Application of Media Channels to Large-Scale Group Communication

Steven Simpson

Computing Department
Lancaster University
Lancaster LA1 4YR, UK

Submitted for the degree of Doctor of Philosophy

September, 1998

Abstract

While computer networks have been of low capacity or poor reliability, their use for real-time communications for human interaction was considered infeasible on any large scale, since audio and video data consumes vast amounts of memory and processing capacity. However, recent and forthcoming enhancements to network quality, along with advanced compression algorithms, falling costs of memory and fast processors, have allowed such applications to develop, as well as protocols to support them.

On top of this, global interconnectivity of local networks has improved, giving scope to expand human communication application to wide areas, and with many participants. Such group communication applications have various networking Quality of Service (QoS) demands, such as lower bandwidth but high reliability. The major challenge in multimedia group communication is therefore about developing network services to support group communication with varying QoS, in order to allow groups of people, potentially very widely distributed around the world, to interact through computers of varying abilities, using combinations of media such as video, audio, and shared editing of documents.

The Media Channel (MC) model is an attempt to unify the control and management of applications of different media types. It provides common group management facilities for all types of application, and allows participation in an application to be expressed simply as contribution (using a source) and/or observation (using a sink). This further allows a multimedia group application to be built up from several ‘unimedia’ group applications (the ‘channels’) through the use of a user agent which manages and simplifies a user's participation in several applications. Participation in multimedia group applications (or ‘sessions’) can then be configured automatically for several users by describing the channels of the session to each involved user agent. This description itself can be supplied through a channel, with the agents acting as sinks. The user can control participation in all the channels of the session by (de)activating the agent's session sink, and while joined, session administrators can inform all

session participants of new channels to which they are invited.

The MC model's main shortfall is its use of a central manager — an architecture which does not scale well to many participants, and which may lead to uneven management for widely distributed participants, particularly as group populations migrate. This thesis considers the issues of multimedia group communication, and attempts to realise the MC model. Some earlier definitions of the MC model are refined in the present thesis, and alternative architectures are designed, implemented and evaluated. These new architectures allow the management components of a media channel to be placed near to the participants, and then to migrate as participants leave the components' areas, and as new ones join in others. At the same time, management information to evaluate policies is compressed and distributed among components. The model is also extended to allow sources to describe their data arbitrarily, enabling sinks to accept or reject them individually depending on availability of local resources.

Acknowledgements

Firstly, I must thank Dr. Geoff Coulson for initially contacting me with the offer of undertaking this research. I am grateful to the EPSRC for their support through a CASE studentship (ref. 9456391X), and additional sponsorship by BT Laboratories is greatly appreciated.

Thanks also go to my academic colleagues, particularly Dr. Francisco García and Dr. Nick Yeadon for allowing me to participate in some of their areas of research, and Dr. Frank Ball and Dr. Ki-Cheon Kim for their critical viewpoints during the early stages of my work. I am greatly indebted to Dr. Silvester Namuye for introducing me to group communications, and giving direction to my thesis, as well as being a close friend. Dr. Andreas Mauthe and Mr. Laurent Mathy also deserve thanks for support in the later stages of my work by pointing out its value, and by suggestions to fill omissions. I am grateful to Ms. Randa El-Marakby, with whom I shared an office, for her hard working and optimistic example.

Finally, I would like to give my sincerest thanks to my supervisor, Prof. David Hutchison, for his support and advocacy of my research efforts, and for his friendship.

Contents

1	**Introduction**	**1**
1.1	Multimedia Group Communication	1
1.2	The Media Channel Model	2
1.3	Structure	4
2	**Multimedia**	**6**
2.1	Definition	6
2.2	Multimedia Network Requirements	7
2.2.1	Quality-of-Service	7
2.2.2	Filtering	8
2.2.2.1	Filtering Process	9
2.2.2.2	Placement of Filters	10
2.2.3	Multimedia Services and Protocols	13
2.2.3.1	HTTP	14
2.2.3.2	RTP	15
2.2.3.3	XTP	16
2.2.3.4	RSVP	16
2.2.4.5	IPv6	17
2.3	Multimedia Encoding	17
2.3.1	Video Encoding	18
2.3.1.1	JPEG	19
2.3.1.2	MPEG	23
2.3.1.3	H.261	25
2.3.1.4	Other Video Formats	26
2.3.2	Audio Encoding	26
2.3.3	Compression of Other Data	27
3	**Group Communication Review**	**29**
3.1	Group Support	29

3.1.1 Group Communication Taxonomy 29
3.1.1.1 Population 29
3.1.1.2 Group Hierarchy 29
3.1.1.3 Communication 30
3.1.2 Group Requirements 31
3.1.3 Related Work 32
3.1.4 Group Communication Support 35
3.1.4.1 Multicasting Services 36
3.1.4.2 Multicast Modes 40
3.1.4.3 Group Management Services 42
3.1.5 Group Naming 44
3.2 Media Channel Model 46
3.2.1 Taxonomy and Services 46
3.2.1.1 Overview 46
3.2.1.2 MC Model using Multipeer Taxonomy 52
3.2.1.2 Services 54
3.2.1.3 Application of MC Service 61
3.2.3 Revisited Centralised Architecture 62
3.2.4 Examples of MC Use 66
3.2.4.1 Example MC Types 66
3.2.4.2 Example MC Sessions 67
3.2.5 Achievements and Limitations 69

4 Distributed Media-Channel Architectures 71

4.1 Communication Scaling 72
4.1.1 Scaling Multicast Modes 72
4.1.1.1 Root-initiated Multicast 72
4.1.1.2 Leaf-initiated Multicast using Multicast Address 73
4.1.1.3 Leaf-initiated Multicast using Root Address 73
4.1.2 Domain Interconnection 73
4.1.2.1 Repeaters 74
4.1.2.2 Positioning of Repeaters 75

4.1.2.3 Management Implications 76
4.1.2.4 Evaluation 76
4.2 Management Scaling and Distribution 77
4.2.1 Management Interaction with Communication 77
4.2.1.1 Multipeer Co-ordination 78
4.2.1.2 Multipeer Emulation 79
4.2.1.3 Multipeer Incorporation 81
4.2.2 Management Architectures 81
4.2.2.1 Domain Hierarchy 81
4.2.2.2 Flat Tree of Domains 82
4.2.2.3 Per-Client Management 83
4.2.3 Addressing for Multiple Managers 83

5 Design & Implementation 85

5.1 Common Components 86
5.1.1 Distributing Connection Information 86
5.1.2 Distributing Integrity-Evaluation Information 88
5.1.3 Distributed Manager-Client Protocol 90
5.1.3.1 Additional Types 90
5.1.3.2 Manager-Source Primitives 90
5.1.3.3 Manager-Sink Primitives 92
5.2 Tree Maintenance 94
5.2.1 Hierarchical Tree Maintenance 94
5.2.2 Flat Tree Maintenance 96
5.3 Implementation 97
5.3.1 Co-operating Manager 97
5.3.2 Audio Utilities 98
5.3.3 Performance Utilities 99
5.3.4 Tree Management Utilities 99

6 Evaluation of the Distributed Media Channel Model 101

6.1 MC Service Evaluation 101

6.2	Session Services Evaluation	102
6.3	MC Architectures Evaluation	103
6.4	MC Implementation Evaluation	105
6.5	Distributed versus Centralised Performance	106
7	**Conclusions**	**110**
7.1	Thesis Summary	110
7.2	Contributions of the Thesis	111
7.3	Further Work	114
Appendix A: Distributed Processing Environment		**116**
A.1	Transport Module	116
A.2	Object Module	118
A.3	Named-Service Module	120
A.3.1	Model	120
A.3.2	Service	121
A.3.3	API	121
7.3	Further Work	114
Appendix B: Event Management		**129**
B.1	Alarm Module	129
B.1.1	Model	129
B.1.2	Service	129
B.1.3	API	129
B.1.3.1	Class Alarm	130
B.1.3.2	Class Alarm::DB	130
B.2	File Event Module	131
B.2.1	Model	131
B.2.2	Service	131
B.2.3	API	132
B.2.3.1	Class FileEventSet	132

B.3 Utilisation 132

Appendix C: Interface Definitions 134

C.1 Manager-Client Interfaces 134

C.2 Factory-Client Interfaces 135

Appendix D: Timing Results 137

References 139

List of Figures and Tables

Figure 2.1: Filter placed at sender 11
Figure 2.2: Filters placed at receivers 11
Figure 2.3: Filters placed among multicasting network nodes 12
Figure 2.4: Layered multicast trees 13
Figure 2.5: Lossy compression using DCT and quantization 20
Figure 2.6: Zig-zag ordering of co-efficients 20
Figure 2.7: Pixels used for prediction 21
Figure 2.8: Lossless encoding using prediction 21
Figure 2.9: Hierarchical encoding 22
Figure 2.10: Component data interleaving 23
Figure 2.11: Motion compensation 24
Figure 2.12: MPEG frame dependencies 24
Figure 2.13: MPEG B-frame ordering 25
Figure 2.14: H.261 macroblock 26
Figure 2.15: μ-law conversion function 27
Figure 3.1: MBone topology 37
Figure 3.2: Root-initiated Multicast 40
Figure 3.3: Leaf-initiated Multicast using Multicast Address 41
Figure 3.4: Leaf-initiated Multicast using Root Address 41
Figure 3.5: Media Channel Hierarchy 47
Figure 3.6: Stream structure 48
Figure 3.7: Source and sink notation 49
Figure 3.8: MC structure 50
Figure 3.9: User agent display 51
Figure 3.10: Media Channel model services 54
Figure 3.11: Joining, leaving and address association primitives 57
Figure 3.12: Addressing interactions 59
Figure 3.13: Leaf-initiated manager interaction 64
Figure 3.14: Root-initiated manager interaction 65
Figure 4.1: A repeater connecting two domains 74
Figure 4.2: A *N*:*N* non-mixing repeater 75
Figure 4.3: A *N*:1 mixing repeater 75
Figure 4.4: Multicast addresses per domain 80
Figure 4.5: Leaf domains co-ordinated by a central manager 82
Figure 5.1: Root-initiated co-ordination problems 87

Figure 5.2: Leaf-initiated emulation supporting co-ordination 88
Figure 5.3: Sharing counter values 89
Figure 5.4: Hard integrity inconsistencies 89
Figure 5.5: Factory/domain hierarchy 94
Figure 5.6: Adding new clients 95
Figure 6.1: Processes for first run (A1) 106
Figure 6.2: Processes for second run (A2) 107
Figure 6.3: Processes for third run (A3) 107
Figure 6.4: getaddr_ind times 108
Figure 6.5: First data packet times 109
Table D.1: getaddr_ind times (s) 137
Table D.2: First packet times (s) 138

1 Introduction

This chapter sets out the motivation and aims of the thesis, and gives a summary of its structure.

1.1 Multimedia Group Communication

Humans already make use of telephone networks to have conversations with each other, and the potential has been seen in computer networks (as their bandwidth increases) to use them for human communication, carrying audio and video as well as text and files, giving rise to the term 'multimedia'. Computer networks such as the Internet are also better suited to delivery of data to multiple destinations, and so there is a second potential to apply that to communication between groups of humans communicating with each other. The field of *multimedia group communication* is about assisting a group of humans communicating with each other through their computer terminals, distributed over a network.

While networks now support efficient multicasting, where transmitted data arrives at multiple destinations without using unnecessary bandwidth (such as the MBone [Eriksson 94] — the backbone for IP (Internet Protocol) multicasting [Deering 89]), and computers now support capture, storage, processing and display of many more media types (*e.g.* video and audio), integration of these two facilities is problematic. Each media type may require a different *Quality of Service* (*QoS*), that is bandwidth, latency, jitter, reliability *etc.*, from the multicasting transport service, and so several separate multicast connections may need to be set up. However, there will be associations that must be maintained between media types (such as synchronisation of audio and video) or between the multiple receivers (altogether referred to as orchestration in [Campbell 92]). These questions are under research, and some solutions have been put forward [Pasquale 92] [Gupta 93b] [Heijenk 94].

Another area of research is in the requirements that users have of conferences (that multimedia group communication systems will support), such as security, participation control, connection integrity (since it is more complicated to determine if a multipeer connection is successful than it is for a unicast connection), and can be described as *group management*.

Heterogeneity amongst computer terminal and network equipment exists to further complicate matters. Some terminals will not support video, or to the same resolution as other terminals receiving the same data. Networks to some terminals may be unable to carry as much bandwidth as others. Users may impose further restrictions because they are not interested in the full set of data available to them, or because they wish to reduce cost. In summary, the users' terminal programs that render various media types should be able to select different resolutions of the conference (*e.g.* video resolution/audio sampling frequency).

1.2 The Media Channel Model

The Media Channel (MC) model for supporting multimedia group communications attempts to digest the problem of expressing and managing a multimedia group interaction by dividing it up into multiple *media channels*, each of which may have a different *media type* (technically a standard to describe how an item of real-world data changes and how to express those changes through communication, *e.g.* a video frame, encoded for transmission) which is named (rather like MIME types [Borenstein 93]). Each channel has a data component whose type is described by the media type, and is known as the channel's *data state*. A management component of each channel (the *channel manager*) keeps a register of end-systems wishing to participate in the channel as receiving a copy of the data state (the *sinks*), and a register of those wishing to define a part of the state (the *sources*). Early steps towards the MC model resulted in an architecture but not a realisation [Namuye 96a], [Namuye 96b].

As an example, a video conference could consist of a single channel of a suitable video media type. Each participant would register as both a sink and a source of the channel, each using the input from his video camera as his part of the channel's data state (the complete state at any point in time is the aggregate of the images currently captured by each of the cameras). Each of the sinks would then receive an image from each of the cameras, through the intervening network, to form a copy of the complete state at each of the sinks, as they were registered for.

More abstract media types can be used to describe the associations between channels [Gupta 93b], an obvious one being that several individual channels form a complete multimedia conference. This would be implemented through a channel of type "session", whose state is the list of channels in the conference, and some information on an appropriate way to connect to each channel (*i.e.* as a source/sink/both). Participants would act as sinks of this channel, and so would receive the information relevant to participating in the other channels. A conference controller would act as a source for the channel, and could alter the information, automatically informing the other participants of the change. Other abstract types could include a security media type, a channel of which informs the managers of the other channels in the session of acceptable participants. A synchronisation type would inform end systems of inter-channel synchronisation requirements (or this would be incorporated into the “session” type).

The Media Channel model, as originally outlined in [Namuye 96b], has a number of shortfalls:

- the manager is centralised, and could be overloaded by too many participants,
- the manager is stationary, and its position could be inappropriate after some time as its population of participants migrates,
- sinks cannot indicate their refusal of individual sources, which the manager may need to know to evaluate policies,
- inter-channel relationships are not defined.

This thesis attempts to build a realisation of the model to assess its feasibility, simultaneously applying the model to large-scale distribution. The overall approach is to distribute the manager over a wide area (to reduce a potentially large load on a single manager), transforming the problem to one of re-applying the manager's existing and new functions to that distribution.

The upshot of this approach is that, in this MC model, a single media channel is represented by multiple media channel managers, and participants register with their nearest manager, selected from the set of managers supporting that channel. (Previously, the channel was identified by its manager's address, which was intrinsically globally unique. In the realisation of the model, a globally unique name must be assigned to the channel, and a facility must exist to map this to the nearest manager, from within a particular domain.) This is similar to research in the World-Wide Web (WWW) where browsers that attempt to download pages (identified by *Uniform Resource Locators* (URLs) [Berners-Lee 94]), and fail, can be provided with a set of alternative pages containing the same information as backup. The pages in the set have been registered under the same *Uniform Resource Name* (URN) in a service that resolves single URNs into multiple URLs [Sollins 94]. If manager addresses can be expressed as URLs, then this service can go some way to specifying the nearest manager by registering their addresses with a single URN identifying the channel, so this area of research is also investigated.

1.3 Structure

The thesis has the following structure.

Chapter 2 describes multimedia and its impact on communication. Encoding schemes for high-bandwidth media are discussed.

Chapter 3 is a review of group communication, and its development towards multimedia applications in terms of service requirements and current research. The media channel model is also revised, and its limitations highlighted.

Chapter 4 shows possible strategies for realising the MC model and its architecture to suit wide-area application. One strategy is selected for implementation.

Chapter 5 designs the MC model to include the wide-area extensions, and illustrates the implementation of the design, including APIs.

Chapter 6 evaluates and measures the achievements of the design and implementation.

Chapter 7 concludes the thesis, comparing the achievements of the design with the requirements set out in chapter 3, and indicates further work to be done.

2 Multimedia

This chapter describes the development of multimedia in computing. Section 1 defines the term, section 2 describes the network requirements for transmission of multimedia, and section 3 describes some encoding schemes for reducing network load.

2.1 Definition

Originally, computers were used to process numbers, mainly in situations where the quantity of data was too large to be done reliably or quickly by hand. Indeed, this is still the case, but the numbers may now be interpreted in many ways other than just as numbers themselves.

The earliest alternative abstract uses for numbers were text and machine code. Text would appear in word-processing documents, source code and configuration files, while machine code would form the programs. Copies of these data had to be completely reliable to be of any use. Other abstractions, such as video frames or audio samples, were difficult to consider as multiple items, as they occupied too much space, or demanded too much processing time.

When digital networks were being developed, the only data to be transferred across them were the low-bandwidth, fault-sensitive media types of text and programs, so the network services were geared towards reliability, usually through retransmission. Now, as networks and computers increase in overall power, there is a potential to use them for processing high-bandwidth data such as audio and motion video which require timely presentation. Computers have now become devices for processing many types of data such as text, pictures and sound, and *multimedia* is used to describe this type of processing. Multimedia computers have the processing power, memory and hardware capable of capturing and rendering the more demanding types, and therefore can handle all types. Multimedia networks can provide reliable services for the earlier types, and timely services for the modern types. Multimedia applications running on multimedia computers make use of the available hardware to process a wide range of media types.

The term ‘media type’ can be used to describe real-world phenomena, like moving images and sound, as well as their electronically captured versions, and the various methods in which these electronic signals are digitised and encoded. Continuous media (CM) are those with a continuously changing state, such that an electronic representation and its transmission involves continuous data flow, where newly arrived data supersedes earlier data, or modifies it slightly. Motion video and audio are types of continuous media. On the other hand, discrete media data at the receiver remains valid for long period of time before alterations arrive. Still video and text documents are example discrete media types. Some media, or uses of media, have real-time requirements: a change in the captured media state should be reflected by each receiver within certain time constraints. Real-time applications can apply to both continuous and discrete media types, and this thesis is mainly concerned with such applications.

2.2 Multimedia Network Requirements

This section describes what is required of a network to carry multimedia, and describes some protocols and media-encoding schemes that have been developed to support multimedia communication.

2.2.1 Quality-of-Service

The modern Internet has two main transport services, TCP and UDP. The latter is little more than an application multiplexer on top of an unreliable network-level datagram service (IP). TCP provides virtual streams (*i.e.* sequenced delivery of data), and reliability through retransmission of lost packets.

TCP is suitable for transport of the old media types of text and programs, where the data is only useful if copied correctly with eventual delivery. For continuous, real-time media, retransmission is inappropriate, as it may be too late to render the lost packets by the time they are received.

Different classes of media need different behaviour from transport across a network in order to be useful upon reception. Differing degrees of reliability are required orthogonally against varying timeliness. Network services can be described in terms of their *Quality of Service* (QoS), which encompasses these parameters, and the degree to which they are guaranteed. A service may not only try to deliver packets on time or reliably, but may also guarantee it. The guarantee may be such that the service will not fail, or it may simply mean that the user is informed of failure.

One aim of QoS is to ensure that network resources are used fairly when higher-bandwidth media are transported, which is particularly important for costing. Another aim is to assist in network management, for example, to reroute connections around busy links — the QoS of the connections can be used to determine which initial or alternative links are suitable to carry those connections. Many publications have resulted from work on QoS in recent years, including [Campbell 92], [García 95].

2.2.2 Filtering

The heterogeneity of network capabilities and terminal equipment can cause some problems with delivery of multimedia. The QoS of the network may be unable to meet the QoS required by the media type in its complete form, or the terminal may be unable to render the data in its complete form. The terminal user may impose additional restrictions upon the network in order to reduce cost.

From the point of view of filtering, these are seen as the same problem. The aim of filtering is to reduce the data in its quality without destroying its usefulness, in order to reduce its impact on the network, or simply to remove the detail that a terminal is unable to render anyway. In other words, some information is not transmitted, either because it is not needed (because the receiver doesn't want it, or can't process it), or because the network cannot carry it all — it is filtered out of the stream.

Filtering is usually only applicable to media that do not require full reliability, such as audio and video, as filtering constitutes loss in all media. Unlike network loss, where whole chunks of information are made unrenderable, the process of filtering is one of deselecting the least necessary parts of the medium before transmission, so loss is distributed evenly across the complete information.

For example, to reduce the overall bandwidth of a video stream, one may extract the colour components of the data, so that it is displayed as a monochrome image. This is also useful if the display device is monochrome, which would not be able to render the colour data anyway. Alternative attributes for reduction are the frame rate (the video is still rendered in real time), the resolution, and the area of the image to be displayed. For audio, the sample size may be reduced, while maintaining the sample rate, which results in a reduction of bandwidth.

Another purpose of filtering is transcoding: converting from one encoding to another, because the receiver is unable to process the original encoding. A transcoding filter is developed in [Yeadon 96a/b].

Filtering can be directly specified in terms of the media type's dimensions, *e.g.* video resolution, colour/monochrome. However, if network-traffic reduction is the aim, it is sometimes more appropriate to specify filtering in terms of network traffic QoS parameters, usually bit rate. It is then necessary to define a mapping from bit rate to the media type's parameters. RSVP, for example, allows network resource reservations for streams from particular (sets of) sources to be expressed to network components, but those components are expected to decide how data flowing through them should be reduced in QoS capacity in order to stay within the reserved limits [Zhang 93].

