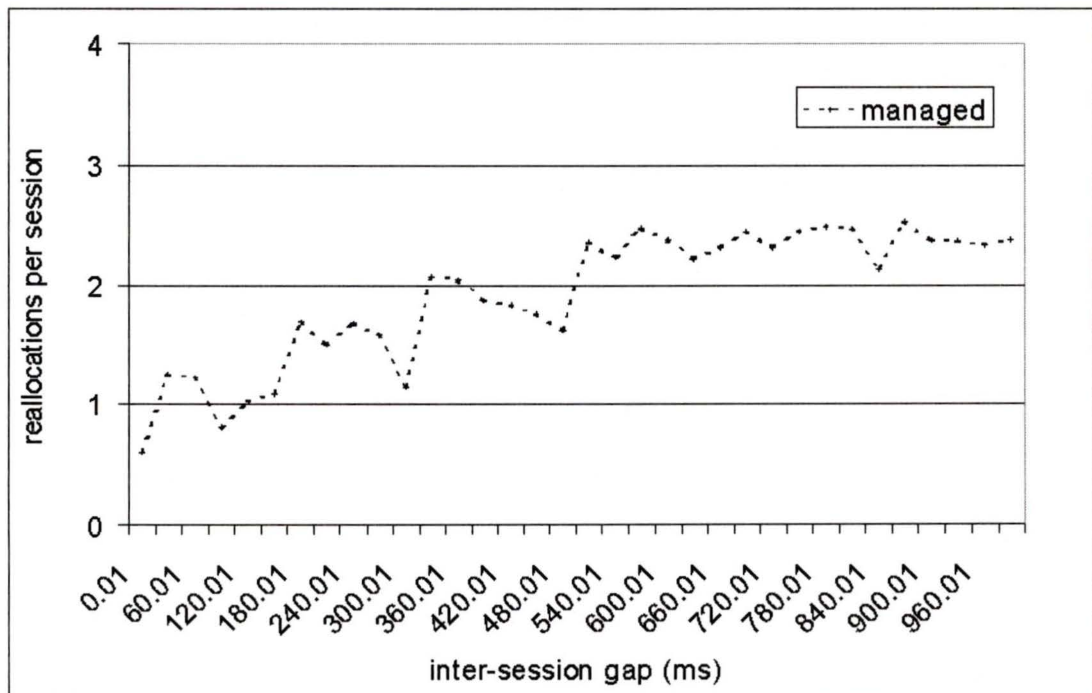


Figure 8.31: Reallocations/session versus Inter-Session Gap

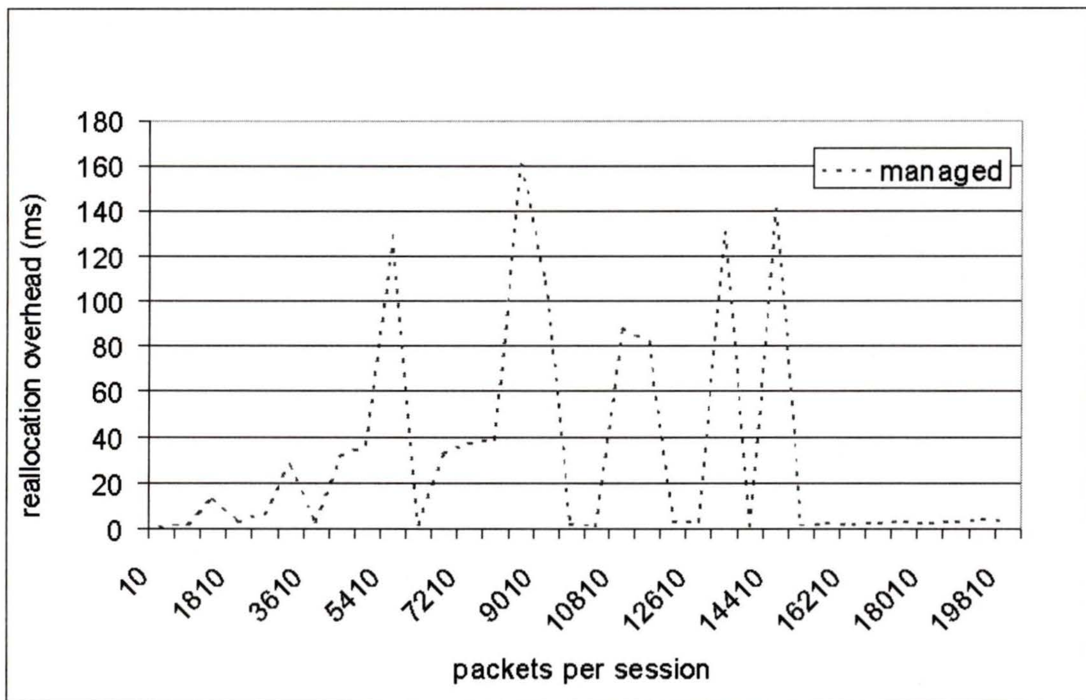


Figure 8.32: Reallocation Overhead versus Number of Packets/Session

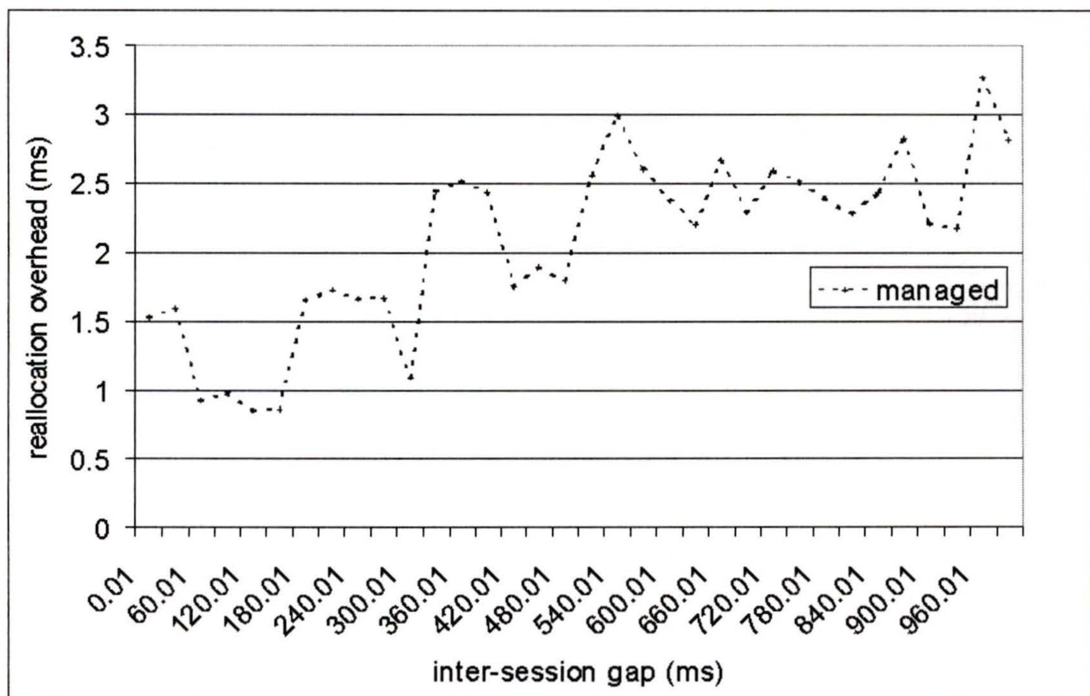


Figure 8.33: Reallocation Overhead versus Inter-Session Gap

Figures 8.34 below illustrates the overheads of messaging between the RMS and RMC modules on a managed topology.

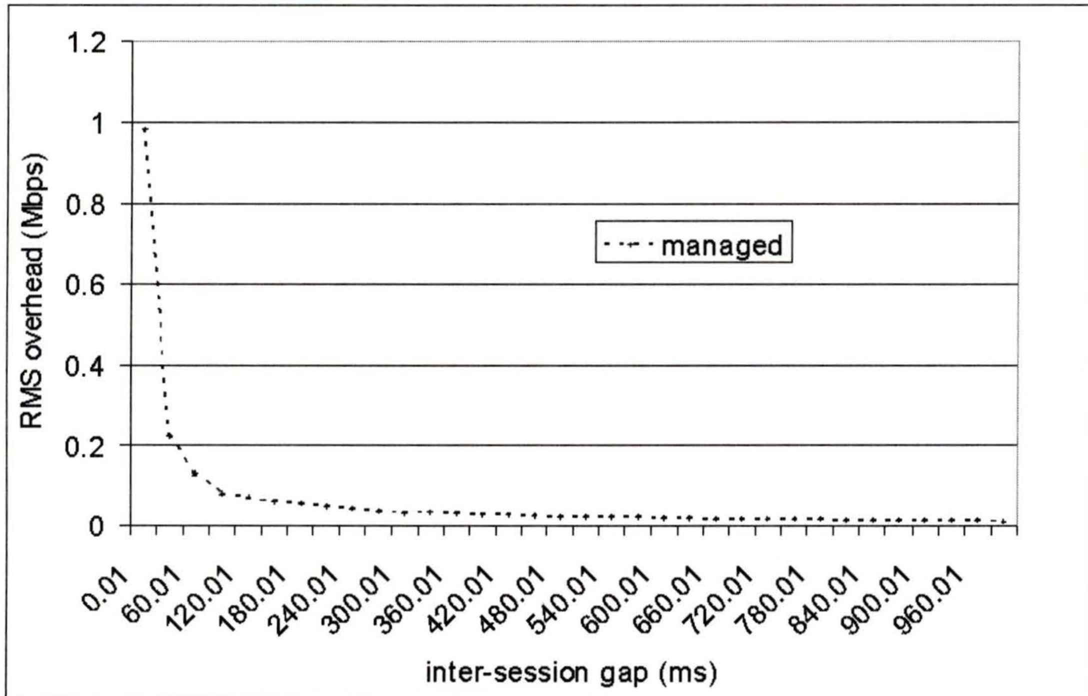


Figure 8.34: RMS Overhead versus Inter-Session Gap

The mean overhead is shown, where the *n_ISG* parameter is varied for each simulation. The decreasing overhead is consistent with the decreasing rate of session arrivals.

8.4 Summary of Results

From our simulations, we conclude with the following observations:

- Faster Session Admission - Managed sessions are admitted quicker than unmanaged sessions. Because the managed topology supports concurrent sessions, a lower mean average waiting time is experienced. See Figures 8.14, 8.15.
- Dynamic Support for Concurrent Sessions - Our dynamic management and adaptive sessions ensures that the maximum number of concurrent session can be supported, within the resource constraints of the network. See Figures 8.16, 8.24, 8.25.

- Lengthy Reallocation/Reacquisition Phases - Managed isochronous streams are halted each time a reallocation request is received (See Figures 8.7 and 8.8). Because an entire bandwidth allocation must be released so that an adjusted amount can be acquired, the resulting idle period can be lengthy (See Figure 8.32). This lengthy release/reacquisition phase could be avoided by a driver which implements incremental allocation/deallocation adjustments to bandwidth utilization. Hence, managed isochronous sessions would be uninterrupted by reallocation requests, only an adjustment to their utilization would be required.
- Low Overhead for RMS to RMC Messaging - A minimal amount of command and control information is passed between the RMC and RMS. See Figures 8.20 and 8.34 for evidence of the resulting traffic under loaded conditions.
- Fast Messaging between the RMS and RMC - Under loaded conditions, the delivery of management messages between the RMC and RMS is consistently fast enough to provide timely responses from the RMS. See Figures 8.26 and 8.27.
- Managed Sessions are more Efficient - Under scenarios of contention, managed sessions spend less time trying to gain admission. Therefore, the percentage of total session time associated with data transfer is higher for the managed session. See Figure 8.28.