2.2.2.1 Filtering Process

In general, the approach to filtering a rich, encoded incoming stream of a CM type into a poorer one is as follows:

1. receive a chunk of the stream,
2. decode the chunk, virtually rendering it,
3. remove the unwanted data, *e.g.* colour,
4. re-encode the remaining data,
5. pass the chunk on.

A media stream may have several dimensions which can be filtered independently of one another. For some of the more complex encoding types, this process of decoding and re-encoding will be time-consuming. To reduce the overhead of these steps, the encoding scheme may be *hierarchically* defined such that the stream is fragmented at the source according to its levels of detail. For example, some packets may contain the monochrome parts of a video image, while other packets may contain the adjustments to the monochrome signals that colour them. The two types of packets have been made easily distinguishable by their headers, so a filter instructed to remove colour from a stream simply drops the packets with headers marked ‘colour’. This type of filtering is known as *discrete filtering* in [Toutain 95]. *Continuous filtering* is required when the various levels of detail are not easily selected from the encoding scheme, and the stream must be largely decoded, stripped/converted, and recoded.

2.2.2.2 Placement of Filters

For a 1:1 transmission of a filterable stream, a filter is best placed at the sender, minimising the amount of network that carries the unfiltered data, and therefore the overall network load. This will be the typical arrangement for a video-on-demand scenario, with data flowing from the server to the client. Two such streams opposing each other could form the multimedia variant of a telephone call. In each case, the receiver would configure the filter at the other end of its stream according to the receiver's rendering capabilities for the particular media type, and the configuration may change over time as the QoS provided by the network changes.

For multicast transmission, *i.e.* 1:*N*, the principle of keeping filters nearer to the sender still applies, but by placing one at the sender, all receivers are forced to receive the same maximum quality, as in figure 2.1.

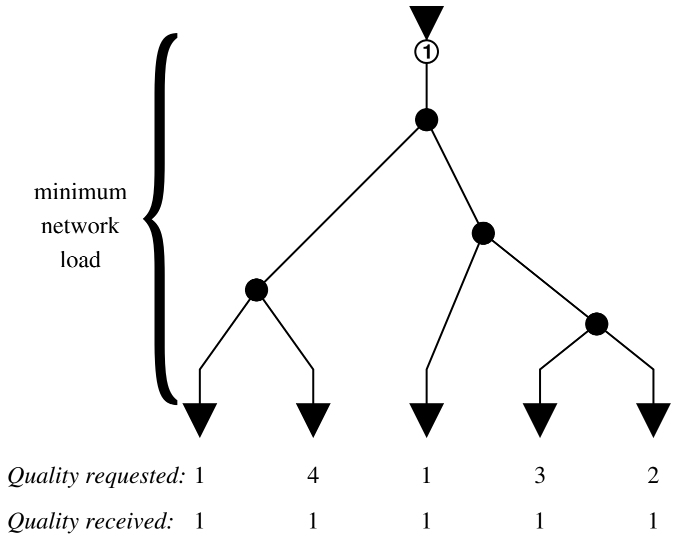


Figure 2.1: Filter placed at sender

Placing individual filters at the receivers on a per-receiver basis achieves the minimum required quality as far as each receiver is concerned, but little for the network, as the full load of the data has already passed over it, as shown in figure 2.2.

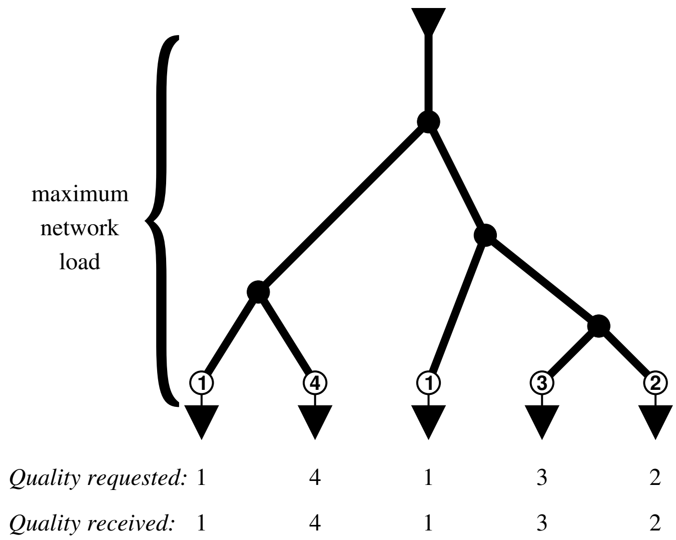


Figure 2.2: Filters placed at receivers

However, if the network supports it, filters may be placed at intermediate points in the network, as close to the sender as possible without limiting the quality to other users, the least selective filters being nearer the sender (figure 2.3).

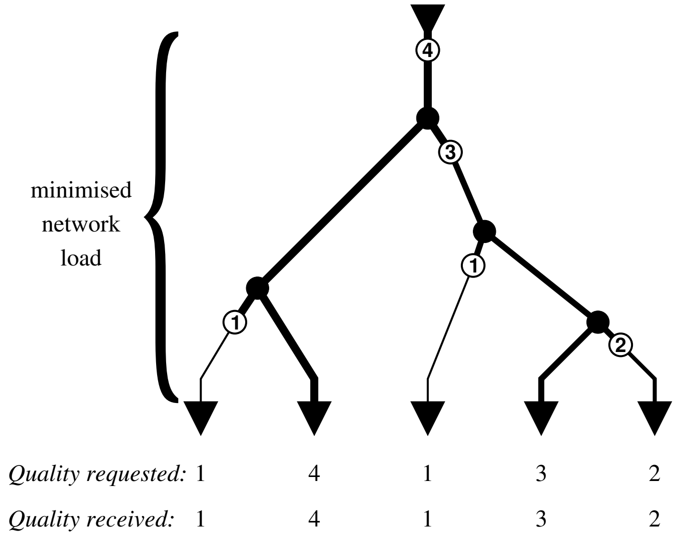


Figure 2.3: Filters placed among multicasting network nodes

The problem of placing filters in what may essentially be a 'black-box' network can be addressed in one case by making use of media types that support discrete filtering, and giving the network the minimum functionality to identify the headers in the media, so it can drop the unwanted packets. For continuous filtering, it may also be possible to define a virtual network of filtering routers on top of the existing network, in a similar way to which MBONE has been implemented with mrouted daemons [Eriksson 94]. Such routers are well placed for filtering support.

An alternative strategy to providing in-network filtering is to divide the stream into multiple streams of varying qualities, travelling over separate, but congruent, dissemination trees, known as layered multicasting. A receiver wanting the full quality binds to all the trees, whereas to receive a lower quality, some of the higher quality trees can be ignored, on a per-receiver basis. This has the advantage of being completely application-independent, but at the expense of some extra network resources such as multicast addresses or virtual channel identifiers. In particular, if a media stream has several independent quality parameters (*e.g.* resolution and colour could be separate), the number of dissemination trees required is the product of the number of values in each of these parameters (*e.g.* 4 resolutions and 2 colour levels requires 8 trees and their associated addressing resources). Figure 2.4 shows the tree of figure 2.3 implemented as four distinct trees carrying four layers of quality.

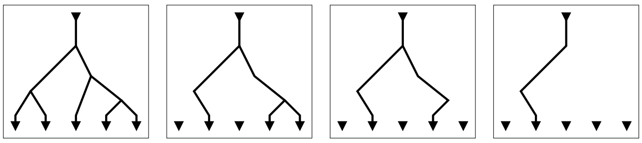


Figure 2.4: Layered multicast trees

Research is under way to define network protocols and services to support application-generic filtering within networks. An algorithm for network node behaviour, and its proof of correctness is explained in [Toutain 95]. For continuous filtering, [Pasquale 93] considers defining a common set of filters, including transcoding, identified by globally-agreed names. A parameterised continuous filter for MPEG video, that might be placed within a network, is implemented at Lancaster, and is described in [Yeadon 96b]. RSVP uses reservation messages to carry arguments for filters that specify QoS for particular sources (RSVP is intended for dissemination trees with multiple senders) [Zhang 93].

In all these schemes, filter specifications are generated by receivers, and then directed towards the sender(s). When received at a multicasting network node on one of its outgoing paths, the node implements a filter for data leaving via that path, according to the specification received. A node will also merge all its received specifications into the strictest one that is at least as liberal as the others, and this is propagated towards the sender(s), to be caught by the next node.

2.2.3 Multimedia Services and Protocols

A number of multimedia protocols now exist, and some are widely in use, while others are under research. They are described briefly below.

2.2.3.1 HTTP

The *HyperText Transfer Protocol* (HTTP) is the main protocol in use in the World-Wide Web (WWW), which is essentially a global, open, multimedia database. HTTP is implemented as a document-delivery protocol over TCP, and so offers reliable delivery with no timeliness. This is appropriate for most of the data that it carries, namely (hyper)text documents and still images which can be rendered together in static pages. Continuous media is downloaded as a complete, static document, and then rendered locally from a cache.

HTTP/0.9 was merely a document-downloading system, and could not indicate different types of data. HTTP/1.0 allowed requests and responses to be arbitrarily parameterised, include an indication of media type through its Content-type header field [Berners-Lee 96]. HTTP/1.0 supports multimedia by describing each document with a *Multimedia Internet Mail Extensions* (MIME) type [Borenstein 93], which unifies vendor-specific document-typing schemes at the client and server using globally-agreed names consisting of a general media-type part (*e.g.* `text`, `audio`, `video`) and an subtype, encoding part (*e.g.* `mpeg`, `ascii`, `basic`). For example, in a Unix system where a file's type is often identified by its name's extension, the server holds a mapping from extension to MIME type, which it transmits with the document, so the client does not have to be aware of the meaning of an extension at the server. When the client receives a document, it uses the MIME type to determine an appropriate method of rendition for the document. Usually this means displaying formatted hypertext (whose type is `text/html`) with in-line images (`image/gif`, `image/jpeg`), and this functionality is built into the client. For documents of other MIME types, the client has been configured to pass these on to *helper programs* which render the document separately. For example, MPEG files (`video/mpeg`) can be rendered by an external MPEG video player.

MIME types can be created at will (it is a matter of configuring the server to generate them), and their meaningfulness is defined by the availability of clients or helper programs which can understand them. Conventionally, MIME types used for development purposes prefix the subtype part with “`x-`”

HTTP can play some part in real-time delivery of continuous media by passing metafiles (identified by a MIME type) which the client has been instructed to pass to a continuous-media player. The metafile contains the address (and other information) of the CM file, and uses its own real-time protocol to transfer the data [Kim 95].

HTTP/1.1 introduces improved performance by making the use of a single connection for several requests/responses the default mode of operation [Fielding 97].

2.2.3.2 RTP

The Real-time Transport Protocol (RTP) is intended to support real-time data transmission across the Internet. It can work across unicast or multicast transport services, providing timing and sequencing information for packets, allowing the quality of transmission to be monitored. However, no QoS guarantees are made, nor is any resource reservation made [Schulzrinne 96].

An RTP session involves communication between multiple participants (in the multicast case; unicast is a trivial simplification), some of which are receivers only, while others are both senders and receivers. Senders transmit RTP packets via the underlying transport service to all the receivers. Participants periodically transmit control packets using RTCP (RTP's control protocol) to other participants over the same service. RTCP provides information to the group members on who else is participating in the session, and allows the senders to be aware of packet loss and delay in the network, and to adapt their streams to fit.

RTP is expected to be implemented by the application using it, rather than as a layer supported by kernel networking components, allowing tailoring of the protocol to specific applications. RTP packets are typically framed within the payload of UDP packets, and consist of a header and a payload. The payload is application-defined, and constitutes part of the RTP tailoring. The header can contain a sequence number and a timestamp supplied by the sender, allowing a receiver to reconstruct the ordering of a packetised stream. A payload type (application-defined) specifies the encoding used for the payload data, and allows senders and receivers to produce and consume different types of encoding for the same broad class of media (*e.g.* various audio encodings). The source identification of the payload data is also contained in the RTP header.

Within the network, RTP-level relays can act as mixers and translators. The latter act as bridges between different network and transport layer services, *e.g.* ST-II [Topolcic 90] and MBone [Eriksson 94]. Mixers can combine and convert multiple incoming streams into a single outgoing stream, to reduce bandwidth. In this case, the header contains details of the contributing senders.

A single RTP session provides a 'unimedia' session. A multimedia session is composed of multiple RTP sessions of different media types, allowing receivers to choose only some media types, if all are not necessary. The source identification and timestamp on streams from different sessions allows the receiver to reconstruct them as a multimedia stream.

RTP is used for a number of real-time applications on the Internet, such as many of the MBone tools, including vic [McCanne 95], wb [Floyd 97], ivs [Turletti 93].

2.2.3.3 XTP

The Xpress Transport Protocol (XTP) provides 1:*N* communication, with configurable facilities such as degrees of reliability, independent rate and flow control, QoS negotiation, and can be used independently of the data delivery service. These features are orthogonal to each other, giving maximum flexibility [Strayer 95].

2.2.3.4 RSVP

RSVP is the Resource ReSerVation Protocol, introduced in [Zhang 93]. It can be applied to a mode of communication involving multiple senders and multiple receivers, where the latter specify their individual QoS requirements in the form of RESV messages that they inject into the communication. These propagate towards the senders (who send PATH messages in the opposing direction to inform the communication service of which way RESV messages should go), and allow the intermediate nodes of the communication to take appropriate action to reduce the QoS required by the streams from the senders (including eliminating some streams altogether), in order to meet the reservation specified by the downstream receivers. RSVP is compatible with the resource reservation set-up aspects of planned Internet integrated services for both real-time and non-real-time applications [Braden 94]. Work is underway at Lancaster to use RSVP in routers to instantiate filters to meet reservation requests [Mauthe 97] .

2.2.4.5 IPv6

The Internet Protocol version 4 (IPv4) is used globally to support the internet and transport protocols like TCP and IP. IPv6 is intended to replace IPv4 by supporting extensions, such as a wider address space, and others for contemporary and future network-users' needs. One need is the ability to apply QoS within routers without connection orientation, and so an IPv6 packet sender may identify that its packets belong to a *flow*, while routers that will carry the packets may be configured to apply specific QoS policies to that flow from that sender (the state in the router is discarded after prolonged disuse, *i.e.* it is soft-state) [Deering 95].

2.3 Multimedia Encoding

In their most detailed forms, the new media types of video and audio use up vast amounts of memory and processing power, and for continuous media, a large amount of bandwidth is required from the transporting network. To reduce the demands on networks, compression schemes have been devised for various media types to reduce their bandwidth. This is often *lossy* — these schemes rely on the human viewer or listener being unable to distinguish small deviations between the original and copied data; alternative schemes that compress large chunks of similar information into less data are *lossless*, and are necessary for old media types that can tolerate no errors.

For data which needs both timely delivery and some reliability, *forward error correction* (FEC) is often employed: critical parts of the data are transmitted with extra redundancy. Other data that is copied with full reliability and no timeliness requirements can use other compression schemes.

For the purposes of transmitting a real-time stream to multiple destinations, some high-bandwidth encoding is hierarchically layered, *i.e.* it is split into several discrete quality levels which can be transmitted separately and recombined to give increasing media quality. The higher-level transmissions are only useful if the lower-level transmissions have been received. A receiver that does not want the full quality can instruct its network service to drop packets containing the data necessary for the higher quality levels, reducing bandwidth and possibly cost to the user.

The rationale of [Yeadon 96b] is to provide multimedia filtering both within flows of a single format, and in converting the format used in a flow from one type to another (transcoding), and so has a chapter (4) summarising various international standards for video and audio encoding, and compares them.

2.3.1 Video Encoding

Still images are stored digitally in computer memory by dividing up the image (supplied by a camera) into a two-dimensional matrix of cells, and then digitally sampling the brightness of each cell in three primary colours (red, green and blue, or one luminance and two chrominance). The reverse process is used to render an image on screen or paper. The quality of this reproduction compared to the original image is improved by using greater resolutions for the cell matrix and for the brightness sampling. Motion video is digitised in the form of several still images (frames) recorded in regular sucession, and then rendered sequentially.

High-resolution, full-colour video occupies a huge amount of memory, so there is great demand for extreme compression to reduce bandwidth over transmission. A digitised version of PAL video (833×625 pixels; 25 frames per second) uses 12 million pixel samples per second, and assuming 24-bit colour resolution, that gives a bit rate of around 300 Mbs^{-1}; even low-resolution video suitable for video conferencing (160×120 pixels; 5 fps; 8-bit monochrome) requires around 800 Kbs^{-1} [Yeadon 96b].

Firstly, lossy compression schemes can take advantage of the limitations of the human eye by removing detail that it is unlikely to detect. For motion video, the limitations are greater (allowing more lossy compression), and in comparing a frame to a temporally adjacent or near-by one, there may be large areas that are unchanged, and do not have to be encoded in both frames (this is known as *inter-frame encoding*). Bandwidth may also be reduced by dropping frames regularly, or simply not capturing them. Within a single image, there may be large areas of the same colour, which may support compression (*intra-frame encoding*), and is applicable to both still and motion video.

The following subsections describe a variety of video compression formats for still and moving images.

2.3.1.1 JPEG

The international standard JPEG (Joint Photographics Expert Group) defined in [ISO 93a] is already widely used in the World-Wide Web to hold still images, usually from photographs (rather than diagrams or icons), since it is designed to be a format for representing 'continuous tone' images. Other requirements that it aimed to meet were to have fidelity and rates of compression comparable with the contemporary state of the art, and to have feasible software implementations. In addition, JPEG has several modes of encoding operation.

The encoding of any image begins with its division into *components*, such as three colours (*e.g.* red, green and blue signals, but more usually one luminance with two (red and blue) chrominance signals), but JPEG does not specify a limit, for flexibility. Each component is then encoded independently, according to the choice of the various modes. Finally, the generated streams for each component are combined to form the whole encoding stream, interleaved according to the application's choice.

All modes of encoding involve a final stage of entropy encoding such as Huffman or arithmetic. Previous stages may be transformations, quantizations or predictions, which improve the compressive effects of the final stage.

An application generating a JPEG image may choose *lossy* or *lossless* encoding, the former producing a compressed data stream that can be decompressed to give a controllable approximation to the original image, and the latter producing one from which the input image can be exactly regenerated.

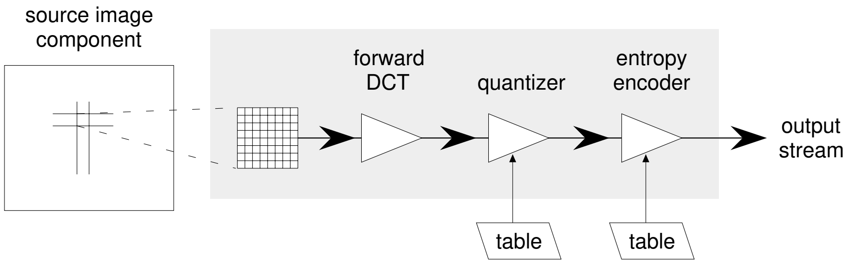


Figure 2.5: Lossy compression using DCT and quantization

Lossy encoding involves dividing the image into 8×8 blocks of 8- or 12-bit pixels, and applying a discrete cosine transform (DCT) to it (this stage is still lossless, though an implementation will only be able to give an approximation; JPEG defines an accuracy test to limit this), and produces an 8×8 table of co-efficients. Each of these is then quantized according to an application-defined, 8×8 quantization table. This is where lossiness is introduced, as quantization is a many-to-one mapping.

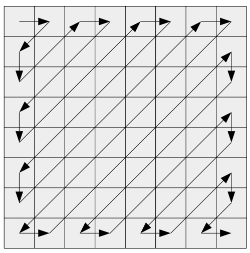


Figure 2.6: Zig-zag ordering of co-efficients

The quantized co-efficients are then sequenced according to the zig-zag pattern illustrated in figure 2.6, and this sequence is submitted for the common stage of entropy encoding. The lossy compression process is summarised in figure 2.5.

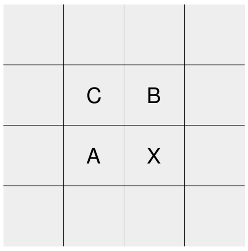


Figure 2.7: Pixels used for prediction

The final stage of lossless compression is preceded by a prediction stage: the value of each 2-16-bit pixel (X in figure 2.7) is compared with an expression derived from the values of three of its neighbours (A, B and C), and the difference is passed to the entropy encoder, as shown in figure 2.8.

Lossless compression achieves ratios of around 2:1, while lossy compression can do much better: as low as 0·25 bits/pixel ‘moderate to good’ quality down from 24 [Wallace 91].

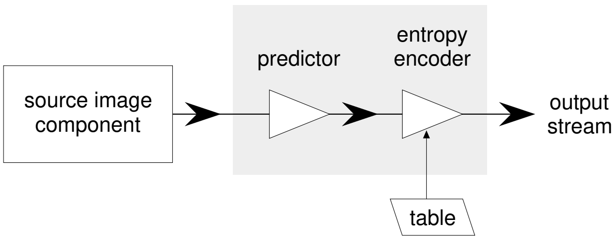


Figure 2.8: Lossless encoding using prediction

Progressive encoding involves retention of an image-sized buffer before the entropy encoding stage. Once full, it is entropy encoded in multiple scans, for example, by selecting from each sample a different range of bits per scan: the more significant bit ranges are encoded in earlier scans. This allows a picture being transmitted over a low-bandwidth network to be built up in several passes, which may be preferred by the viewer. *Sequential* encoding is the mode of operation complementary to progressive.