9 Conclusion

9.1 Summary

At first glance, the bandwidth provided by the IEEE 1394 serial bus appears to be sufficient for any likely combination of audio-visual applications in the home environment. However, at closer examination, data stream rates for high-quality media can easily exceed 100 Mps; a few streams can easily saturate a 400 Mbps bus, therefore, prohibiting other users from access to the bus. A requirement for managing these scarce resources was identified.

As an approach to this lack of management, we proposed a centralized management scheme which allocates resources based on the individual requirements of applications along with the common management goals of the network. Applications would be required to adapt their behaviour to the allocation they receive. Khan's Utility Model provided us with a framework for our management scheme.

Our management architecture addressed the issues of providing management services over IEEE 1394. These included the definition of client (RMC) and server (RMS) modules, in addition to the design of messaging and election protocols. Also addressed, was the compatibility of managed devices (or applications) amongst legacy devices which do not subscribe to our management.

In order to test our architecture on a variety of 1394 topologies, we undertook the development of a 1394 simulation. A prototype of the IEEE 1394 - 1995 Standard was implemented with the OPNET network simulation environment. A driver API was

developed to provide services similar to those of the Windows 2000 1394 device driver. We validated the simulation by a implementing assertions throughout the FSMs of the 1394 model. These assertions, based on the 1394 Standard, were defined with respect to the time, order, and content of the events encountered within the process domain. Additionally, we studied timing data from Probe node.

Our management scheme was implemented on top of the IEEE 1394 OPNET simulation. The management modules, along with the managed and unmanaged applications were written to the API provided by the 1394 simulator. These applications were designed to yield a variety of asynchronous and isochronous traffic.

Simulations of separate managed and unmanaged topologies echoed our motivations for management. The unmanaged topology could only support a single session, whereas in the managed topology, our management services dynamically provided access for several concurrent sessions. Furthermore, our added management incurred minimal additional overheads in delay, and bandwidth for asynchronous messaging.

9.1.1 Uninterrupted Adaptive Streams

A managed isochronous source must go through several steps to adapt: first, the source must stop streaming isochronous packets; then it must release its current bandwidth allocation back to the IRM; the new allocation can then be acquired from the IRM; and finally, the source can restart the stream of isochronous packets. Alternatively, if the source was able to release, or acquire an adjustment between the old and new bandwidth allocations, it could continue to transmit, in the interim, at the quality bounded by the lowest of the new and old allocations. However, the *DR_IsochAllocateBandwidth* driver routine provided by the Windows platform, and implemented in our simulation, does not support the allocation or the deallocation of these adjustments. The driver routine accepts information about the requirements of the source stream and maps it to the required number of bandwidth allocation units; each call to this routine is assumed to be for a separate source. By extending this driver routine to support adjustments in session quality, the number asynchronous transactions targeted towards the IRM are reduced, and the isochronous stream is uninterrupted.

9.1.2 Management of Asynchronous Resources

Although, asynchronous transactions are not provided with the strict timing guarantees of isochronous transactions, they are supported by an arbitration scheme (discussed in Section 2.5) which guarantees fairness within the proportion of total bus bandwidth set aside for asynchronous transactions. The arbitration scheme determines which node gets access to the bus, however, it does not have control over size of asynchronous packets which are transmitted. Therefore, depending on the requirements of asynchronous transactions, the inter-access delay may be unacceptable, given the different priorities of the asynchronous sources.

We believe that our management scheme can also be applied to asynchronous traffic. By collecting the resource requirements from the asynchronous sources and including *asynchronous bandwidth* as a third resource, two management tasks could be accomplished: the bandwidth set-aside could be increased from the minimum 20% of total bus bandwidth; and the individual sources could be directed to operate at a specific level of quality - an assumption we currently make for isochronous sources. This would provide asynchronous transactions with a more deterministic service. The challenge is to determine an effective statistic for asynchronous requirements, so that a suitable amount of resources can be set-aside from the total bus bandwidth.

9.1.3 Implementation on a IEEE 1394 Test-bed

We developed the 1394 OPNET simulation environment to test our management concepts over larger 1394 topologies. The simulation results have proven our concepts to be viable, therefore, porting the OPNET implementation to an actual 1394 environment is a next logical step.

For future work, a small topology based on Intel Pentium class workstations, running Linux Operating System, and equipped with IEEE 1394 interface cards, would provide an effective test-bed for this development effort. The choice of Linux as the OS would provide access to the Linux IEEE-1394 (FireWire) Driver Development Project [10] and other Open Source projects which could be re-tooled to accommodate our approach to resource management.

References

[1] 100C/182/FDIS IEC 61883-1 Consumer audio/video equipment - Digital Interface.

[2] 100C/185/FDIS IEC 61883-4 Consumer audio/video equipment - MPEG-TS data transmission.

[3] IEEE P1394.1 Draft Standard for High Performance Serial Bus Bridges. Draft 0.08, May 9, 2000.

[4] IEEE Std 1212-1991, Command and Status Register (CSR) Standard.

[5] IEEE Std 1394-1995, IEEE Standard for a High Performance Serial Bus.

[6] D. Anderson. Firewire System Architecture: IEEE 1394, First Edition, *ISBN: 0-201-69470-0*. Addison-Wesley, Reading, Massachusetts, 1998.

[7] L. Chen, K.F. Li, M.S. Khan, E. Manning, "Building an Adaptive Multimedia System Using the Utility Model", *Parallel & Distributed Processing, Lecture Notes in Conputer Science 1586*, Springer Verlag, pp. 289-298, ISBN 2-540-65831-9.

[8] M. S. Khan, "Quality Adaptation in a Multi-session Multimedia System: Model, Algorithms and Architecture", *Ph.D. Dissertation*, Department of ECE, University of Victoria. May, 1998.

[9] Adam J. Kunzman and Alan T. Wetzel, "1394 High Performance Serial Bus: The Digital Interface for ATV", *IEEE Transactions on Consumer Electronics*, August 1995, Vol. 14, No. 13.

[10] Linux IEEE-1394 (FireWire) Driver Development Project. Available at: *http://linux1394.sourceforge.net.*

[11] Microsoft, Inc. IEEE 1394 Driver Development Kit documentation for Windows 2000. Available at *http://www.microsoft.com/ddk/ddkdocs/win2K/default.htm,* April 22, 2000.

[12] OPNET, Inc. OPNET Modeler Version 7.1 Modelling Concepts, Chapter 3 - Process Domain, Document D00114, ver. 2 , Washington DC, 2000.

[13] OPNET, Inc. OPNET Modeler Version 7.1 Simulation Kernel, Document D00154, ver. 1, Washington DC, 2000.

[14] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RFC 1889 (RTP: A Transport Protocol for Real-Time Applications), Internet Engineering Task Force, Audio-Video Transport Working Group", Mar. 1995. Available at: *http:// www.cs.columbia.edu/~hgs/rtp/rfc1889.ps.gz.*

[15] Sony, et.al. The HAVi Architecture - Specification of the Home Audio/Video Interoperability (HAVi) Achitecture. Version 1.0, Available at: *http:// www.havi.org/techiinfo/index.html*, January 18, 2000.

[16] Y. Toyoda, "A Simplified Algorithm for Obtaining Approximate Solution to Zero-one Programming Problems", *Management Sciences*. 21:1417-1427, 1975.

[17] Johan Walles, "On Capacity Utilization In IEEE-1394 Firewire", M.Sc. Thesis in Computer Science, Royal Institute of Technology, Stockholm, 1998. Available at: *http://www.student.nada.kth.se/~d92-jwa/exjobb/dok/rapport-en.doc*.

[18] Ingrid J. Wickelgren, "The Facts about Firewire", *IEEE Spectrum*, April 1997, Vol. 34, No. 9.

Appendix A IEEE 1394 Asynchronous Packet Formats

This Appendix contains the packets formats which support the IEEE 1394 asynchronous transactions. Packets associated with the Read, Lock, and Write transaction types are illustrated.

A.1 Support for the Asynchronous Read Transaction

The Asynchronous Read request and response packets are illustrated below. Both quadlet and block versions of the read transaction are given.

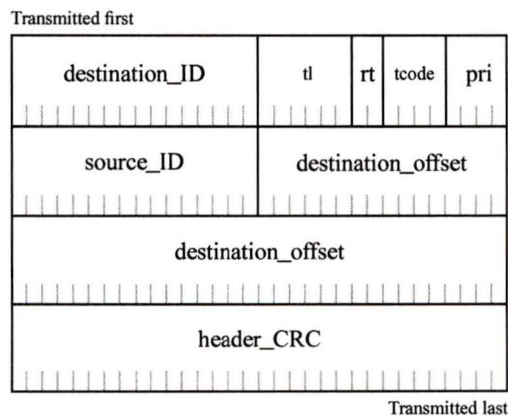


Figure A.1: Quadlet Read Transaction Request Packet

A description of each field in the lock response packet is given below in Table A.1.

Table A.1: Quadlet Read Transaction Request Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (e.g., quadlet read request).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
destination_offset	Destination Offset	Specifies the memory offset within the target node which is to be read.
header_CRC	Header CRC	CRC for the above header information.

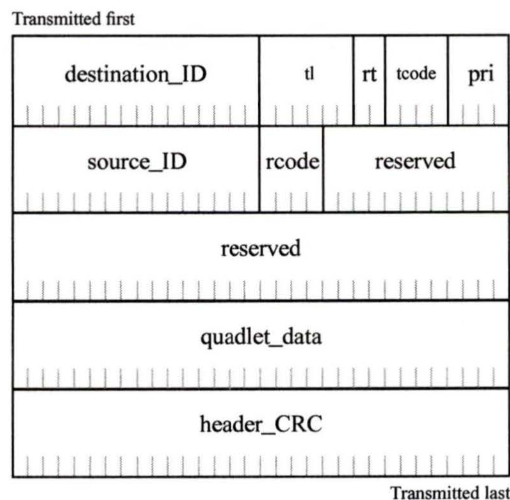


Figure A.2: Quadlet Read Transaction Response Packet

A description of each field in the read response packet is given below in Table A.2.

Table A.2: Quadlet Read Transaction Response Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (e.g., quadlet read response).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
rcode	Response Code	Specifies the success of the read transaction.
reserved	Reserved	Reserved in read request packets.
quadlet_data	Quadlet Data	The requested data.
header_CRC	Header CRC	CRC for the above header information.

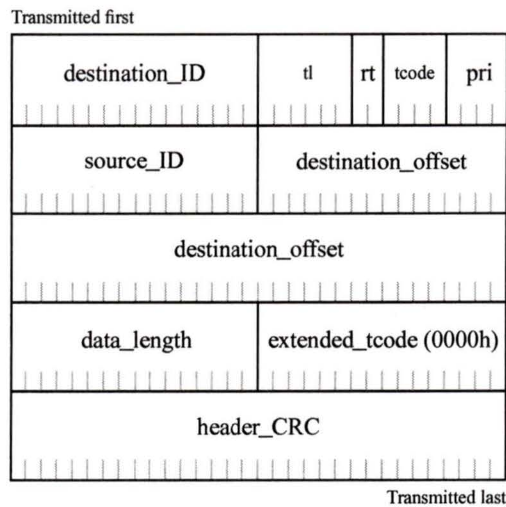


Figure A.3: Block Read Transaction Request Packet

A description of each field in the read request packet is given below in Table A.3.

Table A.3: Block Read Transaction Request Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (e.g., block read request).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).

Abbreviation	Field Name	Description
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
destination_offset	Destination Offset	Specifies the memory offset within the target node which is to be read.
destination_length	Data Length	Specifies the amount of data to be read at the destination offset of the destination node.
extended_code	Extended Transaction Code	Reserved during read request packets.
header_CRC	Header CRC	CRC for the above header information.

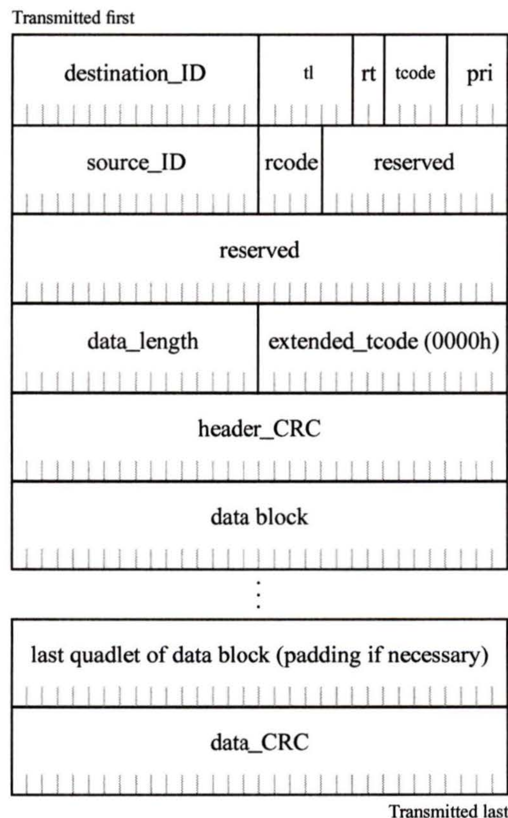


Figure A.4: Block Read Transaction Response Packet

A description of each field in the read response packet is given below in Table A.4.

Table A.4: Block Read Transaction Response Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (i.e., block read response).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
rcode	Response Code	Specifies the success of the read transaction.
reserved	Reserved	Reserved in read request packets.
data_length	Data Length	Size in bytes of requested data.
extended_code	Extended Transaction Code	Reserved in read request packets.
header_CRC	Header CRC	CRC for the above header information.
datablock	Data Block	Data from the target node, requested by the source node.
data_CRC	Data CRC	CRC for the data block.

A.2 Support for the Asynchronous Lock Transaction

The Asynchronous Lock request and response packets are illustrated below.

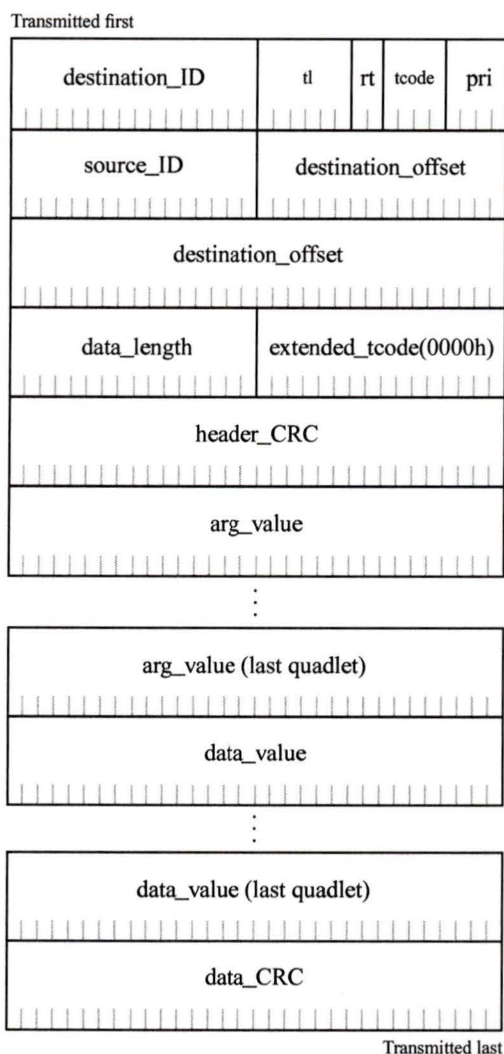


Figure A.5: Lock Transaction Request Packet

A description of each field in the lock request packet is given below in Table A.5.

Table A.5: Lock Transaction Request Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.

Table A.5: Lock Transaction Request Packet Fields

Abbreviation	Field Name	Description
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (i.e., lock request).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
destination_offset	Destination Offset	Specifies the memory offset within the target node which is to be updated.
data_length	Data Length	Specifies the total size of both arg_value and date_value fields.
extended_tcode	Extended Transaction code	Specifies which variation of the lock transaction that is being used (e.g. compare_swap).
header_CRC	Header CRC	CRC for the above header information.
arg_value	Argument Value	The original value at the destination_offset. This field is only used for compare_swap
data_value	Data Value	In the case of a compare_swap, this is the updated value at the destination offset. For a fetch_add operation, is represents the integer which is added to the value at the destination offset
data_CRC	Data Block CRC	The CRC value for the arg_value and date_value.

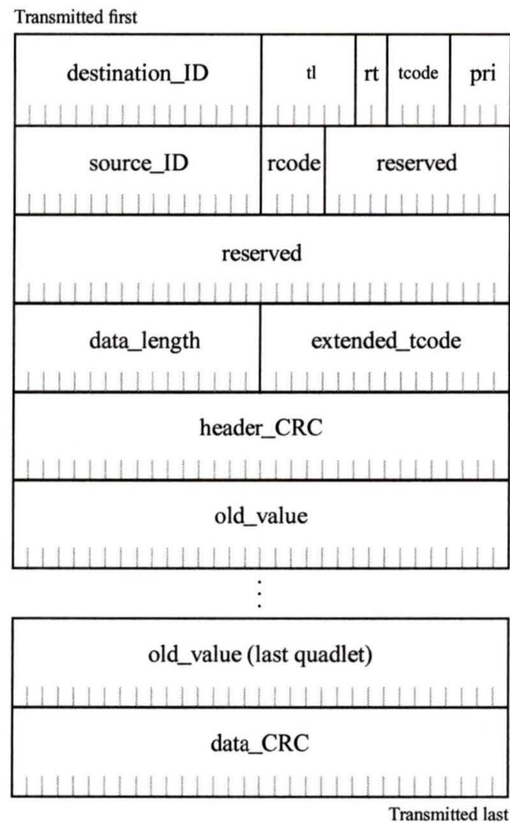


Figure A.6: Lock Transaction Response Packet

A description of each field in the lock response packet is given below in Table A.6.

Table A.6: Lock Transaction Response Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.

Abbreviation	Field Name	Description
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (i.e., lock response).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
rcode	Response Code	Specifies the success of the lock transaction.
data_length	Data Length	Specifies the total size of the old_value field.
extended_tcode	Extended Transaction code	Specifies which variation of the lock transaction that is being used (e.g. compare_swap).
header_CRC	Header CRC	CRC for the above header information.
old_value	Argument Value	The original value at the destination_offset. before an update or a failed update.
data_CRC	Data Block CRC	The CRC value for the old_value field.

A.3 Support for the Asynchronous Write Transaction

The Asynchronous Write request and response packets are illustrated below. Both quadlet and block versions of the write transaction are given.

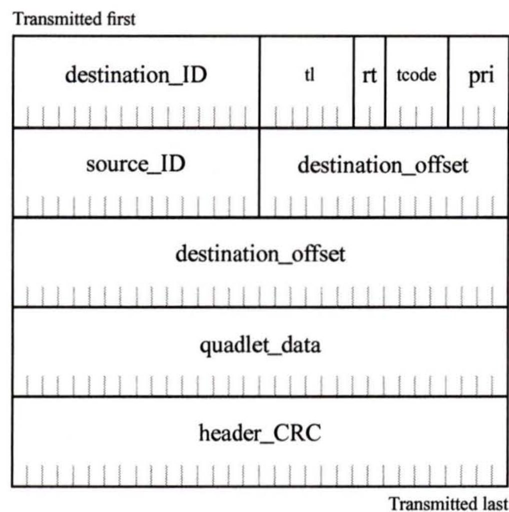


Figure A.7: Quadlet Write Transaction Request Packet

A description of each field in the quadlet write request packet is given below in Table A.7.

Table A.7: Quadlet Write Transaction Request Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (i.e., quadlet write request).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).

Abbreviation	Field Name	Description
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
destination_offset	Destination Offset	Specifies the memory offset within the target node which is to be read.
quadlet_data	Quadlet Data	The data to be written
header_CRC	Header CRC	CRC for the above header information.

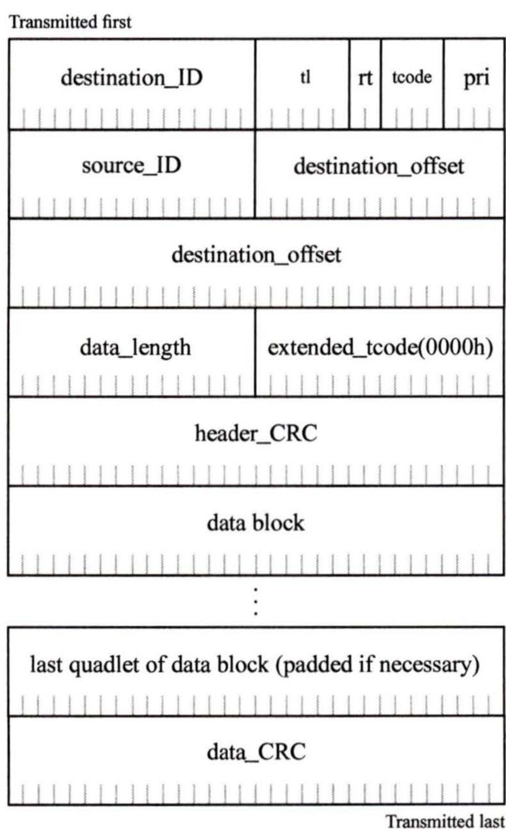


Figure A.8: Block Write Transaction Request Packet

A description of each field in the block write request packet is given below in Table A.8.

Table A.8: Block Write Transaction Request Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (i.e., block write request).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.
destination_offset	Destination Offset	Specifies the memory offset within the target node which is to be read.
destination_length	Data Length	Specifies the amount of data to be written at the destination offset of the destination node.
extended_code	Extended Transaction Code	Reserved during write request packets.
header_CRC	Header CRC	CRC for the above header information.
data block	Data Block	The data to be written at the destination offset of the destination node.
data_CRC	Data CRC	CRC for the data block.