Hierarchical encoding allows the encoded image data to be divided at various points such that an image can be extracted from the data before the division, while the data following can be used to increase its resolution. This allows a receiver of the data for a JPEG-encoded image to afford to lose the information following the division (due to limitations of its memory or processing power, or of the carrying network), while still displaying a useful image. Furthermore, it can select the point of division independently of other receivers, as in a multicast distribution of the image. Hierarchical encoding is achieved by repeatedly passing a prediction of the source image scaled down in resolution to a normal JPEG encoder. The scaling is gradually altered until it reaches identity, and the prediction used is the image obtained by decompressing the output of the previous pass. Although this produces data for as many images as there are passes, the prediction of the previous pass should reduce the entropy of scaled image of the current pass, so they compress much better than an equivalent number of full-quality images. The process is summarised in figure 2.9:

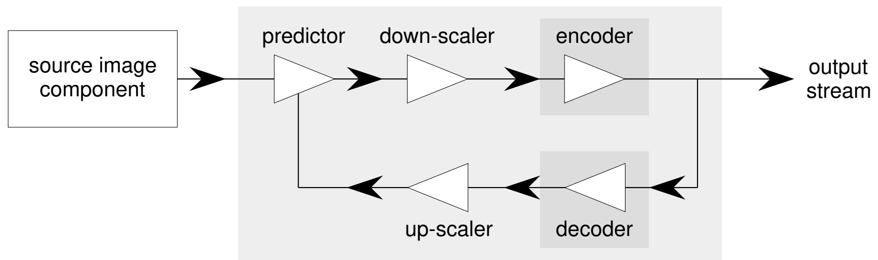


Figure 2.9: Hierarchical encoding

Compressed data from separate components of the same image may be sequenced one after the other, but to assist decoder implementations where display and decoding of multiple components can happen concurrently, JPEG allows parts of one component's data to be interleaved with parts from other components. Figure 2.10 illustrates three co-located 8×8 blocks in separate components being interleaved in the output stream.

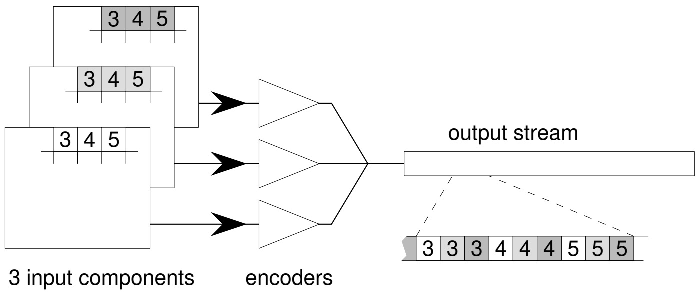


Figure 2.10: Component data interleaving

2.3.1.2 MPEG

The MPEG (Moving Pictures Expert Group) standards are intended for the storage and transmission of motion video, and the audio that may accompany it. MPEG-1 [ISO 93b] aims for maximum bandwidths of about 1.5 Mbs^{-1}, to fit into small media devices such as CDs, etc. MPEG-2 [ISO 94] provides higher qualities for application domains that do not have such restrictions. MPEG standardization aimed to avoid the appearance of *de facto* standards that would be incompatible [Le Gall 91].

In its generic design for representing video, MPEG recognises the asymmetry of some applications, *e.g.* video-on-demand, where the processing capabilities and requirements of the encoder differ greatly from that of the decoder. Other, symmetric applications are more evenly balanced between encoding and decoding, such as video mail and conferencing.

The MPEG format is designed to have good compression while supporting such facilities as random access, searching, editability and error robustness, through the use of reference points that are not highly dependent on other parts of the MPEG stream. The trade-off between processing delay and image quality is also configurable.

Video frames are encoded into MPEG format using techniques to reduce both temporal and spatial redundancy, motion compensation for the former, and DCT-based compression for the latter (similar to JPEG compression). These are inter-frame encoded frames, since they are dependent on other frames. MPEG also has intra-frame encoded frames where motion compensation is not used, and such a frame can be used as a reference point.

Motion compensation, as illustrated in figure 2.11, is applied to each 16×16 *macroblock* of pixels. A suitable motion vector is derived for each macroblock which describes a translation of pixels onto the macroblock's co-ordinates from another frame that is temporally related to the current. The pixels in the macroblock are predicted from the translated pixels, or interpolated from two sets of translated pixels, before spatial compression is applied.

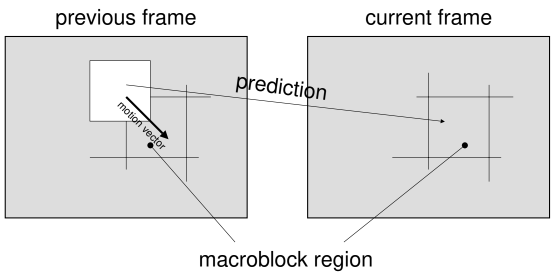


Figure 2.11: Motion compensation

Spatial redundancy is reduced through a DCT transformation of 8×8 blocks, followed by quantization, and then Huffman entropy encoding. Quantization tables can be varied for different types of blocks: intracoded frames may need finer quantization than intercoded frames.

MPEG frames take one of three forms, I-, P- and B-frames.

- I-frames are stand-alone, intra-frame encoded frames.
- P-frames are 'predicted' frames — inter-frame encoding based on the previous I- or P-frame.
- B-frames are 'bi-directionally predicted' frames — inter-frame encoding based on the next P-frame and previous I- or P-frame.

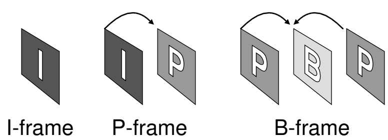


Figure 2.12: MPEG frame dependencies

Because of its dependency on both earlier and later frames, a B-frame cannot be decoded until after a to-be-displayed-later frame arrives. For this reason, P-frames are sent before the B-frames that are to be displayed before them. Figure 2.13 shows a transmission sequence of 10 frames displayed as IBBPBBPBBP.

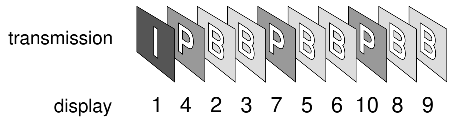


Figure 2.13: MPEG B-frame ordering

The delay introduced by re-ordering is often acceptable when MPEG is used for recorded video rather than live video — all the frames exist long before they are needed for display, and can correspondingly be delivered early, but the sequence is configurable for live applications.

Figure 2.13 is an example of a *group of pictures* (GOP), which is the largest unit that has no dependency on other parts of the MPEG sequence. GOPs can be used for fast, manual searching through a recording, as there is always one I-frame which can be displayed whichever direction the search is heading.

MPEG-2 extends MPEG-1 (and so codecs must be backward-compatible) to allow higher resolutions, and supports filtering and receiver heterogeneity by being scalable in its quality. The base layer defines a minimum quality, while a number of enhancement layers may be combined with it by the decoder to gradually improve the quality in a number of dimensions (*e.g.* spatial resolution, signal-to-noise ratio (SNR)). In transmission, each network packet can belong to a distinct layer in each dimension, in such a way that a network can easily choose to drop higher layers to save resources, or if indicated by the receiver(s).

2.3.1.3 H.261

The H.261 standard [ITU-T 93] has been developed to support live video (and audio) for rates of $p \times 64$ Kbs^{-1} (p is 1 to 30) at around 30 fps [Liou 91]. Compression and decompression are intended to be done in real-time. Applications for H.261 include video-conferencing, and video mail.

Images can be intra- or inter-frame encoded (using motion compensation), and colour is encoded as 4 units of luminance (Y) to 1 each of chrominance (Cb, Cr).

Each H.261 *block* is an 8×8 array of pixels which are optionally predicted from the previous frame, and are DCT-transformed and linearly quantized (similar to JPEG). The step size of quantization can be varied to alter compression speed, in order to meet live compression requirements, with a corresponding reduction in quality. A *macroblock* consists of 2×2 Y blocks, overlayed with 1 Cb block and 1 Cr block. Macroblock are grouped together into *groups of blocks* (GOBs).

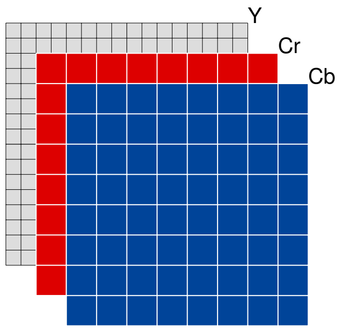


Figure 2.14: H.261 macroblock

Two scanning formats are specified: CIF (Common Intermediate Format: 352×288 luminance pixels; 12 GOBs) and QCIF (Quarter-CIF: 176×144 luminance pixels; 3 GOBs). The precision of these formats assists the real-time requirements of coding and decoding [Yeadon 96b].

H.261 has been implemented in the INRIA Videoconferencing System (IVS) application [Turletti 93] which can run across an IPv4 multicasting service.

2.3.1.4 Other Video Formats

Other proprietary formats for motion video exist. Microsoft's video encoding is the Audio-Video Interleave (AVI) format. QuickTime is Apple's video encoding format. Recorded videos in both these formats are common in the WWW.

2.3.2 Audio Encoding

Sound waves can be converted into analogue electrical waves through a microphone. These can then be digitised for computer-processing by sampling the amplitude of this signal at very high frequencies (in the order of several kHz), giving a signed, linear value for each sample. Lossy audio encoding schemes rely on the statistical properties in the distribution of sample values, and less frequent values are rounded together to the same value. Logarithmic functions are often used to model these distributions.

Varying qualities of audio streams require varying bandwidths. Telephone quality audio can be carried on 64 Kbs^{-1}, using 8 KHz samples. High quality audio needs 1.4 Mbs^{-1} [Mauthe 96]. There have been studies of the relationship between encoded audio quality and perceived audio quality [Watson 97].

μ-law encoding is part of the CCITT recommendation G.711. It uses a logarithmic function to give more precision to sample levels of greater significance. The sign is first extracted from the sample and encoded separately, while the μ-law logarithmic function is applied to the magnitude, and encoded into fewer bits. A signed, linear sample, *x*, of typically 14 bits can be compressed into 8 bits, *m*, with the following expression:

$$m = \mathrm{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)}$$

Figure 2.15: μ-law conversion function

The A-law encoding system uses a similar function to μ-Law. A variety of other audio formats exist, and are found frequently on the WWW. For example, WAV is a common alternative format for short audio sample sequences, RealAudio is used for live transmission and downloading of audio programmes, and MIDI combines sampling with musical control.

2.3.3 Compression of Other Data

Document types that require reliable transmission, such as plain text and programs, can make use of many existing, lossless file-compression utilities. Plain text used in a real-time group interaction (*e.g.* a text conference) is unlikely to have vast bandwidth requirements, so compression is not usually worthwhile. Programs are unlikely to be transmitted in a simultaneous group communication (users will probably download these from Internet/WWW archives temporally independently of each other). This means that the following compression formats are of limited relevance to real-time multimedia group communication.

Unix Compression: Most versions of Unix are supplied with a file compression utility through the commands **`compress`** and **`uncompress`**. After compression, the file is given the suffix `.Z`, and may be reduced in size by about 50%, depending on the redundancy of data in the original file. The compressed file can be uncompressed to give the original file with its original name.

GNU Zip: The recursively acronymous GNU (Gnu's Not Unix) Project includes a compression utility analogous to Unix compression, applying to a single file through the commands **`gzip`** and **`gunzip`**, and using the filename extension `.gz`. GNU ZIP tends to give somewhat better compression than Unix compress [GNU].

InfoZIP: InfoZIP is an archiving utility that retains the structure of an archived directory in an uncompressed form, while applying compression to files stored within [InfoZIP].

3 Group Communication Review

This chapter reviews models, services, protocols and taxonomy applicable to group communication and management. Section 3.1 covers this in general, whereas section 3.2 focuses on a new version of the Media Channel model, enhancing its specification over the first version.

3.1 Group Support

3.1.1 Group Communication Taxonomy

Much work has been carried out to support understanding between researchers by defining a common taxonomy. One such work is ISO's “Multi-peer taxonomy” [ISO 95]. This defines terms to describe groups' populations and topologies, and how these change, and allows some of the terms to be qualified with an arbitrary layer prefix. A summary of the terms used in this thesis is given below.

3.1.1.1 Population

The population of any group can may change over time. A group whose population does not change is a *static group*, while one for which it does is a *dynamic group*. A group where some members may know the identities of all members is a *definite group*, otherwise it is an *indefinite group*.

3.1.1.2 Group Hierarchy

Three levels of group membership are defined: the *enrolment-group*, the *enrolled-group* and the *active-group*. Each can be qualified by a prefix specifying the level, *i.e.* (N)-enrolment-group etc. An enrolled-group is a subset of an enrolment-group. A *closed active-group* is a subset of an enrolled-group, whereas an *open active-group* may extend beyond the enrolled-group.

An enrolment-group is defined by membership criteria, *e.g.* a list of members, or a set of conditions applied to each member. A peer joins an enrolment-group through *registration*, and leaves through *de-registration*.

An enrolled-group is an instance of an enrolment-group. The members are addressable as a single unit. A peer joins an enrolled-group through *enrolment*, and leaves through *de-enrolment*.

3.1.1.3 Communication

The members of an active-group are in communication with each other. An *Active-Group Integrity* (AGI) condition may apply. Failure to meet this condition results in suspension of communication until it is met again, or the end of the active-group itself.

A member of an active group is a *participant*, and a *group-association* (GA) exists between the participants of that group. A *conversation* is a sub-group-association, and specifies an active communication between some of the group-association's members. A group-association is composed of a number of conversations. A conversation may have an arbitrarily complex topology, although four types of conversation are defined. Each one can be seen as a specific case of a latterly defined one.

- *unicast conversation* — one participant sends to another;
- *simplex multicast-conversation* — one participant sends to many others;
- *duplex multicast-conversation* — one participant sends to many others, who also send back to the participant;
- *N-plex multicast conversation* — all participants can send and all participants receive what is sent.

A *C-participant* is a participant of a conversation (this term is distinguished from a participant, which is a member of an active group). A *C-active-group* is the set of C-participants of a conversation, and has an integrity condition called a *C-AGI*.

A group-association composed of many sub-GAs (conversations) may have conditions relating to these sub-GAs that, if not met, suspend or release the GA.

The aggregate of these conditions is the *group-association topology integrity* (ATI).

The ATI of a GA, and the AGI of the GA's active-group are combined in a *group-association integrity condition* (IC).

3.1.2 Group Requirements

Multimedia group applications take several forms, such as public broadcasts of events, audio/video conferencing, shared access to workspace and information (as with Computer Supported Co-operative Work, CSCW), lectures and seminars [Szyperski 93a].

[Almeroth 96] has observed characteristics of real group activities in the MBone. They last varying lengths of time (a few hours to several days), and participants may wish to only view parts of the activity or session, and may in fact be just browsing through available public sessions. In these sessions, the receivers vastly outnumber the senders, are distributed across the world, and may migrate with the local day-night cycle.

Group requirements are characterised in a number of ways by [Mauthe 96]. Network resource requirements are throughput, delay, jitter and error-rate, and some multicasting services already support these (ATM, ST-II).

Groups have topology requirements. In general, group communications involve M:N transfer of data, and some applications (such as conferencing) require all of this, while others only need special cases of 1:N (multicast) connection (*e.g.* television) or 1:1 (unicast) (*e.g.* telephone call). An M:N connection that combines the M streams semantically into one is a *multipeer connection*. An M:N connection may be emulated using M 1:N connections, and is known as a *virtual multipeer connection*.

Existing unicast services (*e.g.* TCP) can easily determine that a connection is unsuccessful since there is only one other end-point that can accept or reject it, or that can discover unreliable transmission, but multicast and (virtual) multipeer services need more sophisticated semantics, since there are many intermediate assessments of success between no reliability and full reliability of transmission to all receivers, or to particular ones. These are the integrity conditions as defined in section 3.1.1.3.

Group applications may involve several connections/conversations of various topologies (a session), and may not fully use all of them at all times. To reduce network load, a group application should be able to specify sharing and prioritising of network resources, the latter to allow intelligent networks to drop less important packets during congestion.

Applications may have ordering requirements for data delivery, typically:

- no ordering;
- source ordering — a source's packets are ordered as the source sent them;
- causal ordering — an application-specific ordering for one or more senders;
- total ordering — ordering at transmission by all senders is preserved at reception.

This is an end-to-end issue to be dealt with from the transport layer up.

Individual participants may want to see different views of shared data, or have end-system, network or cost limitations that force them to receive lower quality.

Management of participation in group applications is also needed. Participants may need to be identified, excluded from some activities (*i.e.* from enrolment groups) and invited to others. Invitation may amount to announcement of an activity in a directory. Participation may be moderated by particular individuals, or may employ voting systems, so group management may require distinguishing of participants through their roles.

3.1.3 Related Work

A number of research projects exist in the field of group management. Some described here are conferencing facilities (MICE/MERCI, BERKOM MMC), while others are more abstract or general services (GCommS, Tenet, GMS).

MICE: Beginning in 1992, the *Multimedia Integrated Conferencing for*

European Researchers project (MICE) set out to provide multimedia conferencing within Europe between various pilot sites, co-ordinated by UCL in London. Existing conferencing technology from other projects were to be used, with the focus on their integration towards a conferencing service [Handley 93] [Sasse 95]. The project sought to provide conferencing to high- and low-end workstations, and conference rooms, across both ISDN lines and the Internet. A centralised component, the CMMC, was used to provide facilities such as multiplexing of video and audio streams, transcoding between different media formats, interconnection between various networks, filtering for low-end systems, and resource reservation and control [Handley 94]. Components of a multimedia conference communicate conference control information across IP multicast [Handley 95]. MICE finished in 1995 and was superseded by MERCI.

MERCI: This project, consisting of many of the partners of MICE, continues the MICE goals of multimedia conferencing over the Internet, but extends its reach to implementations on PCs, and adds facilities for security, recording and heterogeneity of audio/video quality [Kirstein 97].

GCommS: Lancaster University's Group Support for Distributed Communication Systems project aims to develop support for multimedia group communication. The project observed characteristics of group/multicast communication in existing systems, such as distributed systems, operating systems and networking systems, and compared with those of existing group applications. Requirements of multimedia applications in general were characterized, and the limitations of the existing multicasting systems to support them were identified [Mauthe 94,95,96]. The M-Connection Service [Rezende 95] is a product of GCommS to support 1:*N* multicasting with configurable degrees of group reliability.

GCommS also considered group management, initially in the form of a group database. The first version of the Media Channel model [Namuye 96a/b], to be revisited in section 3.2, also forms part of GCommS, and focuses more on the unification of the management of group applications so that they can easily be treated as components of a multimedia collaboration. It allows group applications to choose their own multipeer transport services, while giving them connection instructions to form a (virtual) multipeer interaction between participants. The model may collect enough information about participation in an instance of an application (now referred to as a media channel) to support enforcement of policies such as permitted participants, and connection integrity. The controlling of connections and enforcement of policies then do not have to be fully re-implemented per application, but merely reconfigured. Applications are unified as media channels so that a user's participation in a multimedia group session can be controlled through an agent which treats channels almost identically with only a small amount of configuration for channels of different types.

The Tenet Group: This group at ICSI, Berkeley is concerned with network support for multiparty interactive multimedia (MIM) applications, as characterized in [Szyperski 93a], from which efficient multicasting schemes, such as the half-duplex real-time channel [Syzperski 93b/4], and a simplex real-time channel with resource sharing between channels [Gupta 93a], have been derived. Inter-channel relationships can be specified from end-systems through the use of channel groups [Gupta 93b]. This approach allows resource sharing to be application-defined, providing better QoS guarantees than if the network performed statistical multiplexing of streams on a common resource, which would make the quality depend on other, unrelated, streams [Gupta 95].

BERKOM MMC: The Berliner Kommunikationssystem Multimedia Collaboration Service provides audio/video conferencing to its users, along with shared-application participation to support distributed, collaborative working [Altenhofen 93]. Intended users may have workstations from different vendors, supporting different windowing systems (*e.g.* Windows, X), and have access to group management facilities including a Conference Directory (CD) to hold mainly static information about users, groups and conferences, and a Conference Manager (CM) to deal with the dynamic aspects of conference participation and format. Users view conferences through their local Conference Interface Agents (CIAs), which access the relevant CDs and CMs, through local Audiovisual Components (AVCs), and through local Application Sharing Components (ASCs) which make local applications accessible to other participants. Data between AVCs is carried through the AV Data Protocol which sequences and timestamps streams of audio and/or video data, while connections between AVCs are co-ordinated by and Audiovisual Manager (AVM).

GMS: The Group and Session Management System by Wilde et al [Wilde96,96a,96b,97], like GCommS management, is an attempt to allow group applications for distinct purposes to be managed together as a co-operating unit, firstly by implementing the common aspects of collaborative applications independently of their choice of transport system, and secondly, by providing a widely distributed database to hold relations between users, groups (of users or groups), flows (of application data), and sessions (collections of flows with management and admission control). Forms of authorization to join sessions are defined: open to anyone, limited to a group, or permitted by confirmation by managers.

GMS takes a distributed approach to management from the outset by using domains populated with distributed GMS Session Agents (GSAs) which act as a distributed database and collaborate to perform the management function using a symmetrical GMS System Protocol (GSP) [Wilde 96a]. These are organised into a hierarchy of domains, with the top-level divided into several domains to avoid a bottleneck.

Users of GMS access it through a local GMS User Agent (GUA), which communicates with a nearby GSA through an asymmetrical GMS Access Protocol (GAP) [Wilde 96b] which can run over any reliable, connection-oriented transport service. An API (such as a code library) is provided to cover the GUA access and application-specific transport service.

3.1.4 Group Communication Support

Within a multimedia group session, a large quantity of data will likely be transmitted to several recipients at once, so in order to support this group communication, transport services must exist to multicast data. To reduce the over-all load on the network from transmitting the same data multiple times across segments, these services must be optimised to require data to travel as few times as possible over each segment.

This section describes some existing and planned network services and protocols, and then shows how the interfaces to these can be summarised in a limited number of *multicast addressing modes*.

3.1.4.1 Multicasting Services

IP Multicasting and MBone

The IP network service has a multicasting extension. It allows groups of destinations to be identified through single multicast addresses (class D IP addresses which have their four most significant bits set to 1110), and destinations can add and remove themselves from groups at will (*i.e.* the groups are dynamic). Some addresses are permanently assigned to particular groups (which may currently have no members), while others are transiently assigned for the lifetime of the group.

IP packets delivered using IP multicasting have the same QoS as unicast packets: they may be lost without notification, delivered out of sequence, and may be delivered multiple times.

Receiving data in a multicast group is connection-oriented: a host receives data between joining and leaving the group. To join a group as a destination, a host binds to the group address, initiating Internet Group Management Protocol (IGMP) messages between it and the local router [Deering 89]. The router will now deliver multicast packets that specify the group address as the destination. Sending data to an IP multicast group is connectionless: the sender merely specifies the group address as the destination (per packet), and the sender does not have to have any association with the group.

IP multicasting is supported by a virtual network of multicast capable routers called the Multicast Backbone, or MBone [Eriksson 94]. The routers, called **`mrouted`**, connect to each other through 'tunnels' across the real network paths, as seen in the following figure:

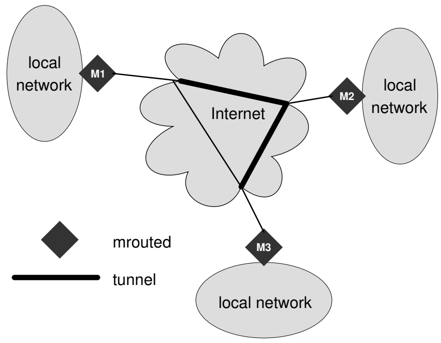


Figure 3.1: MBone topology

MBone has been used to develop multimedia group applications, usually for audio/video conferencing. These have been used in turn to discover how such applications are used [Almeroth 96].

Scalable Reliable Multicast (SRM) by [Floyd 97] uses IP multicasting (or a similar service) to allow detection of lost packets in an *M*:*N* communication. It uses careful naming of application state, and sequencing of messages used to indicate state changes. The sequencing is used to detect lost packets, and issues a negative acknowledgement to the multicast address. Timestamps in the packet determine how long a receiver should wait for other receivers who may also have lost the same packet to transmit an acknowledgement, before it sends its own, and the naming scheme ensures that a receiver can tell that a passing acknowledgement contains the same data that it intends to send (so it doesn't need to load the network with identical data). SRM is used in the wb (whiteboard) MBone application.

ATM Multicasting

Asynchronous Transfer Mode (ATM) networks provide virtual circuits (VCs) between end-systems across a mesh of ATM switches, whereby, after configuration of appropriate switches, 53-octet packets (ATM cells) may travel across the network with minimal addressing information, taking the same route and arriving in order. The cell header includes a VC identifier (VCI) which indexes an entry in a table for a particular port of a receiving switch, and the entry specifies an output port and a new VCI. The cell's VCI is replaced by the new value, and transmitted through the new port, hence the VCI may change on each hop. For an ATM network to support multicasting, its switches must be able to map an incoming <VCI, port> tuple to more than one outgoing tuple.

One difficult issue of ATM concerns the initial establishment of virtual circuits. This is addressed in one form as the specification of a User-Network Interface (UNI) between an ATM switch at the edge of the network, and a client or customer's terminal or local network. This relies on signalling between switches to enable a route to be plotted across the network from one end-system to another, so that the tables in the relevant switches are correctly configured. UNI 3.1 introduced multipoint VCs: the root/sender of a 1:*N* circuit can add or remove leaves/receivers to/from the circuit during its lifetime (root-initiated join/leave) [McDysan 94] [Diot 97]. More recently, UNI 4.0 allows for leaf-initiated control [Wilde 97].

ST-II

ST-II is the Internet Stream Protocol, version 2, as defined in [Topolcic 90]. It sits at the network level, alongside IP, but provides connection-oriented transmission through streams, and stores state information about streams in intermediate nodes (ST agents).

Streams are simplex 1:*N* trees, with data flowing from the root, or *origin*, to the leaves, or *targets*. 1:1 streams are not handled in any special way. The origin can add and remove targets to and from the stream during its lifetime, and targets can remove themselves from the stream. If the origin leaves the stream, it is destroyed.

ST-II allows network and component resource reservation: the origin specifies a *FlowSpec* for the stream, incorporating bandwidth, error rate, delay, *etc.*, and may specify new FlowSpecs later on. Flow control and packet ordering are expected to be handled in a higher layer.

Aggregates of streams can be grouped and identified by a globally unique “Group Name”, allowing ST agents to recognise streams belonging to the same group, and sharing their resources.

IPv6

The Internet Protocol has undergone a revision to meet increasing needs of its users, and its specification is known as IPv6 or IPng (next generation) [Deering95]. Firstly, its addressing range has been extended from 32 bits to 128 bits, to defer exhaustion for assignment to new hosts. It also incorporates some QoS aspects by identifying packets as belonging to flows (identified by an opaque number unique to the source), and associating these flows with soft-state (which disintegrates after a period of disuse if not explicitly cancelled) inside routers.

With the extended addressing comes increased scope for multicasting. IPv6 multicast addresses have 8 bits to distinguish it from other classes of address, 4 flag bits, 4 scope bits, and 112 bits to identify the host group [Hinden 95]. The scope allows addresses to be allocated with varying levels of locality (*e.g.* node, link, site, organisation, global). For example, if a host group is not expected to extend beyond a company network, an address from the scope 'organisation' may be chosen. Alternatively, the scope allows a packet to be addresses to a local subset of a host group. Routers at boundaries between levels of locality will drop multicast packets whose scope does not permit them to move in the higher level.

HTTP

Although HTTP uses a request/response model of interaction to transfer documents of arbitrary types from a public archive to many users, and each user interacts with a HTTP server independently (*i.e.* at different times), this is a form of multicasting. It is not efficient because it is made up of several separate unicast connections whose routes may traverse common network sections. The other multicast services already described are not very applicable to this situation as they deal with transmissions to several destinations at the same time.

However, HTTP supports some multicasting by permitting *proxies* between the client and server. The proxy acts as a server to the client and as a client to the server (so the original client and server behave as though there was no intervening component), and the proxy forwards requests from the client to the server, and forwards responses oppositely.

If the proxy receives a request from one client while awaiting the response from the server for a compatible request from another client, it can avoid sending the second request to the server, and send the same response to both clients once it arrives.

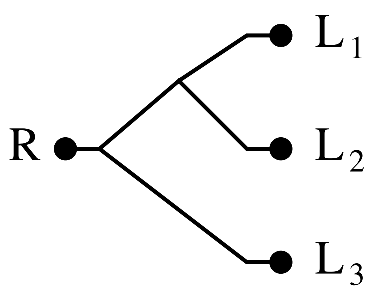
Furthermore, if the proxy stores responses in a local cache, the clients' requests do not have to overlap — before forwarding the request to the server, the proxy looks for a compatible response in the cache, and if found, returns it instead. Cached responses are eventually discarded to reduce the space required, so that it optimally holds only the currently most desired documents [Fielding 97].

3.1.4.2 Multicast Modes

The interfaces to the services in section 3.1.4.1 can be summarised by a limited number of multicast modes listed below, and are used to define some of the functionality required by the MC model, which must provide participants with enough information to allow them to establish their own multicast/multipeer connections that form a (virtual) multipeer *M*:*N* connection.

Root-initiated Multicast

The multicast service sets up a dissemination tree with a single root, and establishes paths to leaves identified by connection instructions from the root. Essentially, the root must know the addresses of all the leaves.



$$R: \{L_1\ L_2\ L_3\}$$

Figure 3.2: Root-initiated Multicast

The term ‘root-initiated’ is preferred to ‘sender-initiated’ as it does not imply any particular direction of data flow.

The expression r-init will be used as a mnemonic for this multicast mode. Naïve multicasting involves supplying the sender with the list of addresses, so it uses r-init mode. Similarly, ATM multicast can be established from the root by issuing connections to the leaves [McDysan 94].

Leaf-initiated Multicast using Multicast Address

A multicast connection is referred to by an allocated multicast address. Leaves join the connection by binding to the address. Roots merely send to the address. Both roots and leaves must be supplied the address of the connection. The root may select the address itself.

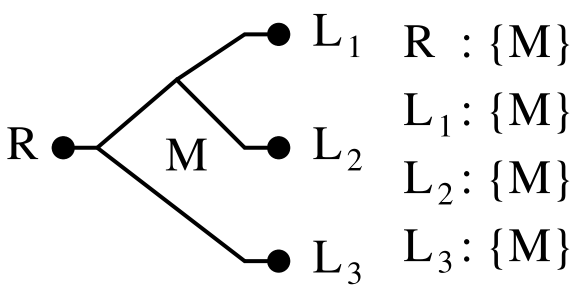


Figure 3.3: Leaf-initiated Multicast using Multicast Address

The expression l-init(m-addr) will be used as a mnemonic for this multicast mode. IP multicast is configured using allocated multicast addresses supplied to both root and leaf [Deering 89], so it has a l-init(m-addr) mode.

Leaf-initiated Multicast using Root Address

A multicast connection consisting of a single root is referred to by the address of the root. Leaves join the connection by finding a path towards the root until the new path overlaps with an existing path. The leaves must be supplied with the address of the root.

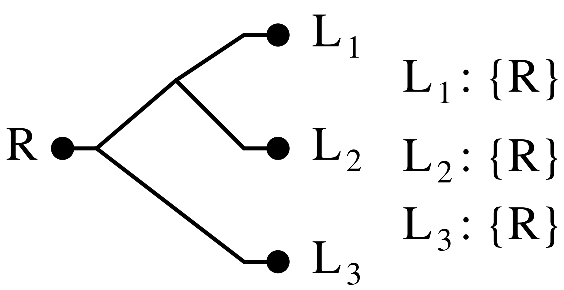


Figure 3.4: Leaf-initiated Multicast using Root Address

The expression l-init(r-addr) will be used as a mnemonic for this multicast mode. Using the root address when binding a leaf to the multicast connection can simplify that initial connection, as the initial connecting packet has a definite direction which assures that it reaches a node already in the connection, and so does not need to proceed further. The Core-Based Tree [Ballardie 95] approach to multicast routing makes use of this, and so could be broadly described as using l-init(r-addr) mode, though, in fact, it still uses a normal multicast address.

Hybrid Multicast Modes and Summary

Other multicast services may have hybrid addressing modes, *i.e.* they appear as different modes to the root and the leaves. For example, a 'third party' – neither the root not the leaves – may be responsible for connectivity over a managed network, given the locations of the end-points. In this case, the third party maintains appropriate connections over the network, and informs the end-points of the access points to the network that connect to those links.

Since the interactions required to instruct end-points in connection for this sort of mode are the same as some of those of each of the other modes, no extra interactions need to be defined. In summary, the following interactions with management are required:

- the root supplies an address of its own on/to which it will transmit;
- the root is supplied with an address to which it will transmit;
- the leaf supplies an address of its own on which it will receive;
- the leaf is supplied with an address to which it must bind or from which it must receive.

It is assumed in the first and fourth case that the ambiguity of behaviour can be resolved by identifying whether the address is a multicast address.

3.1.4.3 Group Management Services

Many of the services and protocols in §3.1.4.1, while supporting a basic function of a multipeer service, multicasting, do not provide facilities that multimedia group applications require, such as QoS filtering, multipeer integrity, and virtual multipeer communication for conferencing. Group management includes these facilities, and a number of schemes have been defined to support them.

[Heijenk 94] defines the notion of a call to represent a multimedia group session with the different media as separate internal components of the call, to which users attach in varying ways (IN, OUT, IN-OUT). Relationships may be defined between media, such as synchronisation of audio and video, and these relations also form part of the call. It defines specialised rôles for users with permissions to modify the call.

[Vin 92] describes a more abstract model of a session which focuses on the timing of data in order to unify abstractions for synchronous and asynchronous collaborations. Streams are 1:*N* communications between participants who belong to defined lists, and sessions are sets of streams. Conferences describe a series of sessions as the levels and types of participation change during a collaboration.

The Real-time Transport Protocol (RTP) [Schulzrinne 96] supports group management in that participants send each other RTCP reports that identify themselves (SDES reports), as well as describing the quality of the data being received (RR and SR reports). The reports allow participants to individually decide that a session is worth continuing, based on who else intends to participate (according to received SDES reports), or based on the current network quality (according to RR and SR reports). RTP does not enforce any management policies, however, and has difficulties when scaling to large groups.

The Resource ReSerVation Protocol (RSVP) [Zhang 93] allows receivers in a multicast group to reserve network resources to be used by senders' streams. They issue filters that propagate towards senders, so that network nodes can make the necessary reservations at the appropriate places. Since the filters can apply to individual senders, it is possible to eliminate them entirely from reception, assuming intermediate nodes can genuinely filter out their packets. This allows for a certain degree of participation control, on a per receiver basis.

XTP has group management features, such as receiver identification and exclusion, and measures of reliability of reception among members (key members, *k* members, all members) [Strayer 95]. Extensions to support reliability for fully-known groups is introduced in [Atwood 96], and reliability for very large groups is proposed for XTP through the Local group Concept, where one participant in a local region acts to support local retransmission of data, and collation of return messages [Hofmann 96].

GMS is a totally integrated approach to group management. All collaborative applications work through the distributed GMS components, allowing GMS to perform management across co-operating applications [Wilde 97]. Flow objects within GMS describe interactions between potential users. A flow is either uni- or bi-directional. It permits a limited number of sending users and receiving users, and defines QoS parameters with maximum and minimum values to describe what senders are expected to contribute and what levels receivers may choose from. The flow also contains addressing information for participation that is updated as users join and leave the flow. Root- and leaf-initiated addressing modes are both supported: the flow contains either a changing list of receiver addresses, or a static multicast address.

Sessions, being sets of flows, allow several flows to be joined and left in a single atomic operation. Each flow belongs to exactly one session, and participating in a session is the only way to participate in a flow. Sessions can have various access permissions, and join and leave operations are subject to them, given the user's authentication (which may involve passwords, challenges and responses, or just the user's claimed identity for very light security).

3.1.5 Group Naming

Assuming that a widely distributed group management service must involve widely distributed components that work coherently to produce the service, clients must connect to a local component of the system. For the particular notion of group that the service provides, the client and the local component must use globally unique identifiers for groups that are recognised by other clients attached to the other components, so that the service components can be sure that clients remote from each other are participating in the same group. This aspect of group support is referred to as *group naming*. This section discusses various schemes and mechanisms employed in a number of contemporary models, and one to be proposed for the model described in this thesis.

Group names allow networks to share resources between communication streams, by identifying that they belong to the same group. They can also be used to set up connections, helping to locate the nearest of the group service's components.

Distributed processing environments (DPEs) have been used to support group communication [Davies 92], where the interface reference of a single object in a distributed system is extended to refer to multiple objects that can share the responsibility for responding to an invocation on such an interface. Being an interface reference, it can be placed in the usual ANSA naming service, the trader, where references are identified by arbitrary attribute-value pairs (properties), and selected through expressions in terms of those attributes [APM 92, §3.11].

ISIS is a DPE with the notion of group processing at its core, serving a number of purposes. One is to share processing responsibility across processes, supporting concurrent processing. Another is to support fault-tolerant processing through replication. Processes join process groups, and tasks are assigned to them to distribute or replicate the function of the group. ISIS groups are addressed through a unique 24-byte sequence, and also have a symbolic pathname [Birman 89].

IPv4 uses a subset of its total range of addresses to identify multicast addresses (32 bits) [Deering 89], and in the forthcoming enhancement, IPv6, which extends the addressing range (to 128 bits) to cope with a greater number of addressable network components and terminals, the multicast addresses are also extended [Hinden 95]. The eight highest-order bits are set to mark an address, followed by some flags (*e.g.* to indicate the addresses' transience), a scope, and the remaining 112 bits form the group's identity. While IPv4 uses its time-to-live (TTL) value in the packet header to limit the range of multicast packets [Deering 89], the scope of IPv6, incorporated into the header, prevents multicast packets from travelling beyond defined boundaries such as *site*, *organisation*, or *link*.

Uniform Resource Names (URNs) are defined to provide global naming for groups of objects that should have the same content on the World-Wide Web

(WWW) [Sollins 94]. If a resource has been replicated around the globe to reduce the load on a server that would otherwise hold the only copy, or to allow browsers to reach the document more locally, it is necessary to direct browsers to the nearest copy, or to distribute their requests to any copy. References to the resource should not refer to any one particular copy, as this concentrates the initial requests by browsers onto the same server, and the user must then manually choose a convenient copy, before the benefits of distribution can be appreciated fully. To assist this, the resource can be associated with a URN, which replaces the location-dependent URL in the reference. Then the browser may pass the URN to a resolution service which will convert it to the URL of any one of the copies, possibly a local one. DNS is expected to be used to implement this service [LaLiberte 95]. If a URN is used to identify a group, it could be made to map to the nearest component supporting group communication and management.

3.2 Media Channel Model

This section defines a new version of the Media Channel (MC) model, the first version having been described briefly in [Namuye 96a] and more fully in [Namuye 96b]. Some example configurations are given, and the model is also described in terms of the Multipeer Taxonomy described in section 3.1.1. Finally, the achievements and limitations of the model in this form are discussed.

3.2.1 Taxonomy and Services

3.2.1.1 Overview

The aim of the MC model is to manage the multiple connections between the end-systems used by the participants of a multimedia group session. The session is first split into a number of *media channels*, each having a particular *media channel type* (MCT) which has an identifying name and describes the generic parameters of the model, making it configurable for many applications. (There is no requirement for them all to have different types.) Each media channel is built up from a number of *streams*, each of the same MCT.

Note that 'media channel type' replaces the term 'media type' as used in

[Namuye 96a,b], as the latter is too general and overloaded.

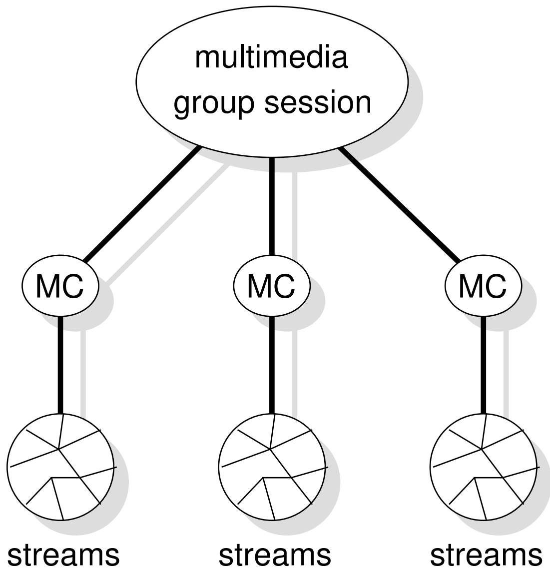


Figure 3.5: Media Channel Hierarchy

Figure 3.5 summarises the hierarchy of components in the MC model. The multimedia nature of the group session is realised through its separation into multiple MCs, each of which deal with a single media type. A bottom-up description of each of the components of the MC model now follows.

Streams

A stream consists of one *producer* and zero or more *consumers*, and the data that flows within it is defined by its media channel type. The number of consumers may change over time. The media channel type defines an application-specific protocol between the users' programs, usually involving data flowing from a producer to the multiple consumers through a dissemination tree, with resolution requests flowing oppositely, and being merged together. These specify the degree to which the main data is scaled down within the network. Figure 3.6 shows an example stream. The precise nature of traffic flow along streams is defined by the MCT, but is always a 1:*N* interaction.

Typically, a producer captures a real-world object (the capture is known as the producer's *data state*), such as a video image, and transmits differences in the state (in a Δ-*state message*) from the previous capture through to the consumers using an encoding scheme specified by the media type. It may also transmit enough information to rebuild the whole captured object from scratch (in a *full-state message*). Again, it is the MCT that gives the precise definition of the format of these messages. Whatever kind of message is delivered, each consumer should receive enough information to rebuild a copy of the object, up to the quality specified by the last resolution request it transmitted towards the producer. This copy is known as a *mirror*.

For example, in an MPEG video stream, full-state messages would hold I frames, while Δ-state messages held P and B frames. The different types of message (Δ– or full-state) is a distinction of the MCT, and the MC service has no interest in them, but the terminology may be helpful in expressing new MCTs.