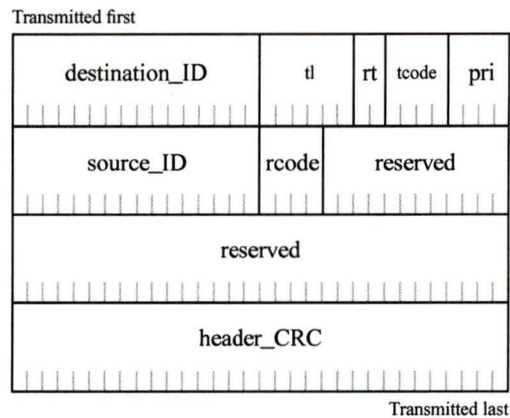


Figure A.9: Write Transaction Response Packet

A description of each field in the write response packet is given below in Table A.9.

Table A.9: Write Transaction Response Packet Fields

Abbreviation	Field Name	Description
destination_ID	Destination Identifier	Specifies the bus and node addresses of the destination node.
tl	Transaction Label	Defines a specific transaction. A response to a request has the same tl as the request.
rt	Retry code	Specifies whether the packet is a retry attempt, and defines which retry protocol is being used.
tcode	Transaction Code	Defines the transaction type (i.e., write response).
pri	Priority	Used in a backplane environment, not in a cable environment (i.e., a tree topology, as we assume).
source_ID	Source Identifier	Specifies the bus and node addresses of the source node.

Abbreviation	Field Name	Description
rcode	Response Code	Specifies the success of the write transaction.
reserved	Reserved	Reserved in write response packet.
header_CRC	Header CRC	CRC for the above header information.

Appendix B RMS Management Messages

This Appendix specifies the message formats used within our resource management protocol. We explain how our messages are carried by over a 1394 bus; illustrate the data structures used for messaging between RMC and RMS management components; and depict the packet formats for our messages.

B.1 Packaging Management Messages

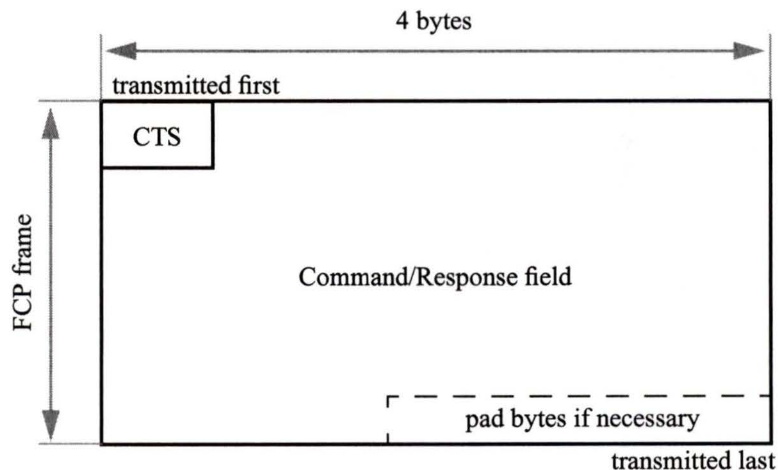


Figure B.1: FCP Packet Format

The IEC 61883-1 Standard [1] specifies the Functional Control Protocol (FCP) for supporting inter-device control on 1394 networks. The FCP defines a way of framing

commands and responses. This simple payload format is used for message exchange between the RMC and RMS components. The 4 bit Command Transaction Set (CTS) field in the above packet format is used to distinguish between different command sets which could be encountered over the bus. For our command set, we assume that the CTS field has been set to 0x1.

The above payload, shown in Figure B.1, is carried within a write request packet (both block and quadlet versions of the write request packet are illustrated in Appendix X).

B.2 Resource Management Message Formats

Below, we illustrate the formats used to support our resource management protocol. For each message type, we have defined a data structure, and a packet format which encapsulates the data structure. We consider the structure and the packet formats separately, because the structures are also used elsewhere in the management system.

A request message contains a specification of the required resources necessary to provide a range of operating qualities. Several data-types are used to encapsulate this information.

```
typedef struct {
      unsigned int ResLevel[MAXRESOURCES]; /* reqs for op-qual */
      unsigned int OppQualBid; /* utility or bid for op-qual */
} RESLEVEL, *PRESLEVEL;
```

Figure B.2: RESLEVEL Structure

The RESLEVEL structure, shown above in Figure B.2, specifies the resource requirements necessary to support a specific operating quality. Hence, it provides a resource mapping between for a single operating quality. The fields are as follows:

- `ResLevel[MAXRESOURCES]` - For a given operating quality, the required amount of each resource is specified in this integer array.
- `OppQualBid` - For a given operating quality, the Utility is specified by this integer.

The requirements and utility of an operating quality are specified within the above

structure. A quality profile is defined in the structure below:

```
typedef struct {
     RESLEVEL ResReq[MAXQOSLEVELS]; /* quality profile */
} QOSPROFILE, *PQOSPROFILE;
```

Figure B.3: QOSPROFILE Structure

The QOSPROFILE structure, shown above in Figure B.3, specifies the resource requirements and utility for each operating quality within the quality profile. The structure is defined as follows:

- ResReq[] - This array of RESLEVEL contains a quality profile, where level of quality increases with the index. For instance, ResReq[0] is defined as the NULL quality, and therefore, it does require any resources.

```
typedef struct {
     USHORT     OppQual;          /* selected quality */
     USHORT     AllocCode;        /* response code */
} RESALC, *PRESALC;
```

Figure B.4: RESALC Structure

The RESALC structure, shown above in Figure B.4, specifies the operating quality that is chosen by the RMS. If reallocations must be granted, the same structure is used. The fields are as follows:

- OppQual - The OppQual field is an index to the ResReq array which was passed within a resource request. This index represents the operating quality which the node can support, based on the allocated set of resources;
- AllocCode - The AllocCode field indicates the success of the operation, or an explanation for failure is returned.

B.3 Resource Management Packet Formats

Above, we have outlined the data structures which support our allocation protocol. In this section we illustrate the packet formats which encapsulate these structures.

The supported packet types are:

1. Resource request packet: Sent from the RMC to the RMS - encapsulates the requirements for the requesting process.
2. Resource allocation packet: Sent from the RMS to the RMC - specifies the operating quality at which the requesting process can operate.
3. Resource release packet: Sent from the RMC to the RMS - informs that the outstanding allocation has been released.

Each of these packet formats is illustrated below.

The format of the resource request message is illustrated below in Figure B.5. The message type, defined as *RMREQ*, is sent from the RMC to the RMS. This 68 byte packet encapsulates resource requirements and other policy-specific information. For this message type, the tcode field is set to 0x4. The entire message is encapsulated within a write block request packet payload.

Once the request packet has been received by the RMS node, an allocation of resources is yielded and then conveyed to the requesting RMC node.

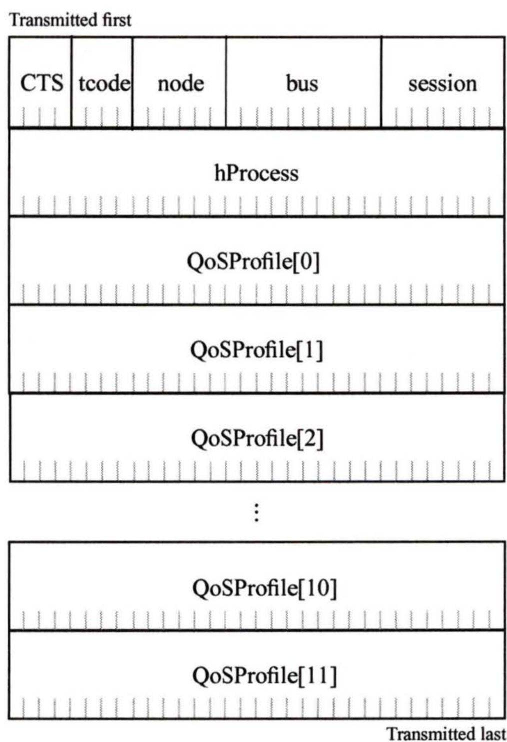


Figure B.5: Resource Request Packet (RMREQ)

A description of each field in the lock request packet is given below in Table 5.1.

Table B.1: Resource Request Packet Fields

Abbreviation	Field Name	Description
CTS	Command Transaction Set	Specifies the command set.
tcode	Transaction Code	Specifies the transaction. (i.e., 0x04 for a resource allocation request).
node	Node ID	Specifies the node ID of the client RMC.
bus	Bus ID	Specifies the bus ID of the client RMC.
session	Session	Specifies the ID of the session.
hProcess	Process ID	The ID of the requesting process.
QoSProfile	Quality Profile	Specifies the requirements and other policy specific information.

The format for the resource allocation packet is shown below in Figure B.6. This 12 byte packet is carried in the payload of a 1394 asynchronous write request packet and is sent from the RMS the an RMC. The message is only sent if the client's allocation has been modified. For this message type, known as RMALC, the tcode field is set to 0x5. The entire message is encapsulated within a write block request packet (as shown in Figure X) payload.

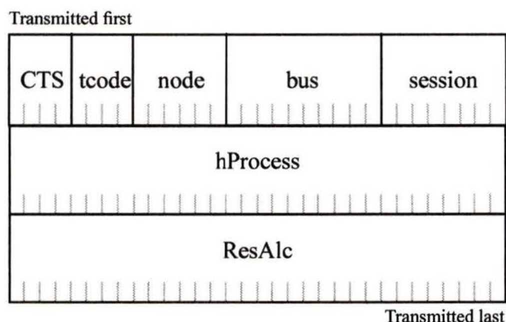


Figure B.6: Resource Allocation Packet (RMALC)

A description of each field in the resource allocation packet is given below in Table B.2.

Table B.2: Resource Allocation Packet Fields

Abbreviation	Field Name	Description
CTS	Command Transaction Set	Specifies the command set.
tcode	Transaction Code	Specifies the transaction. (i.e., 0x05 for a resource allocation).
node	Node ID	Specifies the node ID of the client RMC.
bus	Bus ID	Specifies the bus ID of the client RMC.
session	Sesssion	Specifies the ID of the session.
hProcess	Process ID	The ID of the process receiving the allocation.
ResAlloc	Resource Allocation	Specifies the process from which the resource request originated.

A managed node releases an allocation by first sending a resource release message to the RMS, and then releasing the resources through the IRM. For this message type, the tcode field is set to 0x6. There is no other information other than the standard IDs for the bus, node, and process.

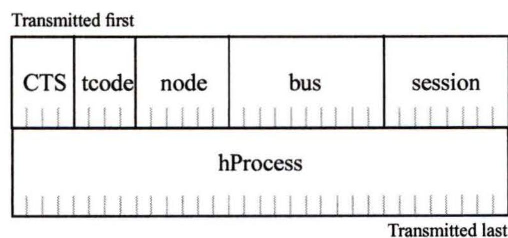


Figure B.7: Resource Release Packet

A description of each field in the resource release packet is given below in Table B.3

Table B.3: Resource Release Packet Fields

Abbreviation	Field Name	Description
CTS	Command Transaction Set	Specifies the command set.
tcode	Transaction Code	Specifies the transaction. (e.g., 0x06 for a resource release).
node	Node ID	Specifies the node ID of the client RMC.
bus	Bus ID	Specifies the bus ID of the client RMC.
session	Session	Specifies the ID of the session.
hProcess	Process ID	The ID of the process receiving the allocation.

Appendix C Resource Management Client

C.1 RMC Architecture

The modules which compose the RMC are illustrated in this Appendix. The APIs for each module (shown below in Figure C.1) are outlined. Then, examples of the inter-module collaborations are presented.