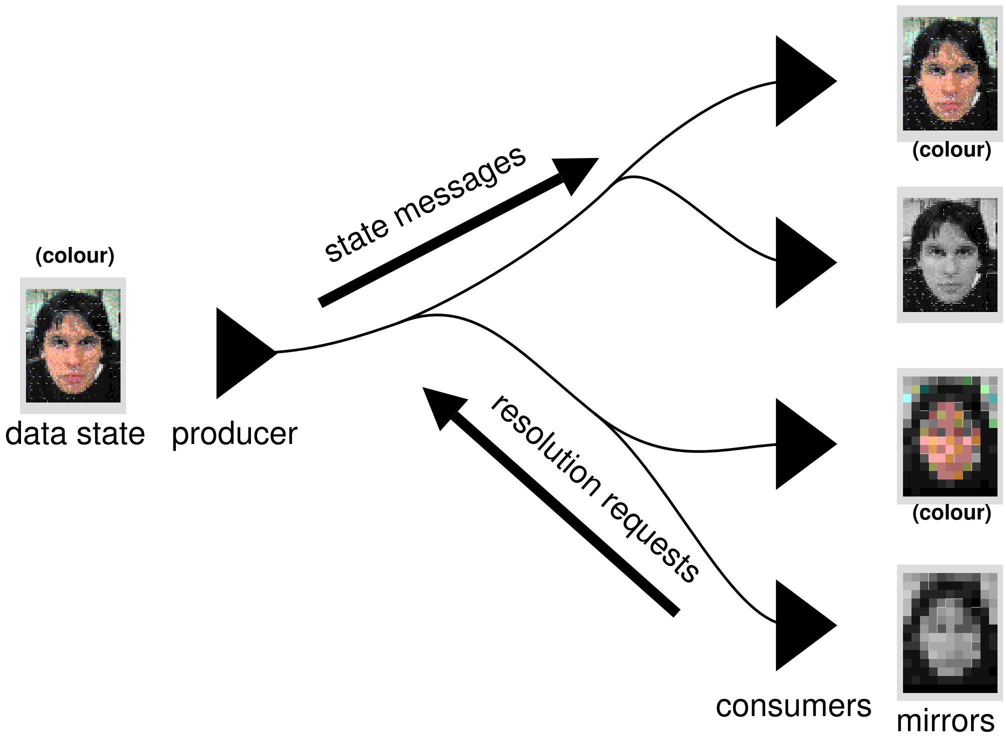


Figure 3.6: Stream structure

Nodes in the dissemination tree may store mirrors themselves, and use this information to inform new consumers of the current data state without having to bother the producer for it, if the flow of data is temporarily static. Resolution requests can be held in the network at nodes in the data's dissemination tree, in order to filter out unwanted data on a per-consumer basis.

Media channel types are named, with a similar purpose to MIME types [Borenstein 93]. A few standard types are supposed to exist, while others can be developed in a local scope. Producer and consumer programs can be selected according to the MCT name of the stream. The standardness of a particular type is measured according to the availability of these programs to potential participants.

Media Channels

A media channel consists of *M sources* and *N sinks*, where *M* and *N* are non-negative integers. Each source consists of one producer which forms part of a stream (so there are *M* streams). Each sink consists of upto *M* consumers, each one connected to a different stream.. Figure 3.7 shows the figurative representations of sources and sinks. All sources, sinks, streams, producers and consumers in a single media channel of media channel type *T* deal only with that type

Figure 3.7: Source and sink notation

Like a stream, a media channel has a *data state* which is the aggregate of the data states of all the component streams, possibly including minimal information on each source, *i.e.* enough information to separate the channel's state into separate stream states. For some MCTs, this extra information may be unnecessary: for multiple audio streams that can be mixed within the network, they all arrive at the sink as a single flow of data, which represents multiple streams, so there are still (upto) *M* abstract consumers, even though the data will likely be passed as it is directly to the audio hardware. A sink may reject particular streams, and so may contain fewer than *M* consumers. Figure 3.8 shows a 3:4 channel with all sinks accepting all the sources' streams.

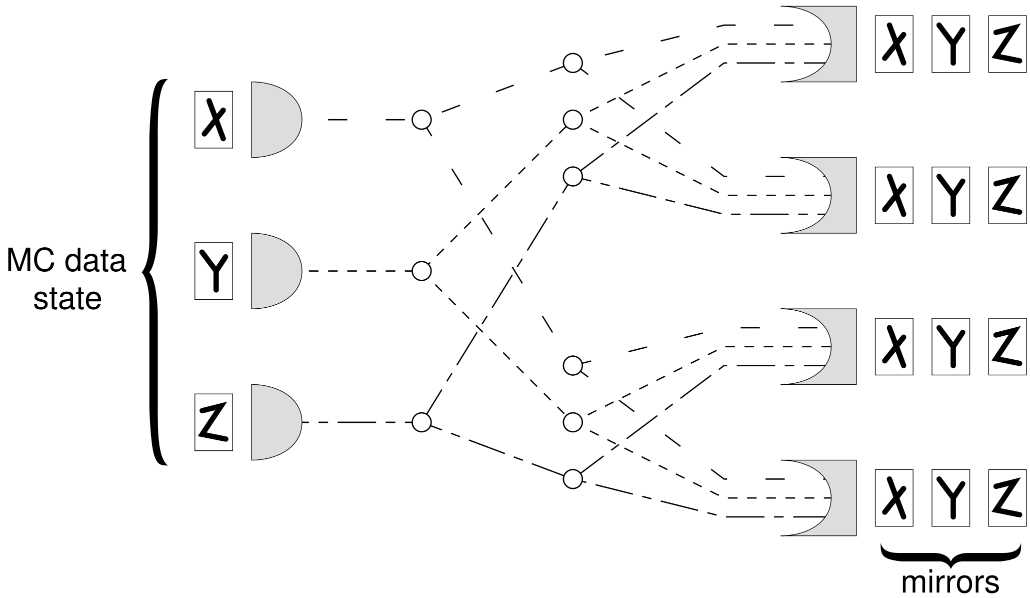


Figure 3.8: MC structure

User programs attempt to enrol and de-enrol themselves with channels at will, so the membership and topology will change over time. A channel with no members can exist indefinitely. Whoever is responsible for establishing a channel is also responsible for tearing it down (though that could simply mean handing responsibility over to someone else).

Several media channels of varying types can now be used to form multimedia group sessions with arbitrary topologies for each media type. To ensure particular topologies, or that only certain users can participate in each channel in a certain way, or to ensure other various integrity conditions are maintained, each channel may be furnished with an arbitrary set of policies which are continuously used to enforce those conditions by distinguishing between those users allowed to participate in a channel and those who are merely registered with it. The policies or their arguments may change over the lifetime of the channel, affecting the participation.

The policies may specify integrity conditions such as the authorised participants, or a quorum of active users required before activation of all streams or individual streams (another case with fewer than *M* consumers per sink), but it is also possible to have a channel with no limitations. The model does not require any particular types of policy to exist in an implementation. Activation of a channel may also be controlled externally, assuming that its integrity conditions are met, and those conditions may be externally observable, allowing some interaction between groups of channels.

User Agents and Local Sessions

The participant of a group session consisting of multiple media channels may be running numerous source and sink programs of various types from an end-system. A *user agent* can be used to manage these multiple components by listing the channels involved, along with an indication of the user's registration in each one (as a source or sink, or both). The user can toggle this state, activating and deactivating the attempted enrolment of the relevant programs.

Channel	Type	Activity
Video conference	`H.261`	
Audio conference	`SunAudio`	
'The Lion King' video	`MPEG-2`	
'The Lion King' audio	`SunAudio`	
'The Lion King' control	`PlaybackCtrl`	

Figure 3.9: User agent display

A user agent's collection of channels and its participation in them, constitutes a *local session*. It is independent of other user agents' local sessions, unlike group sessions described below.

Group Sessions

Several user agents form a *group session* by the coincidence of their participation in the same channels. This makes participation in multiple sessions, hierarchies of sessions, or the temporary combination of sessions trivial. To assist in ensuring that several user agents contain at least the channels involved in a session (so the users are at least participating in that session), an extra channel may be configured, whose data state describes a session (a MCT needs to be defined to support this), and to which the user agents join as sinks. A user agent needs only to join this session channel, and as a sink, it will be given an updated copy of the session description, and can update its table of channels correspondingly.

As well as the list of channels, the session description may contain suggestions for registering with each channel. For some channels, such as video-on-demand, it may be inappropriate for the typical user to participate as a source (and the channel policies may forbid it anyway), so the suggestion accompanying such a channel would be “join only as a sink”. A large audio conference would be another such channel: it would usually be inappropriate for everyone to participate as a matter of course, but a keen user would still be able to activate his source manually. (Malicious participation would be prevented by controlling the enrolment groups in the policies.) The session description may also specify inter-stream and inter-channel relations such as those in [Gupta 93b], *e.g.* audio/video synchronisation, network reservations, media priorities, *etc.*

A group session may have integrity conditions expressed in terms of those of its constituent channels. Failure to meet these conditions causes the channels to be externally de-activated.

3.2.1.2 MC Model using Multipeer Taxonomy

A media channel aims to provide an M:N multipeer connection between a group of M sources and a group of N sinks, with policies to impose limitations on this connection and to determine if establishment or retention of the connection is worthwhile. The channel may be configured so that the connection may be virtual, *i.e.* consisting of $M \times$ (1:upto N) connections, and the policies may be used to reduce the channel to a single, or small number of, 1:N connections.

Firstly, the channel's policies define which end-points may take part in the rôles of source and sink by specifying a restricted MSnk-Enrolment-group for sinks, and a restricted MSrc-Enrolment-group for sources. The channel maintains an unrestricted MSnk-Intention-group which intended sink participants may join or leave at any time in the duration of the channel. From this, the channel derives a

MSnk-Enrolled-group, which is the intersection of the enrolment- and intention-groups; similarly, a MSrc-Enrolled-group is derived from the MSrc-Enrolment-group and the MSrc-Intentions-group:

$$\text{MSnk-Enrolled-group} = \text{MSnk-Enrolment-group} \cap \text{MSnk-Intention-group}$$

$$\text{MSrc-Enrolled-group} = \text{MSrc-Enrolment-group} \cap \text{MSrc-Intention-group}$$

These groups represent those users who wish to participate in the channel, and are permitted to. They may dynamically change as users change their intended participation, or as the policies are changed.

The media channel attempts to maintain a stream from each member of the MSrc-Enrolled-group to all the MSnk-Enrolled-group members, hence the latter group is an open one. Its members may dynamically choose to accept or reject an individual stream based on the identity of the source of the stream and an MCT-defined description of it, hence an active group (MSnk-Active-group) can be derived. A user may change the acceptance state at any time and any number of times.

Each 1:*N* stream, identified by its source, may have an integrity condition, MSnk-C-AGI, which is derived automatically from an MSnk-C-AGI-template, which is a component of the channel's policies. The stream is considered currently unsuccessful by the channel if the stream's MSnk-C-AGI is not met. While successful, the source and sinks will be given addressing instructions that will allow them to be connected according to a communication system defined by the channel's configuration.

An integrity condition in the policies, MC-ATI, may apply to the existence of streams within the channel (key streams are identified by source), and all remaining successful streams may be de-activated while this is not met. An MSnk-AGI may also exist to specify required (numbers of) active participants in general, rather than for particular streams.

Overall the media channel has an integrity condition MC-IC:

$$\text{MC-IC} = \text{MSnk-AGI} \wedge \text{MSnk-ATI}$$

and all streams will be active only when this is met and when an optional, external control permits activation. A channel implementation may enforce soft integrity, where conditions are continually evaluated and applied, or it may be able to have distinct pre- and post-activation integrities, where the former must be met first, and is then discarded in favour of the latter (which is usually less demanding). When that fails, de-activation may be permanent (hard integrity).

Group sessions consists of sets of media channels, and include information on participation and associations between channels, as well as integrity conditions for the session as a whole. Media channels within a session may be seen as ISO multipeer connections/conversations at the session level, with the session's C-AGIs being the MC-ICs. A group session may be equipped with a S-ATI expressed in terms of these C-AGI/MC-ICs, allowing a minimum number of channels, or a specific set of channels to be indicated as required for the session to proceed.

$$\text{S-IC} = \text{S-ATI} = f(\forall \text{MC-IC})$$

Within the context of a session's integrity conditions, streams in a channel will be established only while both the channel's and the session's integrity are maintained.

3.2.1.2 Services

The Media Channel model defines two levels of servicing – one for channels and the other for sessions – in separate administration and participation planes. Channel participation is further divided into MC-source and MC-sink services, referred to together as MC. An overview is shown in figure 3.10:

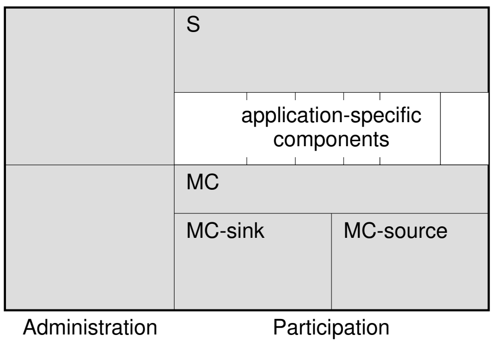


Figure 3.10: Media Channel model services

MC Participation Service

The channel participation service is concerned with the basic activities of a user within an existing channel. If a user needs to re-configure a channel in some way, he must use the administration plane to do so, which may involve very different access rights than required by participation.

Interaction with the service involves indicating a wish to participate as a source or sink (independently), and to observe connection instructions from the service. Sources and sinks identify themselves by specifying the address of an agent that will verify their validity, and provide further identification information. A source must also describe the nature or format of its stream, and with its identification, a sink can make a decision to accept or reject a stream.

Connection instructions come in the form of *address associations* which have unique reference values within the source or sink. For each address association, the client may be asked to supply an address to the service, or to use an address supplied by the service. Additionally for sinks, the address association comes with a description (source identity; stream format) of the stream(s) available through it. In response, it can specify which streams/sources it accepts from that association, and will receive at least those.

Data types used by the participation service include:

- string — a string of characters, used for MCT names
- MC.Agent — the address of an agent authorising a source or sink
- MC.Agents — a variable-length sequence of MC.Agents
- MC.Ref — an address-association reference
- MC.QoSInfo — MCT-defined data to describe a source's stream
- MC.UserAddr — MCT-defined addressing for consumers and producers
- MC.SourceSpec — MC.Agents and a MC.QoSInfo to describe their combined stream
- MC.SourceSpecs — a variable-length sequence of MC.SourceSpec

MC.UserAddr, being a sequence of octets, is usually unprocessed by the service as it passes from client to client, although the service can specify a null address with a zero-length sequence. MC.QoSInfo also will normally be unprocessed, but in case some streams are merged into one address association, the MC.SourceSpecs argument used to describe it is flexible enough to incorporate the MC.QoSInfos from several streams (by setting the sequence length to the number of streams). If the service has sufficient knowledge about the MCT (according to its configuration), it may be able to mix streams, and combine their MC.QoSInfos into one, and an MC.SourceSpecs can represent this too (by containing only one MC.SourceSpec with more than one MC.Agent).

Source Participation Primitives

A service user wishing to participate in the channel configured for an instantiation of the MC-source service indicates its wish with the following primitive:

MC-source.join_req(string *mct*; MC.Agent *agt*; MC.QoSInfo *qos*)

mct should match the name of the MCT of the channel, while *agt* should point to an agent that can claim ownership of this service user, should the channel have participation policies to enforce. *qos* holds arbitrary data describing the stream that the source intends to produce. For example, it may indicate bandwidth required (perhaps per encoding layer in a video stream), or the size of a video image.

To leave the channel as a source, this simple primitive is used:

MC-source.leave_req()

Between these two calls, the source may receive information about address associations. Figure 3.11 shows a source being informed of a single association being created and then destroyed.

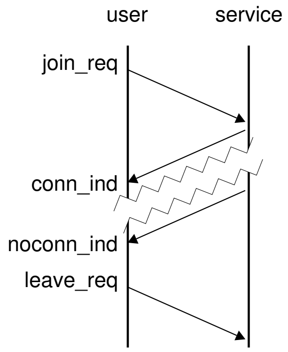


Figure 3.11: Joining, leaving and address association primitives

There is no explicit confirmation as to whether the source has been permitted to join, or if the C-AGI of the stream and the MC-ATI of the channel have been met, but the source will receive no associations (or existing ones will be removed) if they have not. A source may change its stream's description between the join and leave with:

MC-source.setqos_req(MC.QoSInfo *qos*)

An address association identified by *ref* is announced with:

MC-source.conn_ind(MC.Ref *ref*)

and cancelled with:

MC-source.noconn_ind(MC.Ref *ref*)

When cancelled, the source should release any resources that it allocated to the association between its announcement and cancellation.

This primitive indicates that the source should send its stream to *addr* for the association *ref*:

MC-source.useaddr_ind(MC.Ref *ref*; MC.UserAddr *addr*)

while this primitive indicates that an address is required from the source:

MC-source.getaddr_ind(MC.Ref *ref*)

The source should supply an address in a response like this:

MC-source.getaddr_rsp(MC.Ref *ref*; MC.UserAddr *addr*)

It may supply alternative addresses at any time by using the same primitive multiple times for the same association. The most recent useaddr_ind or getaddr_ind primitive overrides the others.

The default state for an address association in the source is equivalent to receiving a useaddr_ind with a null, zero-length address.

Sink Participation Primitives

A service user wishing to participate in the channel configured for an instantiation of the MC-sink service indicates its wish with the following primitive:

MC-sink.join_req(string *mct*; MC.Agent *agt*)

mct should match the name of the MCT of the channel, while *agt* should point to an agent that can claim ownership of this service user, should the channel have participation policies to enforce.

To leave the channel as a sink, this primitive is used:

MC-sink.leave_req()

Between these two calls, the sink may receive information about address associations. Figure 3.11 shows an example sink-service interaction with a single association being created then destroyed. There is no explicit confirmation as to whether the sink has been permitted to join, but the sink will receive no associations (or existing ones will be removed) if it has not.

An address association identified by *ref* is announced with:

MC-sink.conn_ind(MC.Ref *ref*; MC.SourceSpecs *qos*)

and cancelled with:

MC-sink.noconn_ind(MC.Ref *ref*)

When cancelled, the sink should release any resources that it allocated to the association between its announcement and cancellation.

The sink can use *qos* to determine whether it should accept the stream(s), and may indicate its choice to the service (rejection is assumed) at any time, any number of times, using this:

MC-sink.setqos_rsp(MC.Ref *ref*; MC.Agents *okay*)

okay identified the sources whose streams have been accepted. The stream description may change at any time, and the sink is informed through:

MC-sink.setqos_ind(MC.Ref *ref*; MC.SourceSpecs *qos*)

where *qos* replaces the initial value passed through conn_ind, or intervening setqos_inds.

This primitive indicates that the sink should use the address *addr* to obtain the stream(s) for the association ref:

MC-sink.useaddr_ind(MC.Ref *ref*; MC.UserAddr *addr*)

while this primitive indicates that an address that the sink is listening on is required from the sink:

MC-sink.getaddr_ind(MC.Ref *ref*)

The sink should supply an address in a response like this:

MC-sink.getaddr_rsp(MC.Ref *ref*; MC.UserAddr *addr*)

It may supply alternative addresses at any time by using the same primitive multiple times for the same association. The most recent useaddr_ind or getaddr_ind primitive overrides the others. Figure 3.12 summarises addressing interactions.

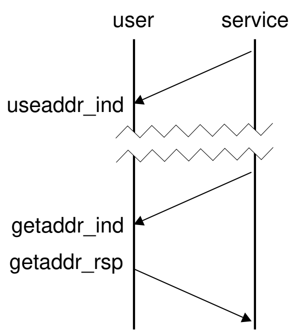


Figure 3.12: Addressing interactions

The default state for an address association in the sink is equivalent to receiving a useaddr_ind with a null, zero-length address.

MC Administration

Channel participation assumes that administration of the channel has been initiated, and is being maintained, so participation can be developed independently of administration, and many forms of administration may be defined. While no particular form is specified here, a channel administration service would allow a channel to be created, assigning a unique name to it, and supplying it with policies, and possibly maintaining them over time. Primitives should include channel creation (with name, type, and policies) and deletion, and the setting and monitoring of policies.

Session Participation

The session service allows a group of channels to be managed as a single local session. The local session service should transparently be able to interrogate a given channel to determine its MCT, and use this information to select the relevant source and sink utilities from its user-supplied configuration, and control them as the user attempts to join or leave a channel. The session may instruct these individual programs to use certain local resources, or to co-operate in some way, to support channel associations. For example, a video sink and an audio sink may be instructed to synchronise their consumers from the same source.

For group session support, a sink utility may be configured for the local session that obtains information about the group session, and modifies the state of the local session, including its channel membership/participation and channel associations.

This thesis is not closely concerned with the precise format of the service elements for local sessions, but should contain primitives to create/destroy local sessions, add/remove channels to/from the local session, and to bind channels with arbitrary associations. It should also allow participation in a channel to be controlled by the user, and so needs join and leave primitives that refer to channels within the local session. These will be subject to identified controllers of the session, where one sort of controller may be the user manipulating a graphical user agent, while another may be a group session sink changing the local session's configuration automatically. The local session will ensure that each source/sink joins depending on the priority of the controllers' indications, so the controllers will need to be given priority (the user will usually be given the highest), and should be able to indicate that their preferences may be deferred to others, in spite of their priority. This allows a user, for example, to choose explicit, manual participation in a channel, or to instruct the agent to accept the group session's specification.

Session Administration

As with channel administration, group session administration may take many forms, and is not precisely specified here. If the approach of using a channel to distribute group session information is taken, then some of the administration falls to that of channels. The remainder is a matter for establishing and maintaining databases containing group session information, and specifying it as part of the MCT for the session-controlling channel. Other approaches will simply replace the use of a channel with a different system.

3.2.1.3 Application of MC Service

The MC service has primitives to allow a service user to join the MSrc- or MSnk-intention groups of a channel independently. Assuming that it is subsequently enrolled, it will receive connection instructions through various address associations, and the nature of these is determined by the channel's configured addressing mode.

Root-initiated address associations

In this mode of multicast addressing, each sink must supply an address for each stream, and each source must send to all of the addresses supplied by each sink for the source's stream. This requires N address associations in each source to connect to N sinks, and M associations in each sink to receive from M sources/ streams.

Sink associations will use the getaddr primitives to obtain destination addresses from each sink, and each of these addresses will be supplied to a source association using the useaddr primitive. The source will recognise the addresses as for unicast, and supply them to its root-initiated multicast termination point, and will therefore be adequately connected.

Leaf-initiated multicast-address associations

In this mode, a source sends its stream to a multicast address assigned to it, while sinks requesting that stream must listen on that address. This uses one association per source, and M associations per sink, if each source has a different address.

The source association will use the getaddr primitives to obtain the multicast address, or the channel may be configured to supply an allocated address to the source using useaddr. Either way, the address is also supplied to each sink using useaddr.

In cases where all sources use the same multicast address, either they will have obtained the address through other means, and supply it using getaddr, or the channel is configured to supply them all with the same address. Each sink needs only one association, but it may receive setqos indications as sources join and leave.

Leaf-initiated root-address associations

This performs identically to l-init(m-addr), except that only getaddr will be used with the source, since only the source can supply its own address.

3.2.3 Revisited Centralised Architecture

In the first version of the MC model, a centralised manager component is used to implement the service, and makes use of individual unicast connections with the service users. This section defines the equivalent component in the present version of the MC model, in the light of the improved service primitives. It is this architecture which may be changed to support wide-area and large-scale distribution. It is desirable if the underlying manager-client protocol does not change greatly, but merely grows. It is more important that the service hides the differences between distributed and centralised architectures.

An architecture for supporting the service in section 3.2.1.2 can now be defined. A channel is managed by a network component called a *media channel manager*. It has an addressable interface through which source and sink programs can communicate over a reliable unicast service. Service primitives map very closely to protocol elements of the interface. The only difference is that a reference from the manager used to identify an address association is exchanged with a corresponding reference from the client. Manager and client will specify each other's reference in exchanged primitives.

The manager maintains a list of sources and a list of sinks, and determines where connections should exist between them. Usually all sources connect to all sinks, but in certain circumstances, this full connectivity may not be required. Three reasons for this are immediately obvious:

- A source and sink may belong to the same user agent, and for a particular channel, it may be unnecessary for a user to receive his own data.
- The user-agent may not be a permissible source/sink of the channel.
- A stream, or all streams, may not be considered worthwhile according to the channel's integrity conditions.

After establishing that a connection should exist between a source and a sink, the manager takes a series of steps to set one up, depending on the mode of multicasting to be used (different modes may be used with different connections within the same channel). When a multicast address is to be used, this sequence is followed:

1. The manager ensures that an address association exists in the source that instructs the source to transmit on the multicast address. (It may have already done this for a previously established connection.)
2. The manager creates an address association in the sink that instructs the sink to listen on the multicast address. This usually happens per source.

Figure 3.13 shows the manager interactions required when a sink joins a channel with a source already present. When a second sink joins, interaction with the source is not needed.

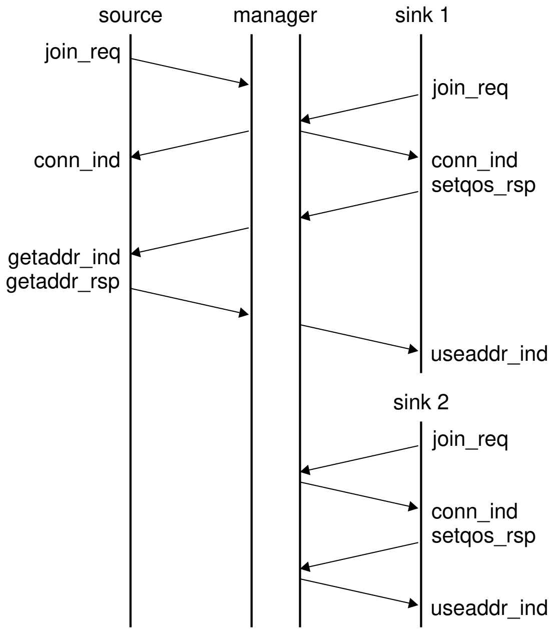


Figure 3.13: Leaf-initiated manager interaction

For root-initiated multicasting (including naïve), where the sender (in this case, the source) transmits to a list of addresses, this sequence is followed:

1. The manager creates an address association in the sink that asks the sink to supply an address that it will listen on.
2. The sink selects an address, starts listening to it, and passes it back to the manager with reference to the association.
3. The manager creates an address association in the source that instructs the source to send to the provided address.

Figure 3.14 illustrates this interaction as one sink joins a channel with one source already joined. Note that the joining of a second sink requires the same work.

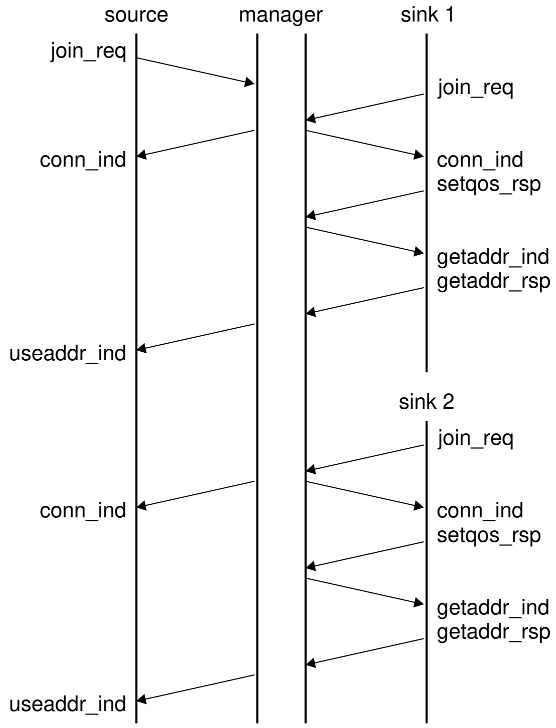


Figure 3.14: Root-initiated manager interaction

When a connection has the potential to exist, but integrity conditions prevent it, the above steps involving establishment of address associations are still made, but no addressing information is exchanged. The associations are needed since the integrity conditions depend on the sinks' acceptance of streams, and a sink can only accept a stream in the context of an association.

When a connection is to be removed, for example because a source or sink is not in the enrolled group of the channel any more, the manager simply removes the relevant address association.

When a connection is considered not worthwhile by the integrity conditions, the manager sends the useaddr_ind primitive with a null address to the relevant source and sinks, which can then close their transport resources, while the sinks can still alter their acceptance of a stream.

In this architecture, the address of the manager's interface and the channel identifier are synonymous. For a multimedia group session, as many managers will exist to support as many channels. For a group session channel, a source may exist that contains a table of the channels that make up the session, along with suggested participation information and network resource sharing details. User agents attached to this channel as sinks will maintain a copy of this table, and will automatically ensure those channels exist in its own table.

3.2.4 Examples of MC Use

To illustrate the intended use and configuration of the Media Channel model, some examples are now presented. Section 3.2.3.1 lists some basic media channel types that may be used to build up the example sessions in section 3.2.3.2.

3.2.4.1 Example MC Types

These types are broadly classed according to the media types they carry.

Video

The MPEG standard [Le Gall 91] was particularly designed for storing recorded video, and many video sequences available on the Internet are in MPEG form. With plans afoot to provide video-on-demand as a teleservice, MPEG will likely be the choice for storing large video files, and later versions make considerations for other video uses such as broadcast video.

A MCT could be defined to use MPEG compression (or the later MPEG-2 which is a hierarchical/layered encoding scheme [ISO 94]) to form the full- and Δ-state messages, while resolution requests specify the maximum MPEG-2 layer to be delivered. The MCT could appropriately be named `MPEG`.

Live video can be better suited to H.261 encoding [Liou 91] [ITU-T 93]. We could define a MCT to use H.261 frames for full- and Δ-state messages, and use the resolution requests to specify levels of quality in its hierarchy.

Audio

A suitable format for full-state transmission of audio would be packets of 8-bit μ-law-encoded, 8000 Hz samples, containing some sequencing information. The name `SunAudio` has been used previously to refer to a specific MCT based on this format [Namuye 96b].

Playback Control

Some group applications involve the playback of a recorded sequence of digitised data. For example, a video-on-demand (VoD) application may supply a control panel for rewinding, pausing and replaying a video at various speeds. For a video distribution channel, using these controls affects the display for all sinks, and is not part of the data state of that channel. In this case, we can define a MCT `PlaybackCtrl` whose data state is ‘the will of the viewers’, and make the VoD server a sink, with the users being the sources (exactly contrary to the video distribution channel).

Shared Documents

Shared documents can be implemented by a server which holds the document, and client programs which can read and display parts of the document, and allow editing commands to be returned to the server. Like the VoD system with a control channel, we can define separate reading and writing channels.

The clients are sinks of the reading channel, which is sourced by the server. The server is also the sink of the writing channel, which is sourced by the clients.

A shared whiteboard would be implemented with two MCTs: `WBDisplay` would pass Δ-state messages from the server to the clients as part of the document was edited, and clients could use resolution requests to indicate the parts of the document they were viewing; `WBEdit` would pass editing messages from the clients to the server.

On the other hand, the wb MBone application operates on a single IP multicast address. A single channel can represent this, and the manager can establish a single address association with each source and each sink, and pass the same multicast address to each association. Being a source or sink makes little difference to the communication, but may affect integrity conditions.

3.2.4.2 Example MC Sessions

Using the example MCTs in the previous section, we can demonstrate the configuration of channels to produce multimedia group sessions, assisted by channels containing session descriptions.

Audio/video ‘telephone’ call

While this is essentially a unicast connection, and is implemented with existing applications (AudioFile can be used to implement a duplex, unicast, audio interaction [Levergood 93]), the users may choose to invite a third person into the discussion, and perhaps more. So if this sort of application can be seen as a multi-user interaction (and so requires the user agent's interface) that just happens to involve only two people, allowing extra participants unexpectedly is a trivial matter.

Clearly, an audio channel and a video channel will be established, and the caller and the callee will attach a source and sink each to both. A session channel can easily set this up, as well as indicate synchronisation requirements between the channels.

The callee will need to be informed of the session channel's details, and then can choose to accept the call by joining his user agent to the session as a sink. The caller first looks up the callee's ‘phone number’ in a directory. This is a session channel of which the callee is permanently a sink. The caller joins that session as a source, and adds his newly-created call session to it. The callee views the new session appearing in his agent's display (and may be aurally alerted to it), then joins it as a sink, and the call is in progress.

Informal meeting with shared viewing and editing of documents

The participants of the meeting need to discuss some plans they are working on. Those plans are accessed through two channels of types defined for displaying and editing such plans, while an audio/video conference is supported by two more channels (`H.261` and `SunAudio`).

The conference may have grown from a ‘telephone’ call, or the participants may have agreed to set it up at a particular time. The shared document channels may be in existence permanently as work-in-progress, merely being added to the session when needed.

Major conference with sessions

This conference has been well planned out in advance. The organisers set up a session channel to represent to entire conference (this channel's identity is published to the invited participants), and consists of a number of subsessions to match the conference's own sessions, and some other administrative channels which provide general conference information (programme, contact details, *etc.*), or which alert participants to the start of various events.

Each subsession consists of an audio conference channel (to which only the current speaker and the chair have source access), and a similarly configured video channel, and for each speaker, a set of channels peculiar to his presentation, added to the subsession when needed. While members of the audience are asking questions, the chair will re-configure the enrolment groups of the conference channels to allow those members to join as sources.

3.2.5 Achievements and Limitations

The MC model has now been improved in the present version, firstly by allowing sinks to reject individual streams (and giving the sources a way of describing their streams, which sinks can use to make such decisions) so that integrity conditions can be enforced. This can also be used to supply some static stream data state to new sinks. Secondly, the service has been defined more precisely and clearly, and is independent of the architecture and protocol used to support it: we can consider other architectures with an understanding of whether they can still support the service.

However, there are still omissions of the model, such as a lack of definition of interaction between the service and a user agent that allows, in the described architecture, the manager to verify the identity of a user to enforce limitations on enrolment groups. At this stage of development, it is sufficient to know that the information required to do this is available to the service and its implementation. Another omission is the definition of interaction between local session and channel clients, and how sharing of network resources will be specified from the end-system, but these behaviours can be isolated in the end-system, and dealt with in future.

There is no way for sinks to indicate QoS guarantee failure for the arbitrary transport systems they may use, if reliability is to form part of the policies of the channel. It would be difficult for them to use stream rejection for this, as the addressing information that may be needed to determine network QoS capacity is invalidated once the channel's MC-IC or the stream's C-AGI fails.

There are also problems with the centralised approach to management. Although this allows policies to be easily evaluated, a single manager with a large number of clients may have to process a large number of interactions with them (depending on addressing mode) in a short space of time. There may also be problems in locating a central manager if the meaning of 'central' is unclear. Over time, participation in a channel might migrate so that management occurs somewhere unrelated to the current members. The delays, unreliability or cost of long-distance communication with the manager may impose unreasonable limitation on the use of channels. Also, the failure of the manager destroys the whole channel. The following chapters attempt to address the problems arising from a centralised approach.

4 Distributed Media-Channel Architectures

In the MC model described in section 3.2, a media channel is realised by the existence of a centralised media-channel manager. This manager must deal with requests, possibly simultaneously, from several clients at arbitrary positions in the network. When the number of clients is small, and they are in the same vicinity as the manager, the manager is unlikely to be overwhelmed with requests, and each one will be dealt with with a similar quality. As the number increases, the prospect of a single manager being able to deal with all requests becomes less feasible — a widely broadcast channel about to show a popular programme may be the subject of a sudden surge of activity which a single manager could not cope with (in a reasonable amount of time). Less significantly, as the clients become more varied in their distribution, those further away from the manager may be disadvantaged in dealing with it.

Clearly, an attempt must be made to distribute the functions of the manager over a wider area, allowing clients to connect transparently to their local management component in some defined region or *domain*. Management components interacts with the each other to produce the behaviour of the original single manager.

This chapter describes possible new architectures to be applied to the Media Channel (MC) model to enable it to scale to wide areas and large numbers of participants, particularly those acting as sinks. Section 4.1 considers multicast mechanisms that can be scaled (classed according to addressing mode), to be used by end-systems in the new architecture, describes a method of stream relay between management domains that may help to simplify local management. Section 4.2 considers methods for distributing the MC manager. Applicable to all strategies is that, according to the MC revision in section 3.2, the address of the manager is synonymous with the identity of the channel, and this is lost when multiple managers co-operate, so an effort must be made to allow all components of the distributed manager to be addressed as a unit at one level, while being addressed individually at a lower level (where clients communicate directly with the components), and this is also discussed in section 4.3.

4.1 Communication Scaling

If the clients of a channel are many, not only will the management need to scale well to cope with them, but the underlying multicast transport service that connects them and carries the streams must do also. Instead of considering particular multicast services and mechanisms, the following subsection looks at the scaling of addressing modes, as these determine the amount of activity in the end-points of multicast systems. Following that we consider the implications of dividing clients into domains, and connecting them up using a kind of multipeer emulation that helps to break up a multipeer connection with poor scalability into several connections with fewer end-points.

4.1.1 Scaling Multicast Modes

The multicast modes described in section 3.1.4.2 are now reconsidered for large-scale and wide-area use.

4.1.1.1 Root-initiated Multicast

When this mode, r-init, is applied by the manager to a particular stream, each sink is asked to provide an address of its own (on which it will receive data), and each of these addresses is passed on to the source. Using r-init on a large scale results in a large number of transactions between the manager and the source, and between the source and its multicasting network service, resulting in large connection overheads if many sinks join simultaneously (known as *sender-implosion* [Hofmann 96]). The source may also be using naïve multicasting causing further overheads in transmission.

However, r-init may still be useful when sources in a local domain are attached to network elements set up by the manager. These elements are repeaters which simply act as a node in a dissemination tree, but can use different multicast modes to transmit and receive. Sources in the same local domain as the sink of the repeater for that domain can use r-init to connect to it. It will then use something more appropriate for wide-area multicasting. In this case, each source will be unaware of the true number of sinks (which is a direct consequence of using a configuration which aims to reduce the number of transactions a source has to make when connecting to new sinks).

4.1.1.2 Leaf-initiated Multicast using Multicast Address

This involves the assignment of a single multicast address to the source, and its notification to each sink. For large numbers of sinks, this scales well. However, [Hofmann 96] highlights problems of flow control that may be required between sender and receiver. Also, the address must be allocated globally. In MBone, the Session Directory can be used to select addresses randomly, relying on the underuse of the total addressing space [Diot 97].

4.1.1.3 Leaf-initiated Multicast using Root Address

This scales in a similar manner to l-init(m-addr) above, and has the advantage that an address need not be allocated globally (since it contains the root address which was statically allocated to it). However, because the address must include transport-level information to distinguish roots at the same host, intermediate nodes may have to store more information per connection than for a multicast address.

4.1.2 Domain Interconnection

In addition to the new multicast services in section 3.1, we can also emulate multicast distribution by configuring several MC managers co-operating to support a single channel so that media streams are passed between the domains that each manager operates over, transparently to the clients. A component between two managers' domains can be instantiated as a source of one, and a sink of the other. In the latter context, the manager instructs its sources to send to the sink (they are unaware that they are transmitting to an internal component of the multicast stream, whose responsibility is to forward data into the domain of the other manager). In the former, the source has been instructed by the manager to send to the sinks in its domain, thus connecting the sources in the other domain to the sinks in this domain, and maintaining the connectivity that is expected from clients connected to a single manager.

4.1.2.1 Repeaters

A *repeater* is an interface between two managers' domains to carry data streams from one to the other. It consists of a sink associated with one manager, and zero or more sources associated with another manager, depending on the number of physical streams that the sink is receiving. Figure 4.1 shows how a repeater sits between two managers' domains.

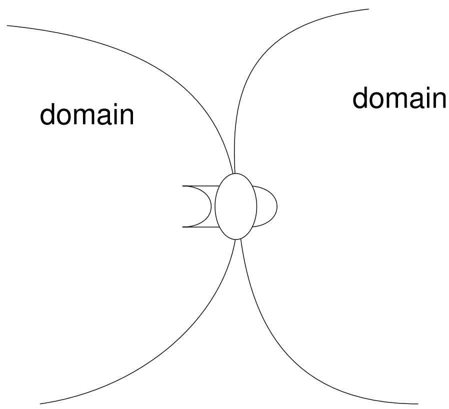


Figure 4.1: A repeater connecting two domains

A repeater receiving N streams will produce at most N streams, and may produce fewer if the carried media type supports mixing, such as audio. Figure 4.2 shows a repeater producing its maximum number of streams. The repeater may apply a filter to each stream, based on QoS requirements downstream.

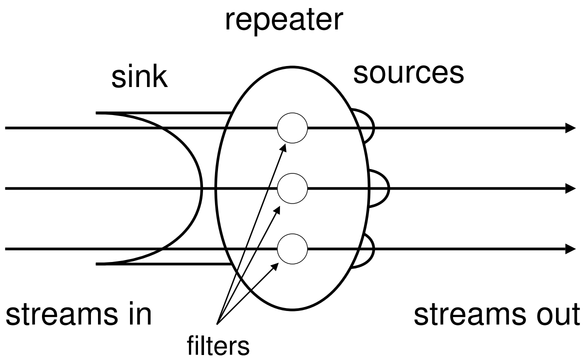


Figure 4.2: A *N*:*N* non-mixing repeater

Figure 4.3 shows a repeater that mixes its input streams into one outgoing, and may also apply a filter to it.

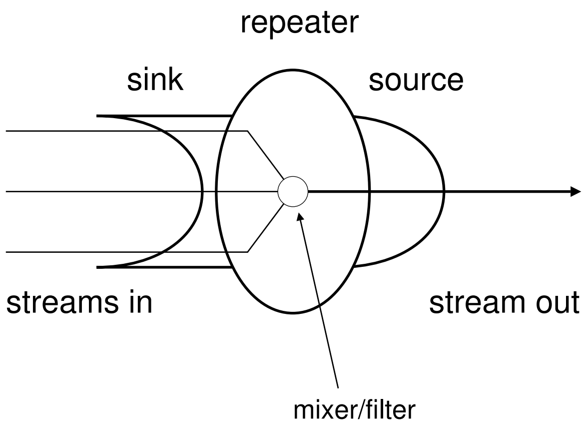


Figure 4.3: A *N*:1 mixing repeater

4.1.2.2 Positioning of Repeaters

Given a particular arrangement of domains, a sufficient number of repeaters must be established to connect them together. The model that a single manager maintains (*i.e.* a set of sources and a set of sinks) can be extended to include a set of neighbours. In general, two managers *A* and *B*, who are neighbours, can ensure their streams are connected by establishing two repeaters. One is a source of *A* and a sink of *B*, and the other is *vice versa*.

To divide the responsibility between the two managers, each one could set up only the repeaters that will be its sinks; its neighbours will set up repeaters that will be its sources, and their sinks. Alternatively, if a system is devised to establish managers in new domains, it should also be able to determine where to replace repeaters.

4.1.2.3 Management Implications

The MC model described in §3.2 describes the clients of a manager as sources or sinks. The number of each of these, as well as who is using them, may be used as arguments to policies implemented in a manager. As repeaters will be seen as ordinary sources or sinks by the managers they connect to, they may effect the evaluation of policy, and establish connections when none should exist. Therefore, when connected to a manager, they need to indicate to it that they are not to be counted within the policies, and this facility should be supported by the manager-client interfaces.

We can imagine that the MC service is implemented by a client-manager protocol whose primitives closely match those of the service. To support repeating sources and sinks, we can define additional protocol primitives that have no service equivalent, so that the service is unchanged. Ordinary sources and sinks (*i.e.* not part of repeaters) would not use these additional primitives to support their join service primitives. A repeating source could use this alternative to the join primitive (expressed as though it was a service primitive):

MC-source.joinrpt_req(string *mct*; MC.Agent *agt*; MC.SourceSpecs *qos*)

while a repeating sink would use:

MC-sink.joinrpt_req(string *mct*; MC.Agent *agt*)

Ordinary sources and sinks of a channel are still the sole MC service users, whereas repeaters are management components that use an extension of the underlying protocol. It will be seen later on that if these extensions are defined to take some extra parameters, and are used in co-ordination with other new protocol primitives, a repeater can also be used to pass some information used to evaluate integrity conditions between management domains.