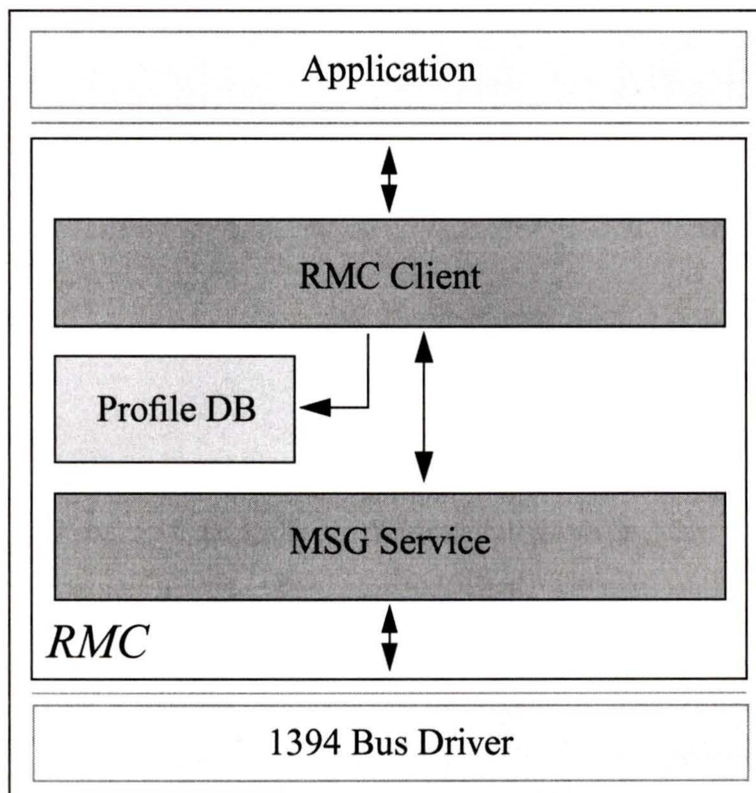


Figure C.1: RMC Modules

C.1.1 RMC Client Module

The RMC is a proxy for the RMS. Local applications call the RMC to determine the quality level which they can accommodate.

The RMC API is listed below.

RMC::Request

Prototype

Status RMC::Request(in RMREQ *RMReq*)

Parameters

- *RmReq* - A structure (defined in Appendix B) containing request parameters.

Description

Request specifies the resource requirements for the session. This routine adds the request information to the RMC PDB and then use the MSGS to pass the request to the RMS node.

RMC::Release

Prototype

Status RMC::Release(in RMREL *RMRel*)

Parameters

- *RMRel* - A structure (defined in Appendix B) containing release parameters.

Description

Release is used to inform the RMC and RMS that previously acquired resources are no longer required, and that they are no longer being utilized. This routine first updates the client information in the PDB table and then send a release request to the RMS node via the MSGS Service module.

RMC::Receive

Prototype

Status RMC::Receive(in REQMSG ReqMsg)

Parameters

- *ReqMsg* - A management message.

Description

Receive is callback which is called by the MSG Service upon receiving a management message.

<Client>::Acquire

Prototype

Status <Client>::Acquire(in RMALC *RMAlc*)

Parameters

- *RMAlc* - A structure (defined in Appendix B) containing allocation information.

Description

Acquire is a callback which conveys quality information to the application client. The client is obligated to adapt its quality upon receiving this message.

C.1.2 MSG Service

The management messages passed between RMS and RMC modules must be packaged into write request packets and then transmitted using a write asynchronous transaction. And then, upon reception, the messages must be extracted from the packet payload. Furthermore, the register (or target memory location) into which the messages are written, must be configured before messages can be successfully received. The MSG module fulfils these requirements.

The MSG module is also responsible for determining the address of the elected RMS. The MSG module attempts to access and read a well-known register on the IRM node (details in Section 5.2.2) to determine the address of the RMS node.
The API for MSG is listed below.

MSG::Send

Prototype

Status MSG::Send(in REQMSG ReqMsg)

Parameters

- *ReqMsg* - A management message.

Description

Send is called by the RMC Client module for the purpose of forwarding a message to the RMS node. The message is packetized and sent to the RMS node by invoking a write request.

MSG::Receive

Prototype

REQMSG MSG::Receive(in WRRQPKT WRReqPkt)

Parameters

- *WRReqPkt* - A write request packet (as defined in Appendix A) encapsulates a resource management message.

Description

Receive is called when packet payload has arrived. The management message is extracted from the payload and passed to the RMC Client.

C.1.3 RMC Profile DB

The RMC can support multiple clients, for instance: multiple applications and multiple sessions for each application. To provide a degree of transparency to the supported applications, the RMC filters out stale management messages; the DB contents are used to determine the validity of a session. The RMC Profile DB (PDB) maintains a record for each session which is initiated at the node. The format of this record is illustrated below in Figure C.2.

```
typedef struct {
     ULONG      hProcess;        /* Process ID of client     */
     QOSPROFILE QoSProfile;      /* QoS profile of client    */
     RESALC     ResAlc;          /* Current allocation       */
     ULONG      pFlags;          /* Misc. flags              */
     USHORT     nSession;        /* Session ID               */
} RMCPDBREC, *PRMCPDBREC;
```

Figure C.2: RMC Profile DB Record

The API for the RMC Profile DB is listed below.

PDB::Add

Prototype

Status DB::Add(in RMREQ *RMReq*)

Parameters

- *RMRel* - A structure (defined in Appendix B) containing release parameters.

Description

Add inserts a new *RMSPDBREC* record into the PDB table, containing resource requirement and client information.

PDB::Get

Prototype

RMCPDBRec* PDB::Delete(in ObjectID *hProcess*)

Parameters

- *hProcess* - An identifier for the client.

Description

Retrieves the client requirements and allocation information from the PDB table.

PDB::Delete

Prototype

Status PDB::Delete(in ObjectID *hProcess*)

Parameters

- *hProcess* - An identifier for the client.

Description

Delete removes the client requirements and allocation information from the PDB table.

C.2 RMC Operational Scenarios

C.2.1 A Resource Request

This scenario traces the events that occur when an application makes a request via the

RMC. These illustrated events are local to the RMC. This scenario can be viewed from the RMS perspective in Appendix D. The inter-module calls during this operation are illustrated below in Figure C.3.

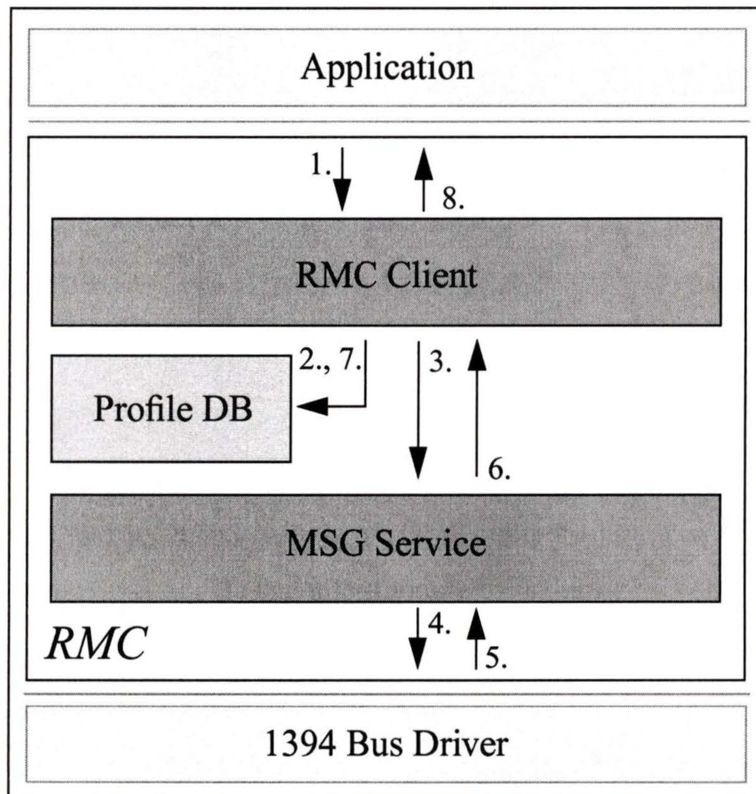


Figure C.3: Resource Request Scenario

1. Application –> RMC Client
   An application, that is supported by the RMC, calls RMC::Request() to initiate a request for an allocation.

2. RMC Client –> Profile DB
   To add the session information to the Profile DB, PDB::Add() is called.

3. RMC Client –> MSG Service
   The RMC Client module assembles a management message (RMREQ, defined in Appendix B) for the specific client and then passes the message to the Message Service module using MSG::Send().

4. MSG Service -> 1394 Driver
   The MSG Service module packages the management message into a payload and calls

DR_AsyncWrite() (defined in Appendix E) to send the management message to the relevant node.

5. 1394 Driver -> MSG Service
   A message is received from the RMS. MSG::Receive() is called by the 1394 driver upon receiving a asynchronous write request within the memory space of the RMC command register. The payload of the write request is passed to the MSG Service module, where the management message is extracted.

6. MSG Service –> RMC Client
   RMC::Receive() is called by the MSG Service module. The management message (either RMALC, defined in Appendix B) is passed to the RMC Control Module.

7. RMC Client -> Profile DB
   The allocation information is save in the Profile DB module. PDB::Get() is called to acquire the record, then the record is updated.

8. RMC Client –> Application
   The RMC Client calls <Client>::Acquire() to pass the received allocation information. Based on the allocation information received, the application must acquire resources from the IRM.

C.2.2 An Adaptation Indication

This scenario traces the events that occur when the RMS sends a re-allocation message and it is received and processed by the RMC. The illustrated events are local to the RMC. This scenario can be viewed from the RMS perspective in Appendix D. The inter-module calls during this operation are illustrated below in Figure C.4.

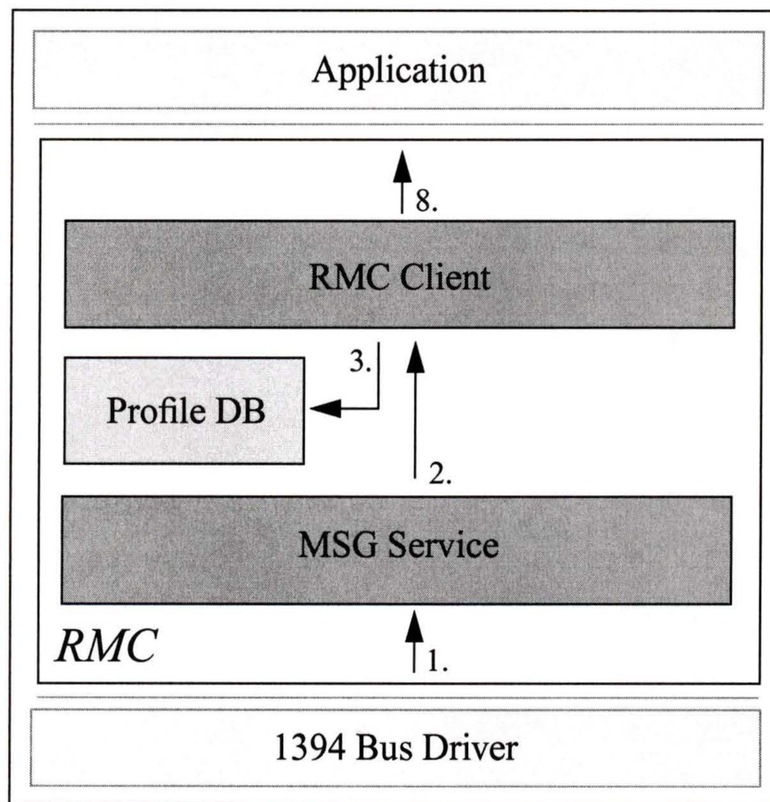


Figure C.4: Adaptation Indication Scenario

1. 1394 Driver -> MSG Service
   An message is received from the RMS. MSG::Receive() is called by the 1394 driver upon receiving a asynchronous write request targeted at the memory space of the RMC command register. The payload of the write request is passed to the MSG Service module, where the management message is extracted.