4.1.2.4 Evaluation

The multicast-emulation approach allows the implementation of a distributed manager (*i.e.* several managers acting as one) to control data streams without relying on a particular, globally available multicast service. The interactions between the manager components can be tested, and protocols for those interactions can be developed.

The emulation may be inefficient in its data transfer, as it is placed above the transport layer in intermediate nodes, whereas real multicast mechanisms are intended as network- or transport-layer services, but this does present an opportunity to place MCT-specific stream processing (such as filtering and mixing as in [Yeadon 96b]) apparently inside the stream's connection. Later, full use of multicast services may be made.

The functionality required to establish managers in the right places, and to keep them informed of their immediate neighbours, can be re-used to position the data repeaters. The model is viable over any network, and though it may be superseded by more efficient (real) multicast mechanisms, it may still be useful in local areas where those mechanisms do not permeate.

4.2 Management Scaling and Distribution

The section describes various strategies that may be employed to distribute the functions of the media channel manager. Firstly, a study of the effects of particular styles of management strategies on data transfer is given.

4.2.1 Management Interaction with Communication

While group management is notionally separate from transport of multimedia, the distribution of the management function will likely correspond closely to the media distribution, as the clients of the latter are also the clients of the former. Three juxtapositions between management and multipeer transport are considered:

- *multipeer co-ordination*, where the distributed managers make the clients fully aware of each other;
- *multipeer emulation*, where managers are responsible for informing

clients to connect to intermediate nodes in a multicast connection;

- *multipeer incorporation*, where a multipeer distribution/collation tree is already established, and the managers merely allow or disallow passing data.

4.2.1.1 Multipeer Co-ordination

This is the most loose interaction between management and transport, where the managers (controlling neighbouring domains) interact with each other to ensure that all clients can directly connect to each other. This aim has an impact on the work done by the managers, depending on the multicast mode used.

One important consideration in this interaction is that a manager must maintain the same state as is required in a single-manager scenario. If the amount is high in that case, then it will also be high each co-operating manager.

Root-initiated multipeer co-ordination

In the single-manager scenario for this multicast mode, each sink must provide (through an interaction with the manager) an address for each source to send to, and this requirement still applies in the distributed-manager scenario. In order for a source in a remote domain to reach a sink in a local domain, the local manager must first be informed by the remote manager that the remote source exists, then it must ask the local sink for an address, then it must return this address to the remote manager, which forwards it to the remote source. A consequence of this is that all managers will have to maintain the same state, a table of sources *vs.* sinks, and many of the advantages of distribution are lost.

Leaf-initiated multipeer co-ordination using multicast address

In this scheme, a manager must assign a multicast address to each of its local sources, and must forward these addresses to the other managers. These addresses are then passed on to sinks which bind to them.

A manager must maintain a list of addresses, one per source (of which there are usually few), and all these addresses must be supplied to each of the manager's sink. Consequently, this uses fewer resources of state and inter-manager interaction than the root-initiated case. However, multicast addresses must be allocated from a global reservoir.

Leaf-initiated multipeer co-ordination using root address

This is similar to the multicast variation: the maintained state and inter-manager interaction are the same, but the sources choose their own addresses from their hosts' addresses.

4.2.1.2 Multipeer Emulation

Each manager controls a domain, and instantiates anonymous sources and sinks (repeaters) to allow source streams to be distributed beyond its own domain, and to receive data from other domains on behalf of the sinks in its domain. The multicast mode used has an impact on the behaviour of the managers, and how they interact.

In this scheme, the topology of the managers defines the topology of the multipeer transport. Within a domain, an efficient multicast mechanism may be used, but the inter-domain multicast is achieved by placing repeaters between domains, so it is emulated. With repeaters carrying data between domains, the manager merely has to inform its neighbouring domains' managers of any repeaters it establishes between them. This is done simply by, for example, instructing the sink part of the repeater to register with itself, and instructing the source part to register with the neighbouring manager, or *vice versa*. The neighbouring manager is informed automatically in this case, *i.e.*, without any additional protocol.

Root-initiated multipeer emulation

On the small scale of a single manager's domain, where there are few sinks with which to bother each source, so root-initiated multicast is simple and effective. Though all non-repeating sources in the channel are represented in some form in the manager, state information that it maintains on sinks is reduced, since these are coalesced into single repeating sinks, if they occur in other domains, and little interaction is needed between managers as sinks join and leave, though as with all multipeer emulation, streams are constrained to paths similar to the management topology.

Leaf-initiated multipeer emulation using multicast address

If repeaters are used to connect domains, each source within a domain must be assigned a local (*e.g.*, low TTL) multicast address by the manager, and this is relayed to all the local sinks, and therefore to those acting as repeaters, thus distributing the data to neighbours. Sources acting as repeaters are assigned addresses just as other sources are, so data will also flow in from neighbours. Figure 4.4 illustrates the use of separate multicast addresses in each domain for the same stream.

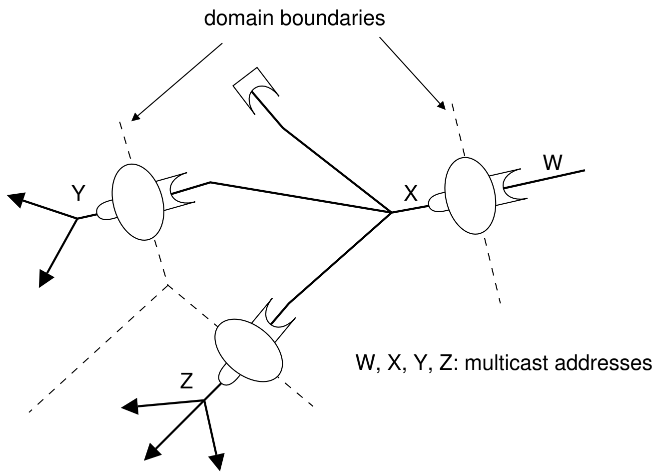


Figure 4.4: Multicast addresses per domain

Each manager holds a list of addresses, one per source, the same as with multipeer co-ordination, but the addresses of remote sources are not provided directly, but appear as local sources in the form of repeaters. Therefore, the only real advantage emulation has over co-operation in this multicast mode is that no extra protocol definition is required between the managers.

Leaf-initiated multipeer emulation using root address

The behaviour here is identical to using a multicast address, except that the address is generated by the source. The advantages of this relationship between management and transport over co-operation in this multicast mode are similar to those of the multicast address mode.

Note that we can use multipeer emulation in a different way when using a leaf-initiated multicast mode. Instead of the repeaters capturing streams from one domain and forwarding them to another, they can simply forward the address used by the source in one domain to the sinks in the other. This allows a stream's connection to build its own dissemination tree independently of the management topology (as with multipeer co-ordination, but without the extra, explicit inter-manager communication), and so the load of several streams may be distributed more effectively.

4.2.1.3 Multipeer Incorporation

In this case, management entities occur in the multicast nodes for the data communication, and it is assumed that the clients of the channel have made use of the multicast communication to join the channel, and to work out the topology of the distribution tree — a single tree is used for all communication, rather than one per source (and the service defined earlier is flexible enough to represent multiple streams arriving through a single address association). Its multicast address is used as the address of the channel. Managers need not occur at every node, and may only occur at the leaves.

Because the management is incorporated into the communication, a manager can only enable and disable communication that would pass through its node, so only leaf-initiated root-address connection instructions have any meaning to a sink (the sink would be told to receive from a number of ordinary addresses), and only leaf-initiated multicast-address connection instructions have any meaning to a source (the source is instructed to send to a single multicast address (that of the channel) or not). The purpose of the managers is then to communicate via the multicast mechanism to inform each other of the current nature of participation, so that they all make the same connection decisions based on their shared policy.

4.2.2 Management Architectures

This section lists some possible architectures of distributed management, and considers their specific relations to the data transfer that they might control.

4.2.2.1 Domain Hierarchy

In this scheme, each co-operating manager handles a local domain of its own. Another manager co-ordinates the streams from these domains into the others, as shown in figure 4.5.

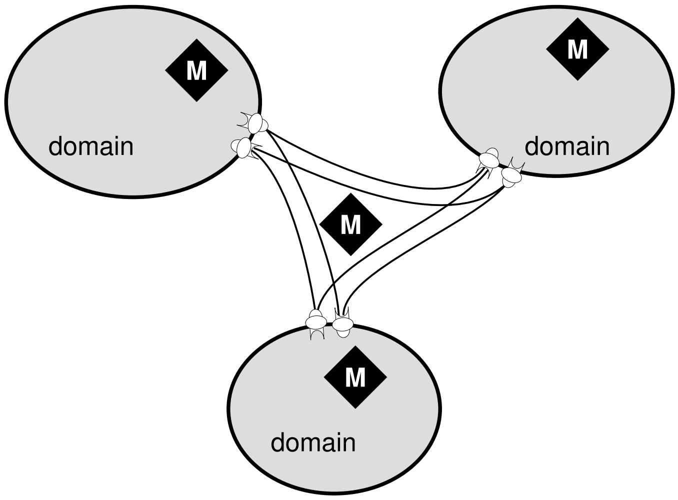


Figure 4.5: Leaf domains co-ordinated by a central manager

Each domain can operate as though implementing a multipeer emulation, with the central manager operating a domain with no real sources or sinks (only repeaters).

This architecture requires a mechanism to determine the location of a central manager for a new channel, and the central manager must be informed of new managers (or *vice versa*) in order to set up the appropriate repeaters (or other inter-manager associations). This still encourages centralised management, but responsibility for an area may be devolved to increasingly higher levels, so that the highest layers, though covering a wide area, deal with only a few repeating sinks. This can be seen as a hierarchy of domains with only the leaves having non-repeating sources and sinks.

4.2.2.2 Flat Tree of Domains

This architecture places domains along the branches of a tree. The tree has no centre or bias (so could be described as 'flat'), and each domain has any number of neighbours, and is responsible for relaying data from each of its neighbours to each of its other neighbours. When a new domain is required, it may be added as a branch from an existing domain, or inserted between two neighbouring domains if appropriate.

A similar architecture is presented in [Wilde 96/7], where managers correspond to Wilde's GMS System Agents interacting through a GMS system protocol. However, Wilde's GMS is a comprehensive group management system, dealing with whole sessions, directory services, group naming and QoS, while the MC model only deals with security and group integrity (and group naming in a 1-manager-per-channel scenario).

This architecture requires support to establish a new domain and its manager in the tree, when to establish additional domains (and to prune them), and how to find a nearby domain.

4.2.2.3 Per-Client Management

A manager exists in each client using a reliable multicast service (such as SRM [Floyd 97]) to ensure each manager holds the same management state of the group, and can implement policy for its client, within a multipeer co-ordination relationship. The channel would be identified by the multicast address, but stream communication could exist on entirely separate connections with different addressing modes and independent topologies and QoS characteristics.

4.2.3 Addressing for Multiple Managers

When first connecting to a channel, clients need the address of a manager for that channel. In the first version of the MC model, this is easily obtained, as it is synonymous with the address of the channel, which we assume the client has obtained via some other method. This synonymity does not apply in the new version in section 3.2, so a service must exist to allow a client to obtain a manager address from a channel name, and preferably, the obtained address must also point to the nearest of several managers acting on the specified channel's behalf. For the per-client architecture, this service is trivial, but with inter-connected domains and a central manager per domain, this service must also be coupled with one to establish and prune domains and their managers as necessary.

Requirements in the WWW have led to the on-going development of services that might also be applicable to the problem of addressing multiple managers. WWW sites can be accessed across the world, and consequently, the quality of access may vary with greater distances. A single site in the WWW may also be overloaded by the volume of traffic, caused by users from the whole world, that connects to it.

To alleviate these problems, site managers establish *mirror sites* that are remote from the original, but close to some group of users who have difficulty accessing the original site. Each mirror site is a copy of the original site (reliably transmitted), so visitors to the original site are offered links to the remote site, which the user determines whether it is nearer. Users are drawn to the nearest mirror sites, reducing the load on any one site, and improving access quality.

This still has a drawback in that initially, a web surfer must still visit the main site once to find a local mirror, or if someone wants to link to the website, he must provide links to at least some of the mirrors even though they represent the same resource. Uniform Resource Names (URNs) are proposed to deal with this: a resource on the web is registered under a URN by supplying a URN resolution service with addresses of all mirrors. The URN is location-independent, so even the original site may be treated as another mirror, and may be abandoned without affecting the URN. The URN is used in place of the many URLs [Berners-Lee 94] that describe each mirror, so a reference to the resource is just a single link to the URN. The URN resolution service converts a URN to one or more URLs, from which a client may select a nearby mirror, or the service may be able to perform that itself. The URN also permits a resource (even if it has no mirrors) to migrate over time easily, since links to it do not have to change.

The requirement to make a WWW resource identifier location-independent is similar to the requirement to instruct a new channel client to use a local management resource, except that channel resources may be highly dynamic, whereas website mirrors are established more steadily and remain for long periods in order to be useful. If a URL can be used to express a manager address, then a URN can represent a channel, but the resolution service must be fast enough to accept (de)registrations of managers going into and out of use.

5 Design & Implementation

This chapter considers the design of the components, and their interaction, of the two distributed architectures described in the previous chapter that are based on the use of ordinary media channel managers which have been adapted to co-operate with each other in neighbouring domains. These components consist of a co-operating manager, and repeaters. In each of these architectures, domains are connected to each other in the form of a tree, and so the components will work equally well in each. However, other components are needed to provide a service that will maintain the tree, creating new branches to reach new participants, and pruning them when they become redundant. These components will identify a tree, and therefore a channel, by a name.

One of these architectures (in section 4.2.2.2) involves giving each domain (each node of the tree) an equal significance, and consequently, the tree has no root (it could be said to be flat). Such a tree needs a sophisticated service to maintain it, *i.e.* to determine that a new client is in a domain with no manager, to create a new manager, and to locate an appropriate neighbouring manager and inform them that they are neighbours.

The other architecture (in section 4.2.2.1) gives a definite hierarchy to the tree, with a common root. Its tree-maintenance service can be simpler, as to connect a new manager set up in a domain at one level simply requires that the address of the manager for that channel in the next level up is obtained (possibly creating the upper manager in the same, recursive process), and that these two managers are made neighbours of each other.

This chapter first deals with the design of the common components: the manager, the repeater, and the manager-client protocol that is capable of dealing with both repeaters and normal clients. Then it designs a tree-maintenance system for the hierarchical architecture, and outlines designs for the flat architecture.

Finally, implementation details of the common components, the flat tree-maintenance utilities, and some example source and sink programs are given.

5.1 Common Components

Designing a co-operating manager involves passing of information regarding:

- policies (enrolment groups, integrity conditions),
- connections (addressing of stream end-points), and
- integrity evaluation (to decide whether integrity is achieved).

Enrolment can be dealt with by a local manager.

If we assume that the policies are largely static, they can easily be held by some group management database which a new manager interrogates on initialisation, so that is not addressed in detail here.

5.1.1 Distributing Connection Information

In multipeer co-ordination, the protocol elements for getaddr and useaddr must somehow be carried between the relevant domains. For example, with a root-initiated stream, a manager with a new local sink must know how many sources there are (even though these may be other domains), and how their streams are described, so that it can determine the correct number of associations to make with that sink, and what descriptions to pass to each association. Having obtained addresses for each of them, it must then forward each of these to the appropriate source, and this looses much of the advantage of manager distribution, as source descriptions and sink addresses are replicated in each manager (see figure 5.1). Leaf-initiated schemes are simpler: an address from each source is obtained only once, then sinks can be connected locally by supplying the memorised address.

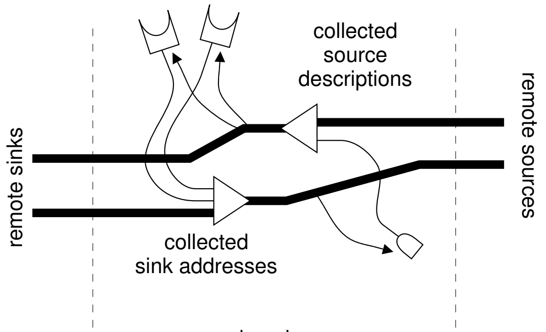


Figure 5.1: Root-initiated co-ordination problems

In multipeer emulation, each non-repeating source in the whole channel will have some form of representation (in fact, a repeating source) in each manager (since a repeating sink that it connects to will create a representative source in the next domain), and all non-repeating sink in a domain will be coalesced into a single repeating sink in each of the neighbouring domains. Managers then only have to connect up their local sources and sinks (whether repeating or not), and full connectivity will be established. This form is clearly desirable, but may not outweigh the problems of constraining the streams along the same paths as the management tree. However, for leaf-initiated modes of addressing, a repeater does not have to carry any data, but merely forward a root/multicast address from the source's getaddr_rsp primitive to the sink's useaddr_ind primitive. Inter-manager communication is kept simple (the benefit of multipeer emulation), while streams are free to take any route prescribed by the application's choice of transport system. Indeed, the repeaters are now supporting multipeer co-ordination, as in figure 5.2.

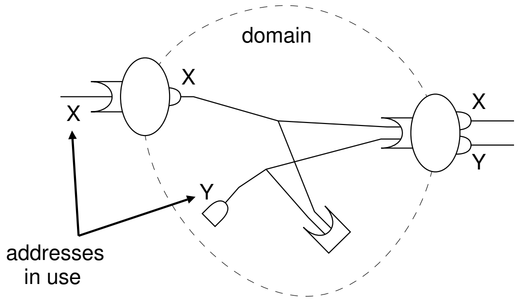


Figure 5.2: Leaf-initiated emulation supporting co-ordination

5.1.2 Distributing Integrity-Evaluation Information

In a single-manager architecture, the manager can make use of the identities of its sources to identify streams and their associated C-AGIs, and can evaluate these by observing the identities of its accepting sinks.

In multipeer co-ordination, each manager must be made explicitly aware of its neighbours' sources and sinks (and their stream acceptance), through a defined inter-manager protocol, in order to perform the same evaluation. Again, multipeer co-operation is defeating some of the distribution advantages by listing all sinks, but this information can be compressed by expressing the C-AGI for each stream as a boolean expression whose terms are the states of counters above defined thresholds.

For example, a stream may require that 3 sinks accept it before it can be used. The information required is a single counter for the number of sinks accepting the stream. A set of key members is expressed as a counter whose value is the number of accepting sinks belonging to that set, with a threshold of the same size.

A manager includes its local sinks in the values of the relevant counters, as well as values obtained for the same counter from its neighbours, and so can evaluate the integrity itself. Conversely, it must provide this value to each neighbour, minus that neighbour's supplied value, as shown in figure 5.3.

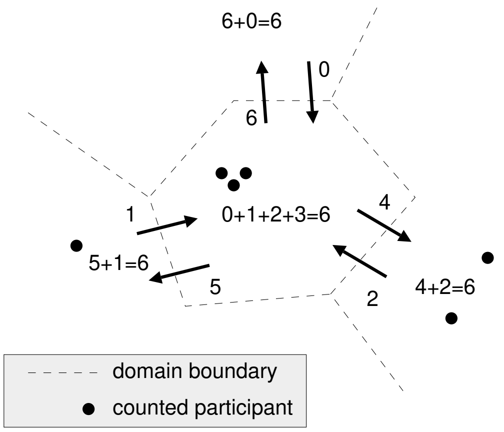


Figure 5.3: Sharing counter values

A change in the counter will propagate to all domains eventually, though this may cause problems if hard integrity is enforced (see figure 5.4): a) a sink may leave one domain (on the right), causing a counter to fall below the threshold (5 in this example) in a wave, while a sink in a remote domain (on the left) may join, causing an increasing-counter wave already above the threshold. Until these waves meet, as in b), those domains with the low counter value will be in a failure state, and the hard integrity will treat this as permanent, as seen in c), after the counter value has settled. Meanwhile, the integrity in the other domains will appear to have been maintained.

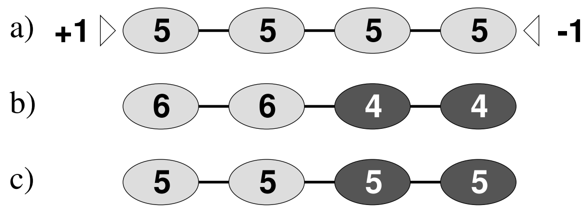


Figure 5.4: Hard integrity inconsistencies

For multipeer co-operation, an inter-manager protocol is needed to transfer counter values.

In multipeer emulation, whether repeaters carry data (as is necessary for root-initiated multicast) or not (as is appropriate for leaf-initiated multicast), each real source will have some representation in each manager. If we limit the supported integrity conditions to the ATI and C-AGIs, we can also limit the counter values to those applying to specific sources (rather than an MSnk-AGI, which applies to the whole channel). We can then extend the manager-client protocol to allow repeating sources and sinks to carry counter values per stream (for sources) or per address association (for sinks).

5.1.3 Distributed Manager-Client Protocol

In the non-distributed architecture, the protocol appears very similar to the MC participation service. However, in the distributed architecture, some clients may be repeaters, and these may be able to carry integrity information. If the new protocol is designed to allow this, it is unnecessary to design an extra inter-manager protocol, and this approach is taken for the implementation.

The interface is defined in appendix C. The co-operating manager supports the `MC.SourceManager` and `MC.SinkManager` interfaces, while a source must support the `MC.ManagerSource` interface, and a sink must support the `MC.ManagerSink` interface.

5.1.3.1 Additional Types

Some data types are used in the protocol, but are hidden in the service.