2. MSG Service –> RMC Client
   RMS::Receive() is called by the MSG Service module. The management message RMALC, defined in Appendix B) is passed to the RMC Client module.

3. RMC Client –> Profile DB
   The reallocation information is saved in the Profile DB module. PDB::Get() is called to acquire the record, then the record is updated.

4. RMC Client –> Application
   The RMC->Client calls <Client>::Acquire() to pass the received reallocation information. Based on the reallocation information received, the application must adapt by releasing or acquiring resources from the IRM.

C.2.3 A Release Request

This scenario traces the events which occur after the application releases its resources, and when it informs the RMC of the release. The illustrated events are local to the RMC. This scenario can be viewed from the RMS perspective in Appendix D. The inter-module calls during this operation are illustrated below in Figure C.5.

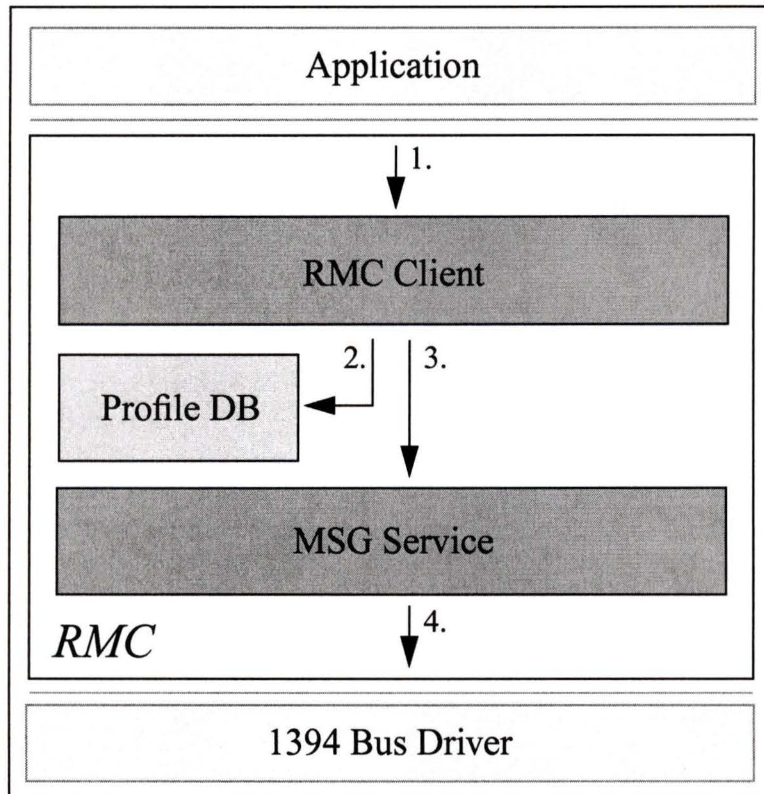


Figure C.5: Release Request Scenario

1. Application –> RMC Client
   An application, that is supported by the RMC, calls RMC::Release() to inform the RMS that it has released the resource associated with its allocation.

2. RMC Client –> Profile DB
   To delete the session information from the Profile DB, PDB::Delete() is called.

3. RMC Client –> MSG Service
   The RMC Client module assembles a management message (RMREL, defined in Appendix B) for the specific and then passes the message to the Message Service module using MSG::Send().

4. MSG Service –> 1394 Driver
   The MSG Service module packages the management message into a payload and calls DR_AsyncWrite() (defined in Appendix E) to send the management message to the relevant node.

Appendix D Resource Management Server

D.1 RMS Architecture

The modules which compose the RMS are illustrated in this Appendix. The APIs for each module (shown below in Figure D.1) are outlined. Then, examples of the inter-module collaborations are presented.

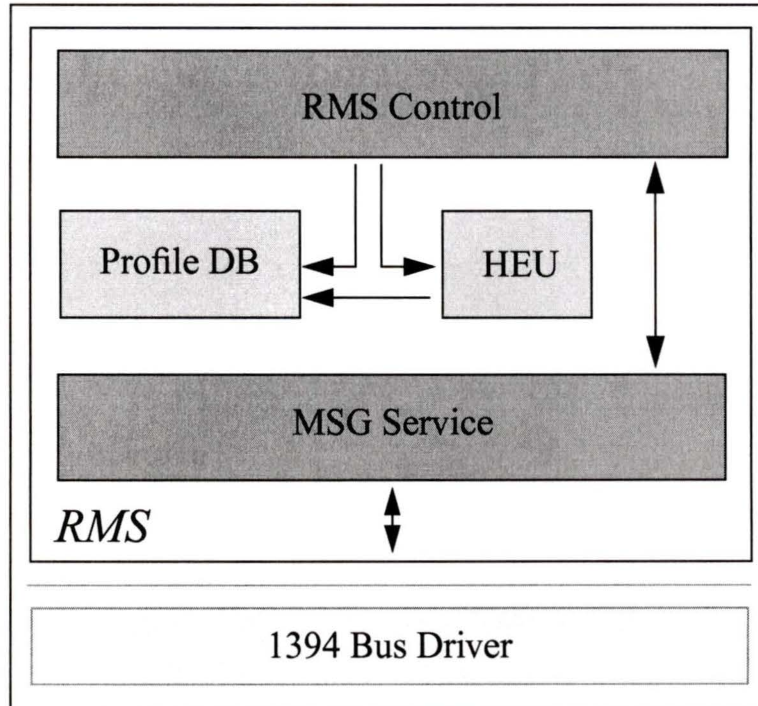


Figure D.1: RMS Modules

D.1.1 RMS Control

The RMS Control module is responsible for orchestrating the events which occur as a request arrives from an RMC. The RMS Control module is only called by the MSG Service module - passing along management messages.
The API for MSG is listed below.

RMS::Receive

Prototype

Status RMS::Receive(in REQMSG ReqMsg)

Parameters

- *ReqMsg* - A management message.

Description

Receive is callback which is called by the MSG Service upon receiving a management message.

D.1.2 MSG Service

The management messages passed between RMS and RMC modules must be packaged into write request packets and then transmitted using a write asynchronous transaction. And then, upon reception, the messages must be extracted from the packet payload. Furthermore, the command register (or target memory location) into which the messages are written, must be configured before messages can be successfully received. The MSG module fulfils these requirements.

The MSG module is also responsible for the post-reset RMS election. Each bus-resident and RMS-capable node is represented by a MSG module that attempts to access and update a well-known register on the IRM node (details in Section 5.2.1). The first node to modify the register wins the election. Each MSG module which losses the election should not receive any management messages, and therefore, remains dormant.
The API for MSG is listed below.

MSG::Send

Prototype

Status MSG::Send(in REQMSG ReqMsg)

Parameters

- *ReqMsg* - A management message.

Description

Request is called by the RMS Control module for the purpose of forwarding a message to the RMS node. The message is packetized and sent to the RMS node by invoking a write request.

MSG::Receive

Prototype

REQMSG MSG::Receive(in WRRQPKT WRReqPkt)

Parameters

- *WRReqPkt* - A write request packet (as defined in Appendix A) encapsulates a resource management message.

Description

Receive is called when packet payload has arrived. The management message is extracted from the payload and passed to the RMS control module.

D.1.3 RMS Profile DB

The RMS Profile DB (PDB) maintains a single table of information including resource requirements and resulting allocations for all sessions hosted by the RMCs within the network. The format of this record is illustrated below in Figure D.2.

```
typedef struct {
     USHORT     nNodeID;        /* Node ID of client        */
     USHORT     nBusID;         /* Bus ID of client         */
     ULONG      hProcess;       /* Process ID of client     */
     QOSPROFILE QoSProfile;     /* QoS profile of client    */
     RESALC     ResAlc;         /* Current allocation       */
     ULONG      pFlags;         /* Misc. flags              */
     USHORT     nSession;       /* Session ID               */
} RMSPDBREC, *PRMSPDBREC;
```

Figure D.2: RMS Profile DB Record

PDB::Add

Prototype

Status DB::Add(in RMREQ *RMReq*)

Parameters

- *RMRel* - A structure (defined in Appendix B) containing release parameters.

Description

Add inserts a new *RMSPDBREC* record into the PDB table, containing resource requirement and client information.

PDB::Get

Prototype

RMSPDBRec* PDB::Get(in ObjectID *hProcess*)

Parameters

- *hProcess* - An identifier for the client.

Description

Get retrieves the client requirements and allocation information from the PDB table.

PDB::GetSize

Prototype

USHORT PDB::GetSize()

Parameters

None

Description

GetSize returns the number of records in the PDB table.

PDB::GetRecN

Prototype

RMSPDBRec* PDB::GetRecN(in ULONG *RecN*)

Parameters

- *RecN* - The record sequence number 0..(Size-1).

Description

GetRecN retrieves the record by the sequence number.

PDB::GetFlag

Prototype

RMSPDBRec* PDB::GetFlag(in ULONG *flags*)

Parameters

- *flags* - An identifier for the client.

Description

GetFlag retrieves the first client record whose *pFlags* field meets the conditions specified by the *flags* parameter.

PDB::Delete

Prototype

Status PDB::Delete(in ObjectID *hProcess*)

Parameters

- *hProcess* - An identifier for the client.

Description

Delete removes the client requirements and allocation information from the PDB table.

D.1.4 HEU Module

The HEU module is responsible for calculating a near-optimal set of operating qualities for the sessions resident within the RMS PDB. The HEU module extracts the session information records from the RMS PDB. After solving the allocation problem, the HEU updates the records which correspond to the affected sessions.

HEU::Load

Prototype

Status HEU::Load( )

Parameters

None

Description

Load directs the HEU module to collect records from the RMS PDB, and then initiate its own data structures with the session profile, resource mapping, and utility information.

HEU::Solve

Prototype

Status HEU::Solve( )

Parameters

None

Description

Load directs the HEU module to execute the heuristic algorithm and yield a solution vector which specifies the operating qualities of each session.

HEU::Save

Prototype

Status HEU::Save( )

Parameters

None

Description

Save transfers the HEU computation results back to the RMS PDB.

D.2 RMS Operational Scenarios

D.2.1 Resource Request/Release

The RMS can expect both *request* and *release* messages from any RMC. Due to the

similarities of these operations, we have illustrated the processing of both messages types in one scenario, shown below in Figure D.3.

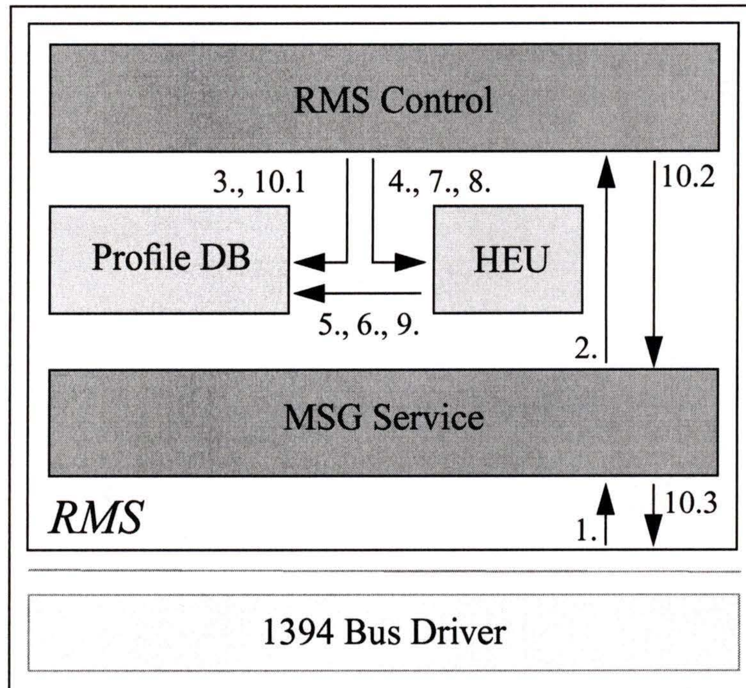


Figure D.3: Request/Release Indication Scenario

1. 1394 Driver –> MSG Service
   An message is received from a RMC. MSG::Receive() is called by the 1394 driver upon receiving a asynchronous write request targeted at the memory space of the RMS command register. The payload of the write request is passed to the MSG Service module, where the management message is extracted.

2. MSG Service –> RMS Control
   RMS::Receive() is called by the MSG Service module. The management message (either RMREQ or RMREL, as defined in Appendix B) is passed to the RMS Control module.

3. RMS Control –> Profile DB (request or release)
   Depending upon the type of message, a session definition could be added or deleted from the Profile DB module.
   For a *request*: PDB::Add() is called.
   For a *release*: PDB::Delete() is called.

4. RMS Control –> HEU

The RMS Control module calls HEU::Load() to initialize the HEU module.

5. HEU –> Profile DB
   The HEU module calls PDB::GetSize() to determine the number of records on the Profile DB.

6. HEU –> Profile DB[1]
   The HEU module calls PDB::GetRecN() to retrieve each record by its index.

7. RMS Control –> HEU
   The RMS Control module calls HEU::Solve() to yield a solution.

8. RMS Control –> HEU
   The RMS Control module calls HEU::Save() to update the records in the Profile DB with the new allocations yielded in the previous step.

9. HEU –> Profile DB[2]
   The HEU module calls PDB::GetRecN() to retrieve each record by its index and update its allocation information. Each session with a modified allocation has its flags set.

10. Distribute allocations[3]
    The next three steps distribute new/modified allocation information to a client.

    10.1 RMS Control –> Profile DB
    RMS Control calls PDB::GetFlag() to get the next record which has a flag set that indicates a modified allocation.

    10.2 RMS Control –> MSG Service
    RMS Control assembles a management message (RMALC, as defined in Appendix B) for the specific client and then passes the message to the Message Service module using MSG::Send(). Then the flag on the client record is cleared.

    10.3 MSG Service –> 1394 Driver
    The MSG Service module packages the management message into a payload and calls DR_AsyncWrite() (defined in Appendix E) to send the management message to the relevant node.

---

1. Operation is called multiple times, once for each record in the Profile DB.
2. Operation is called multiple times, once for each record in the Profile DB.
3. Operation is called multiple times, once for each session whose allocation has changed.

Appendix E OPNET IEEE 1394 API

This Appendix specifies the API for the IEEE 1394 OPNET model, described in Chapter 6. The API is used by applications illustrated in Chapter 7. The procedures in this API were influenced by the Microsoft Windows 2000 1394 Device Driver Kit [11].

E.1 Asynchronous API

```
DRSTATUS DR_AsyncRead (
        IO_ADDRESS DestinationAddress,    /* bus, node, offset     */
        ULONG      nNumberOfBytesToRead,  /* Size to read          */
        ULONG      nBlockSize,            /* Fragmentation size    */
        ULONG      fulFlags,              /* Reserved              */
        HANDLE     pBlock                 /* Destination block     */
)
```

Purpose: Performs an asynchronous read transaction.

Description: DR_AsyncRead fetches a block of memory from a device on the 1394 bus. The requested block of memory must be within an available range of addresses. Currently, each node offers the range of 0x000000000000 to 0x000000000400.

The call requires the IO_ADDRESS structure *DestinationAddress* which specifies the bus, node, and destination offset addresses of the target node. The number of bytes to be read are specified by the *nNumberOfBytesToRead* parameter. The *fulFlags* parameter is

reserved for future use. The *pBlock* parameter is a handle to the data block to which the received data is copied. The data block must be at least *nNumberOfBytesToRead* bytes in size. The *nBlockSize* parameter sets the maximum size of an asynchronous packet payload, hence the driver routine could make multiple transaction level calls to receive the requested data block. For simplicity, we currently assume that (nNumberOfBytesToRead == nBlockSize), hence one transaction level call for each invocation of this routine. Therefore, nBlockSize must not exceed the maximum packet size for the chosen bus speed.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: An error associated with the transaction layer.

```
DRSTATUS DR_AsyncWrite (
        IO_ADDRESSDestinationAddress,       /* bus, node, offset    */
        ULONG     nNumberOfBytesToWrite,    /* Size to write        */
        ULONG     nBlockSize,               /* Fragmentation size   */
        ULONG     fulFlags,                 /* Reserved             */
        HANDLE    pBlock                    /* Destination block    */
)
```

Purpose: Performs an asynchronous write transaction.

Description: DR_AsyncWrite writes a block data to an area of memory on a target device. The target memory must be within an available range of addresses. Currently, each node offers the range of 0x000000000000 to 0x000000000400.

The call requires the IO_ADDRESS structure *DestinationAddress* which specifies the bus, node, and destination offset addresses of the target node. The number of bytes to be written are conveyed by the *nNumberOfBytesToWrite* parameter. The *fulFlags* parameter

is reserved for future use. The *pBlock* parameter is a handle to the source data block. The data block must be at least *nNumberOfBytesToWrite* bytes in size. The *nBlockSize* parameter sets the maximum size of an asynchronous packet payload, hence the driver routine could make multiple transaction level calls to write the requested data block. For simplicity, we currently assume that (nNumberOfBytesToWrite == nBlockSize), hence one transaction level call for each invocation of this routine. Therefore, nBlockSize must not exceed the maximum packet size for the chosen bus speed.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: An error associated with the transaction layer.

DRSTATUS DR_AsyncLock (

```
        IO_ADDRESSDestinationAddress,/* bus, node, offset              */
        ULONG      nDataLength,         /* Size of both argument and data*/
        ULONG      fulTransactionType,  /* Type of Lock transaction      */
        ULONG      fulFlags,            /* Reserved                      */
        HANDLE     pArgumentValue,      /* Source argument_value         */
        HANDLE     pDataValue,          /* Source data_value             */
        HANDLE     pBlock               /* Destination for old_value     */
)
```

Purpose: Performs an asynchronous lock transaction.

Description: DR_AsyncLock modifies a block data on a target device. The target memory must be within an available range of addresses. Currently, each node offers the range of 0x000000000000 to 0x000000000400.

The call requires the IO_ADDRESS structure *DestinationAddress* which specifies the bus, node, and destination offset addresses of the target node. The *nDataLength* parameter

specifies the size of both argument and data parameters in the case of the compare_swap transaction. The nDataLength parameter specifies only the size of the data parameter in the case of a fetch_add transaction. And the *fulFlags* parameter is reserved for future use. The *fulTransactionType* parameter specifies the type of Lock transaction request:

1. "compare_swap"

The "compare_swap" transaction is used when a memory block must be modified. As several processes may attempt to modify the same target block, inconsistencies may result. The "compare_swap" transaction requires two data parameters: the "Argument Value" (referenced by *pArgumentValue*), and the "Data Value" (referenced by *pDataValue*). Presumably, the value of the target block has previously been read and then modified. When the target node executes a "compare_swap" transaction, it verifies that the target memory block is equal to the contents of *pArgumentValue*. If both are equal, then the value of the target block is updated with the contents of the *pDataValue* parameter. Otherwise, the update is not committed.

The parameter *nDataLength* specifies the total length of both *pArgumentValue* and *pDataValue* parameters. Both of the parameters must be either 4 or 8 bytes in length.

Upon completion of the transaction, the pre-update contents of the target memory block are copied into the data block referenced by the *pBlock* parameter.

2. "fetch_add"

The "fetch_add" transaction adds the value of the "Data Value" parameter (referenced by *pDataValue*) to the target memory block. This transaction does not use the "Argument Value" parameter, therefore, the value of *nDataLength* only refers to the length of the "Data Value" parameter. The pointer *pDataValue* must reference a block of either 4 or 8 bytes.

When using both variations of the lock transaction, *pBlock* references a data block which contains a value which is returned by the lock transaction. The pre-update data value of the target block is returned. This is useful if a "compare_swap" operation has failed, hence an additional read is not required before the transaction is retried, *pBlock* must be equivalent in dimension to *pDataValue*.

The caller of this routine is blocked until the entire transaction is completed. Upon

completion, the calling process receives a remote interrupt, of code INTR_DRFAIL or INTR_DRSUCCESS. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
LCKID	: A compare_swap inconsistency was detected.
FAIL	: An error associated with the transaction layer.

```
DRSTATUS DR_SetCallBack (
        PADDRESS_OFFSET    pLowAddr,   /* Low addr for range */
        PADDRESS_OFFSET    pHighAddr,  /* High addr for range*/
        Objid              hProcess,   /* Target process ID  */
        USHORT             fulOps,     /* Ops trigger        */
        USHORT             fulPerms,   /* Permissions        */
        PSRBLIST           pSRBList    /* SRB list           */
    )
```

Purpose: Allows a target process to specify conditions for a callback and access rights to a block of memory.

Description: The client specifies an address space, along with and the operations on the address space (i.e., write or lock) which triggers a callback upon receiving an asynchronous transaction request. The permission flags are reserved for future use; they limit access to various regions based on an operation type.

fulOps Flags:

0x0001 : callback upon write;

0x0002 : callback upon read;

0x0004 : callback upon lock;

fulPerms Flags:

0x0001 : allow a write;

0x0002 : allow a read;

0x0004 : allow a lock;

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: Could not add an additional callback condition.

E.2 Resource Acquisition API

```
DRSTATUS DR_IsochAllocateBandwidth (
        ULONG nMaxBytesPerFrameRequested, /* Maximum payload  */
        ULONG          fulSpeed,           /* Transmission rate  */
        PBANDWIDTH     pBandwidth,         /* Acquired bandwidth*/
        ULONG          *pBWAvail,          /* Available bandwidth*/
        ULONG          *pfulSpeedSelected  /* Actual rate */
)
```

Purpose: Acquires isochronous bandwidth from the IRM.

Description: DR_IsochAllocateBandwidth acquires isochronous allocation units based on the bus speed and the number of bytes require per frame. After the successful completion of this transaction the application must not transmit a frame which exceeds the specified maximum.

If the application is the source of more than one channel, this routine can be called more than once. The sum of all the requisitions must be the maximum size of any concatenated packet.

The *nMaxBytesPerFrameRequested* parameters specifies the maximum number of bytes which the application intends to transmit per 125 μs cycle, at the speed which is specified by the flag *fulSpeed*. Our simulation supports the 100Mbps transmission speed.

The *pBWAvail* references a ULONG. Upon completion of the transaction, a count of the total remaining bandwidth at the IRM is copied to this ULONG. This is useful

information if the transaction has failed due to insufficient bandwidth.

The speed which was selected for subsequent transmissions is based on speed map information. This speed may differ from the original speed specified by the *fulSpeed* parameter. For our simulation, 100 Mbps is the selected speed and it is copied to the ULONG referenced by *pfulSpeedSelected.*

The parameter *pBandwidth* references a BANDWIDTH structure. If the acquisition of resources was successful, the number of allocation units is copied to the BANDWIDTH structure.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRFAIL or INTR_DRSUCESS. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
RESNA	: Failed due to insufficient available bandwidth .
FAIL	: An error associated with the transaction layer.

DRSTATUS DR_IsochAllocateChannel (

USHORT	**nRequestedChannel,**	/* Desired channel */
USHORT	***pChannel,**	/* Acquired channel */
ULONG	***pChannelsAvailable**	/* Available channels */

)

Purpose: Performs an isochronous channel allocation.

Description: DR_IsochAllocateChannel acquires a channel from the IRM. Either a specific channel or the next available channel ID can be requested. The caller provides a reference *pChannel* to a USHORT, and a reference *pChannelsAvailable* to a ULONG[2]. The requested channel is specified by the *nRequestedChannel* parameter. If any arbitrary channel is adequate, 64 , an invalid channel ID, is passed, otherwise, 0..63 are valid requests. If the allocation was successful, the acquired channel is referenced by the *pChannel* parameter. Upon completion of the transaction, a copy of the remaining

channels register at the IRM is copied to the data block referenced by the *pChannelsAvailable* parameter.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRFAIL or INTR_DRSUCESS. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
RESNA	: Failed due to the channel not being available.
FAIL	: An error associated with the transaction layer.

DRSTATUS DR_AllocateBuffers (

```
USHORT    nMaxBytesPerFrame,    /* Capacity            */
USHORT    nNumberOfBuffers,     /* Number of buffers   */
PBRNG     *pBuffer              /* The resulting buffer*/
)
```

Purpose: Allocate and initialize a PBRNG structure.

Description: DR_AllocateBuffers allocates and initializes a queue of buffers which are used for isochronous talk and listen operations. However, the memory for each buffer element is not allocated - such memory is allocated by the source process. The capacity of each buffer is specified by the *nMaxBytesPerFrame* parameter and is the maximum size required for an isochronous packet associated with a specific channel. The number of buffers is specified by the *nNumberOfBuffers* parameters.

A reference to a PBRNG structure is passed as the *pBuffer* parameter. Upon allocation of the buffers, the PBRNG structure refers to the newly allocated BRNG structure.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: Failed due to memory availability.

DRSTATUS DR_IsochFreeBandwidth (PBANDWIDTH pBandwidth)

Purpose: Release a bandwidth allocation.

Description: DR_IsochFreeBandwidth releases isochronous bandwidth which was previously allocated by the DR_IsochAllocateBandwidth transaction. The reference to the BANDWIDTH structure, initially returned by DR_IsochAllocateBandwidth, can be passed by the *pBandwidth* parameter. Upon calling this routine, the resources associated with *pBandwidth* must not be utilized.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: An error associated with the transaction layer.

DRSTATUS DR_IsochFreeChannel (USHORT nChannel)

Purpose: Release a channel allocation.

Description: DR_IsochFreeChannel releases an isochronous channel which was previously acquired from the IRM by DR_IsochAllocateChannel. The channel to be released is passed in the *nChannel* parameter.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: An error associated with the transaction layer.

DRSTATUS DR_DeallocateBuffers (PBRNG pBuffer)

Purpose: Dellocate any resident buffers and the BRNG structure.

Description: DR_IsochAllocateBuffers releases the resources associated with BRNG structure referenced by the *pBuffer* parameter.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: Failed due to a corrupt BRNG structure.

E.3 Isochronous API

```
DRSTATUS DR_IsochListen(USHORT     nChannel,   /* Source channel */
                        PBRNG      pBuffer,    /* Sink buffer */
                        Objid      hProcess    /* Sink process */
)
```

Purpose: Bind a sink buffer with a source channel.

Description: DR_IsochListen binds an isochronous channel with an input buffer for the purpose of receiving traffic over a specific channel. The desired channel is specified by the *nChannel* parameter. The allocated buffer is referenced by the *pBuffer* parameter. And the *hProcess* parameter (acquired by the OPNET kernel procedure *op_self_id()*) is the object ID of the sink process which should be signalled upon the reception of each packet over the specified channel.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

After successful completion of this routine, isochronous packet arrivals are added to the queue referenced by *pBuffer*, and a remote interrupt of code INTR_BFIP is sent to the process ID *hProcess*.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: Failed due to the channel not being available.

```
DRSTATUS DR_IsochTalk (USHORT    nChannel,    /* Sink channel */
                       PBRNG     pBuffer      /* Source buffer     */
    )
```

Purpose: Bind a source buffer with a sink channel.

Description: DR_IsochTalk binds an isochronous channel with an source buffer, for the purpose of sending traffic over a specific channel. The desired channel is specified by the *nChannel* parameter. The allocated buffer is referenced by the *pBuffer* parameter.

The caller of this routine is blocked until the entire transaction is completed. Upon completion, the calling process receives a remote interrupt, of code INTR_DRSUCCESS or INTR_DRFAIL. The return code *DRSTATUS also indicates the success of the transaction, or the specific reason for failure.

After successful completion of this routine, when the source process adds a new data block to the buffer referenced by *pBuffer*, the driver initiates a isochronous link-layer transaction over the channel specified by the *nChannel*. The driver yields one transaction per channel, per cycle, as long as data blocks are populating the buffer.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: An error associated with pBuffer.

```
DRSTATUS DR_IsochStop (USHORT    nChannel,    /* Channel ID          */
                       ULONG     fulFlags,    /* Stop talk or listen  */
                       Objid     hProcess     /* Requesting process */
    )
```

Purpose: Stop isochronous listen or talk operations.

Description: DR_IsochStop unbinds a channel and buffer (source or sink) association, therefore, terminating the reception or transmission of associated packets. The *nChannel* parameter specifies the channel ID. The *fulFlags* parameter dictates the type of operation:

0x0001	: stop the listen operation
0x0002	: stop the talk operation

And the *hProcess* parameter (acquired by the OPNET kernel procedure *op_self_id()*) is the object ID of the source process which initially called the DR_IsochTalk routine.

In the case of DR_IsochStop being called to terminate packet transmission, the association between the channel number and the buffer is broken. Hence, any subsequent insertions to the buffer are ignored by the driver.

Returns:

SUCCESS	: A successful completion of the routine.
FAIL	: Failed due to invalid parameters.

E.4 Miscellaneous API

USHORT DR_GetNodeID (void)

Purpose: Returns the ID of the node.

Description: The node ID is not dynamically determined within the simulation, as there are no tree identification or self identification phases modelled. The "NODE" attribute is manually set within the network domain of the OPNET simulation. This routine returns this attribute value.

Returns:

63	: Indicates an error.
0..62	: Indicates the node ID.

USHORT DR_GetBusID (void)

Purpose: Returns the ID of the local bus.

Description: For purposes of the current simulation this call will always return '1'. Currently, the OPNET simulation does not support multiple buses.

Returns:

-1	: Indicates an error
0..1023	: Indicates the bus ID

USHORT DR_GetIRMID (void)

Purpose: Returns the node ID of the IRM.

Description: As the IRM election or the self identification phase is not addressed by our simulation, we can manually set a node ID for the IRM.

Returns:

63 : Indicates an error

0..62 : Indicates the node ID

PIO_ADDRESS DR_GetIOAddress (

USHORT	**nBus,**	/* Bus ID	*/
USHORT	**nNode,**	/* Node ID	*/
ULONG	**nOffset**	/* Destination offset	*/

)

Purpose: Returns an IO_ADDRESS structure.

Description: Returns an IO_ADDRESS structure which specifies the nBus, nNode, and nOffset parameters. We only implement a 32bit offset, we assume that the high order 16 bits are 0x0000. However, the 48 bit offset is still defined within the relevant request packet formats.

Returns:

OPC_NIL : Indicates an allocation error.

void* MEM_GetRegAddr (

ADDRESS_OFFSET AddressOffset /* 48bit address offset */

)

Purpose: Translates a virtual offset into a pointer.

Description: Offset addresses, specified by 1394 asynchronous transaction requests, must be translated into pointers for our simulation environment. Therefore, the contents of a memory block, which is referenced by a virtual memory address, can be can accessed and modified.

Returns:

OPC_NIL : Indicates a translation error.

E.5 Isochronous Buffer API

USHORT BUFF_IsEmpty (PBRNG pBrng /* Pointer to queue of buffers */

)

Purpose: Determine if buffers occupy the queue.

Description: Call BUFF_IsEmpty before calling routines which require a non-empty queue.

Returns:

TRUE : Buffer(s) have not been utilized.

FALSE : Buffers have been utilized.

USHORT BUFF_IsFull(PBRNG pBrng /* Pointer to queue of buffers */

)

Purpose: Determine if there is capacity for another buffer.

Description: Call BUFF_IsFull before calling routines which require a non-full queue.

Returns:

TRUE : All buffers have been utilized.

FALSE : Not all buffers have been utilized.

```
BUFF_AccessHead(PBRNG pBrng,          /* Pointer to queue of buffers */
                void **pBuffer,       /* Pointer to acquired buffer  */
                USHORT *nActSize,     /* Actual size of buffer       */
                USHORT *nMdlSize      /* Modelled size of buffer     */
)
```

Purpose: Retreive and remove the buffer at the head of the queue.

Description: A pointer to the head buffer is returned in the *pBuffer* parameter. The actual size and modelled size of the buffer are returned in the *nActSize* and *nMdlSize* parameters,

respectively.

Returns:

SUCCESS	: The routine was successful.
FAIL	: The queue was empty.

```
USHORT    BUFF_WriteTail (PBRNG pBrng,  /* Pointer to queue of buffers */
          void *pPayload,               /* Pointer to the inserted buffer */
          USHORT nActSize,              /* Actual size of buffer        */
          USHORT nMdlSize               /* Modelled size of buffer      */
  )
```

Purpose: Add a buffer to the tail of the queue.

Description: A data block referenced by *pPayload* is added to the queue *pBrng*. The actual size and modelled size of the buffer are returned in the *nActSize* and *nMdlSize* parameters, respectively.

Returns:

SUCCESS	: The routine was successful.
FAIL	: The queue was full or the buffer was too large.

Vita

Surname: Foxgord Given Names: John Ross

Place of Birth: Victoria, British Columbia.

Education Institutions Attended:

University of Victoria	1996 —2001
University of Victoria	1989 —1994
Camosun College	1987 —1989

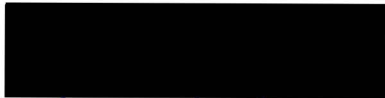
Degrees:

B.Sc. (Co-op)	University of Victoria	1994

Partial Copyright Licence

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Application Level Resource Management for the IEEE 1394 Serial Bus.

Author 

John Ross Foxgord

August 29, 2001