- `MC.Manager` - a manager address
- `MC.CounterVal` - a counter value (a signed integer)
- `MC.CounterVals` - a sequence of `MC.CounterVal`s
- `MC.SourceScore` - counter values associated with an agent-identified (`MC.Agent`) source
- `MC.SourceScores` - a sequence of `MC.SourceScore`s

5.1.3.2 Manager-Source Primitives

A source joins a manager by sending it a `join_req` (form normal sources) or `joinrpt_req` (for repeating sources), specifying `mct` for type-checking, and its agent `agt` for identification/authorization.

```
// in MC.SourceManager
join_req join_ind(string mct; Agent agt; QoSInfo qos);
joinrpt_req joinrpt_ind(string mct; Agent agt; SourceSpecs qos; Manager
mgr);
```

A repeater must also specify the address mgr of the manager controlling the hidden domain (the manager then knows not to loop back its streams to a sink that will carry them into the originating domain). All sources must describe the stream they carry with the qos argument. For normal sources, this is an opaque structure to the manager (and agt identifies the source), but a repeater can specify a mixed stream (with several sources sharing the same opaque description) or several streams using the same transport (a set of source-description pairs) using its structured MC.SourceSpecs argument, and agt is not used. The manager can easily form a MC.SourceSpecs from the agt and qos of a normal source.

A source can update its qos value at any time by issuing one of the setqos_req primitives, depending on whether it is a repeater:

```
// in MC.SourceManager
setqos_req setqos_ind(QoSInfo qos);
setqos_req setqos_ind(SourceSpecs qos);
```

Repeaters have a duty to carry counter information between managers. A repeating source can indicate to the manager the counter values for some sources through a setscore_req primitive:

```
// in MC.SourceManager
setscore_req setscore_ind(SourceScores scores);
```

The manager can account for further values on its own side:

```
// in MC.ManagerSource
setscore_rsp setscore_cfm(SourceScores scores);
```

Although these are request/response messages, they can be transmitted in any order, as many times as necessary. Not all counters have to specified in each call — only those that have changed are necessary.

The manager will establish a number of address associations with each and either kind of enrolled source. It issues a conn_req to indicate that a new association exists, and the manager refers to it as mgr. The source should respond with conn_rsp, indicating that it refers to the association as usr. Subsequent calls from manager to source will refer to the association by usr, while opposite calls will use mgr.

```
// in MC.ManagerSource
```

```
conn_req conn_ind(Ref mgr);
// in MC.SourceManager
conn_rsp conn_cfm(Ref mgr, usr);
```

For each association, the manager will instruct the source to use an address addr using `useaddr_req`, or to supply an address addr using `getaddr_req` — the source responds with `getaddr_rsp`. The manager may close the association with `noconn_req`. Apart from the distinction between manager and client references, these primitives on the client side correspond directly to the service primitives:

```
// in MC.ManagerSource
getaddr_req getaddr_ind(Ref usr);
useaddr_req useaddr_ind(Ref usr; UserAddr addr);
setqos_rsp setqos_cfm(Ref usr; Agents okay);
noconn_req noconn_ind(Ref usr);
// in MC.SourceManager
getaddr_rsp getaddr_cfm(Ref mgr; UserAddr addr);
```

The `setqos_cfm` allows a repeater to determine which sources in its sink's domain it should accept with `setqos_rsp` (see the sink primitives below). The information is received by the source part of the repeater (in the other domain), as passed internally to the sink part.

Finally, a source leaves control by the manager by issuing `leave_req`:

```
// in MC.SourceManager
leave_req leave_ind();
```

5.1.3.3 Manager-Sink Primitives

A sink joins a manager by sending it a `join_req` (form normal sinks) or `joinrpt_req` (for repeating sinks), specifying `mct` for type-checking, and its agent `agt` for identification/authorization.

```
// in MC.SinkManager
join_req join_ind(string mct; Agent agt; QoSInfo qos);
joinrpt_req joinrpt_ind(string mct; Agent agt; Manager mgr);
```

A repeater must also specify the address `mgr` of the manager controlling the hidden domain (the manager then knows not to loop back streams from a source that connects to the same domain).

The manager will establish a number of address associations with each and either kind of of enrolled sink. It issues a `conn_req` to indicate that a new association exists, accepting it depends on the description `qos`, and the manager refers to it as `mgr`. The sink should respond with `conn_rsp`, indicating that it refers to the association as `usr`. Subsequent calls from manager to sink will refer to the association by `usr`, while opposite calls will use `mgr`.

```
// in MC.ManagerSink
conn_req conn_ind(Ref mgr; SourceSpecs qos);
// in MC.SinkManager
conn_rsp conn_cfm(Ref mgr, usr);
```

The manager may indicate a new value for `qos` by sending a `setqos_req`.

```
// in MC.ManagerSink
setqos_req setqos_ind(Ref usr; SourceSpecs qos);
```

A sink can indicate its acceptance of individual streams by listing their sources `okay` with `setqos_rsp`. Other sources are rejected. The sink may update its acceptance for a particular association at any time this way.

```
// in MC.SinkManager
setqos_rsp setqos_cfm(Ref mgr; Agents okay);
```

Repeaters have a duty to carry counter information between managers. This is distributed among the associations according to the sources represented by each one. A manager indicates the counter values for the sources on one association through a `setscore_req` primitive:

```
// in MC.ManagerSink
setscore_req setscore_ind(Ref usr; SourceScores scores);
```

A repeating sink accounts for the values on its hidden side using `setscore_rsp`:

```
// in MC.SinkManager
setscore_rsp setscore_cfm(Ref usr; SourceScores scores);
```

Although these are request/response messages, they can be transmitted in any order, as many times as necessary. Not all counters have to specified in each call — only those that have changed are necessary.

For each address association in the sink, the manager will instruct the sink to use an address `addr` using `useaddr_req`, or to supply an address `addr` using `getaddr_req` — the sink responds with `getaddr_rsp`. The manager may close the association with `noconn_req`. Apart from the distinction between manager and client references, these primitives on the client side correspond directly to the service primitives.

```
// in MC.ManagerSink
getaddr_req getaddr_ind(Ref usr);
useaddr_req useaddr_ind(Ref usr; UserAddr addr);
noconn_req noconn_ind(Ref usr);
// in MC.SinkManager
getaddr_rsp getaddr_cfm(Ref mgr; UserAddr addr);
```

Finally, a sink leaves control by the manager by issuing `leave_req`:

```
// in MC.SinkManager
leave_req leave_ind();
```

5.2 Tree Maintenance

To use the manager-client protocol, a new client must obtain the address of a manager, and this should be a nearby one, *i.e.* in the same domain. A service must exist to obtain this address, given a channel name, and if necessary, a new manager should be created, and connected to an existing manager supporting the channel.

5.2.1 Hierarchical Tree Maintenance

A simple way to achieve this is to have a hierarchy of factories, one factory per domain. The factory has an interface used by sources and sinks in its domain, and also by factories in child domains. Figure 5.5 below shows a portion of a hierarchy of factories.

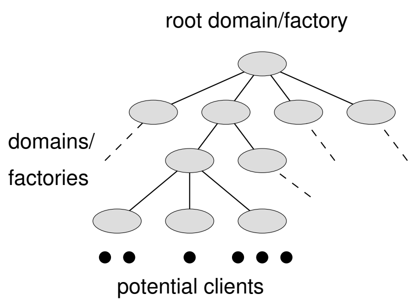


Figure 5.5: Factory/domain hierarchy

The factory interface allows a client to request a manager address for a particular channel. The factory looks up the channel in a database of existing managers, and returns the appropriate manager's address. If it cannot be found, the factory starts a new manager, giving it configuration details, enters the manager in its database, and returns the new address. After creating new manager, it must now attach it to the existing manager tree. Unless it is the top-level domain, it must ask its parent factory to supply the address of the manager for the channel in the parent domain. Given this address and that of its own manager, the factory can now initiate a pair of repeaters to connect the parent and child managers together. Additionally, the parent factory will create the parent manager if necessary, and ensure it is connected to its parent (if there is a parent domain). Figure 5.6 a) shows an unpopulated tree of factories, with new managers and repeaters created for the new participant. b) shows how two new clients require fewer additional internal components.

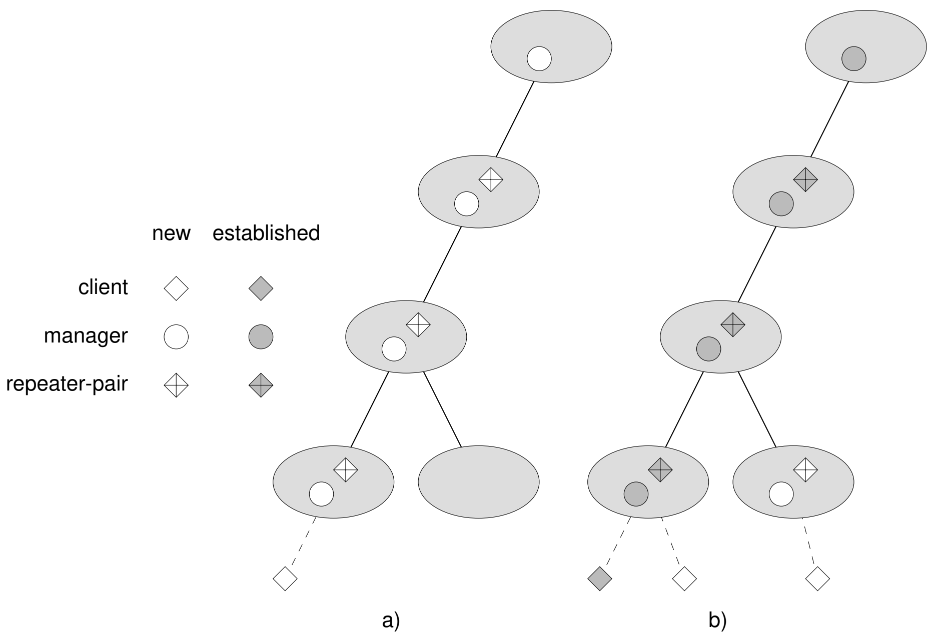


Figure 5.6: Adding new clients

To prune unused managers, a factory must establish that its manager has no real sources or sinks, and at most one repeating source or sink. If so, it can destroy the repeaters and the manager. Leaf managers will have to be removed first, as other managers will have 2 or more repeating sources and sinks (one to the parent, and at least one to a child).

A more sophisticated service to maintain a hierarchy of managers may detect strings of managers that have no real sources or sinks, and are only used to connect other pairs of managers together. Having done so, it may remove the intervening managers, and connect the terminating managers directly. This may be done earlier, precluding the existence of those managers in the first place.

This approach leads to some other difficulties: a deleted manager may later be needed for a new client. The service must be able to insert the manager between the existing managers, and simultaneously detach the existing pair of repeaters while creating new ones to connect the new manager, so as not to disturb the existing managers' integrity states.

5.2.2 Flat Tree Maintenance

The service to support a flat tree of managers can have a similar interface to that for hierarchical managers: a new client needs to ensure that a manager for a specified channel exists in the local domain, and needs its address, while the service will handle the manager's connection to a nearby neighbour transparently.

This could be achieved firstly by having a factory/database mapping channel names to managers in the local domain (as with hierarchical trees), which the new client interrogates to obtain a local manager address. Using a sufficiently dynamic URN-resolving service, each new manager can register its address with the name of its channel as a URN, and will deregister when it terminates. After creating a manager, the factory can interrogate the URN-resolving service to obtain the address of another, nearby manager, and then make the managers neighbours of each other. This last step must involve careful management of repeaters, since it is not as clear as in the hierarchical model who has responsibility for removing them when they become redundant.

As with hierarchical trees, additional sophistication in the tree-maintenance service would allow new managers to be inserted between existing neighbours, and to be removed, without transient errors in integrity evaluation.

5.3 Implementation

Implemented components of the hierarchical tree Media Channel model are written in C++ for Unix platforms, and use a simple distributed processing environment (DPE) that supports distributed objects with multiple, named, simplex interfaces consisting of methods with arbitrary typed arguments (see appendix A). The DPE integrates easily with other systems that use the File Event abstraction in appendix B, which allows separately developed components of a Unix process to express the conditions of their activation as synchronous events (alarms) and asynchronous events (file descriptors becoming ready through Unix `select()`) without multithreading.

Managers and clients communicate using the following interface definition in appendix C.

Details of the implementation of a co-operating manager, and repeaters, sources and sinks for audio conferences and performance measurements now follow.

5.3.1 Co-operating Manager

The manager is coded to accept normal source and sink clients, and also to recognise repeating sources and sinks. Only universal enrolment groups are supported. No integrity conditions are supported (and repeating clients are not supplied with counter values, and returned values will be ignored), but rejections of individual streams by individual sinks are recognised, and only sinks that accept a stream will be instructed to connect to it. Only root-initiated streams are established between clients. A repeating source and sink with the same *mgr* argument in their joinrpt_req will not be connected.

A mesh of `MCInteraction`s is maintained, one per source per sink, and each `MCInteraction` is an element of two doubly linked lists (the source client is the head of one, and the sink client is the head of the other), and holds an internal MC.Ref which clients use to identify it, along with the MC.Refs returned by each of the source and sink in their conn_rsp messages. Also held is the list of source agents that the sink has accepted. If any of these match the source agent(s) in the stream description, the source and sink will be connected.

To run the manager a MCT type, **mct**, must be specified. The manager will bind to an address **addr** if specified:

```
% mcmanager -type mct [-bind addr]
```

5.3.2 Audio Utilities

Audio utilities use the media channel type `SunAudio`, and consist of:

- a source that packetises over UDP an 8-bit/sample, 8 KHz audio stream, either from a file in a continuous cycle, or from local hardware;
- a sink that plays such a stream through local hardware;
- a repeater that mixes its incoming streams into one, and delivers this through its source.

Interfacing with audio hardware is through AudioFile [Levergood 93] which multiplexes attachment to the `/dev/audio` device in the style of the X Window System, so multiple audio clients can operate independently. The repeater uses a 0·5 KB table to convert the μ-law, logarithmic input samples into signed linear values before mixing. Mixed samples are then converted back to logarithmic values using a 64 KB table. The tables were compiled from a C source file generated using the code adapted from:

`file://evans.ee.adfa.oz.au/pub/dsp/misc/ulaw_reese`

The sinks accept all sources specified with the stream description supplied with each address association immediately.

The source and sink are run with these commands:

```
% ausource addr
% ausink addr
```

where **addr** is the address of a manager. The repeater takes two addresses:

```
% aurepeater -src srcaddr -snk snkaddr
```

srcaddr is the address of a manager that sources within the repeater will join. **snkaddr** is the address of a manager that the sink within the repeater will join.

These programs must be killed to end participation.

5.3.3 Performance Utilities

These consist of a source, sink and repeater using the MCT `Measure`. The source transmits a small packet over UDP as soon as it can send through an association, sinks accept associations immediately, and the repeater source will send to each association as soon as its sink association has received a packet.

The source displays the time that it sends its join request, while the sink displays the time that it returns each of its getaddr_rsps, and the time that it receives the first packet.

The source and sink are run with these commands:

```
% testsource addr
% testsink addr
```

where **addr** is the address of a manager. The repeater takes two addresses:

```
% testrepeater -src srcaddr -snk snkaddr
```

srcaddr is the address of a manager that sources within the repeater will join. **snkaddr** is the address of a manager that the sink within the repeater will join.

These programs must be killed to end participation.

5.3.4 Tree Management Utilities

A factory to support maintenance of a hierarchy of domains has been written, together with a client program. An interface definition file in appendix C is used for their communication.

The factory supports a `FactoryServer` interface, and the client supports a `FactoryClient` interface. The client connects to the factory, and issues a `getmgr_req(name)`. The factory receives a `getmgr_ind(name)` through its interface, and looks up the name in its database of channels (a binary tree with the channel name as the key field).

Each entry in the database holds a manager address, and two pipes to a pair of repeaters. If no entry can be found, it is created, and a pipe to a manager process is opened. The manager will first print its address so that the factory can write the address into the entry.

Whether the manager needed to be created or not, the factory can now return the manager's address `addr` to the client through a `getmgr_rsp(name, addr)`. The client receives a `getmgr_ind(name, addr)`, displays the address, and terminates.

After returning the manager address, the factory checks to see if it has a configured parent factory, and if it has started any repeaters. If there is a parent but no repeaters, it contacts the parent and sends it `getmgr_req(name)`. When the parent responds with the address of a manager in the parent domain, the factory opens the pipes to two new repeaters, with reversed **`-src`** and **`-snk`** arguments.

A factory is run with this command:

```
% mcfactorys [-parent par] [-bind poid] -rpt rptcmd -mgr mgrcmd
```

`par` is specified if the factory has a parent at that address. **`poid`** specified a local address that the factory should use (by default, it will choose one itself). **`mgrcmd`** and **`rptcmd`** are the commands to be passed to `popen` to start manager and pipe processes. The factory's address is printed immediately.

The client is run with the following command:

```
% mcfactory [-server addr] chname
```

`addr` is the address of the factory to contact, but by default this is taken from the environment variable `MCFACTORY`. **`chname`** is the channel name.

The resultant manager address is printed as soon as it is obtained, then the program stops. This is intended to be used in conjunction with a shell's ability to use the output of one command as the argument of another, as follows, for example:

```
% testsink `mcfactory MyChannel`
```

6 Evaluation of the Distributed Media Channel Model

This chapter discusses some of the achievements and limitations of the distributed Media Channel model, its possible architectures, and its implementation. In particular, the performance of the distributed implementation is compared with a more centralised implementation to demonstrate its advantages. The components of the new model are also compared with those of the first architecture described in [Namuye 96b].

6.1 MC Service Evaluation

The MC service is improved over earlier versions by defining the primitives required to allow sinks to indicate their acceptance or rejection of individual sources, allowing the integrity conditions of a channel's policies to be evaluated, indicating to sources and sinks whether multipeer transport connections between them are worthwhile. Acceptance is dependent on the identity of a source, and an arbitrary application-defined stream description, which were absent from the old model (which assumed all sinks would try to join to all sources).

The service also leaves the addressing format and the media transport system that will connect sources and sinks directly as application-defined, making the service adaptable to many media channel types and applications. A channel can be configured to use various addressing modes, so that sources and sinks are presented with a common interface for interaction, which then allows channels to be controlled in a uniform manner. A general M:N topology is supported (so 1:N and 1:1 are trivial simplifications), and may involve a real or virtual multipeer interaction.

Layered multicasting to support heterogeneity across receivers in a group communication is achieveable since the format of connection addresses handled by the MC service is application-defined, so an address as seen by the MC service may actually be a set of addresses supplying the different levels of quality of a stream.

Stream descriptions can reflect otherwise transparent mixing and coalescing of streams from several sources, if the implementation or the addressing mode causes it. Application-defined behaviour and data formats are summarised by the service as a MCT, so the service can do elementary type-checking of its users.

The service is designed to be suitable for several implementations and architectures, such as control by a management object in the network, or the transmission of information across clients bound to a single multicast address, which then use local facilities for evaluating policies.

An omission from the MC service is monitoring of reliability. A media channel has the potential to monitor active multipeer connections for reports of reliability failure or success from individual clients, and to determine whether policy-defined reliability conditions have been met. The channel can inform participants of an unreliable multipeer connection to take some appropriate, client-defined action.

6.2 Session Services Evaluation

User agents can be used to activate and terminate channel clients on a user's end-system by providing a user interface to a local sesion. After interrogating a specified channel for its MCT, the local session is able to select an appropriate client for participation, according to the agent's configuration. Local sessions are the unit of management for user agents, and are collections of channels with participation information. A local session may have channels added to or removed from it, and a user agent that holds it will start or stop the relevant clients for those channels.

To produce the effect of co-incident participation in a group of channels acting as a session, a controlled group session (that is, a description of several channel associations) may be established, and then participants of the session use this description to configure their own local sessions, and to specify other channel associations (such as resource sharing or synchronisation).

The mechanism to obtain group session control information and to hand it to a user agent is not specified or limited to any particular form. This allows for the use of a method involving placing the group session information on a channel of its own (with a specific session MCT), and configuring user agents/local sessions to use themselves as sinks of this channel type. This method firstly allows a change to the controlled session to be made available to all participants, and secondly allows a session to be treated as a channel, and therefore placed inside other sessions. A user is then able to join a session (and therefore several channels as a unit) by simply participating as a sink of the session channel. There is no requirement for a channel to belong to exactly one group session, so two independently administrated sessions may specify a common shared resource.

Since a controlling mechanism isn't specified, other mechanisms to supply controlled sessions to user agents (or ones that enhance or supersede the method described above) are also possible.

6.3 MC Architectures Evaluation

The MC model specifies possible architectures to implement its service based on the use of one or more managers. Each manager is responsible for clients in its own domain, allowing management to occur locally if global interaction is unnecessary. The use of repeaters to connect domains by passing connection information between managers scales well to large numbers of sinks, since these are aggregated when they are represented by repeaters in domains other than their own.

Repeaters may carry the MCT-defined streams between domains, in which case they must be written for specific MCTs, and may introduce additional end-to-end delays on the transmission of streams, but also provide locations for MCT-specific stream processing such as filtering and mixing, and can perform multicasting. Indeed, if root-initiated addressing is used, this multicasting is preferable, since the repeating sink must aggregate addresses obtained from the address associations of its repeating source(s), in order to pass an address through its address association for each source (this is simpler if it supplies an address of its own, rather than sending the whole set of addresses to the source, since the source requires fewer interactions, and its addressing information is of fixed size). Different addressing modes may be used in either domain of a repeater.

Repeaters always pass C-AGI information between managers, and this scalable to large numbers of sinks, since only summaries of the sinks relevant to the policies are passed around, though any change in these summaries is propagated to all managers. Transient inconsistencies may arise as two propagations from opposite ends of the manager network cancel each other out while a counter value is near its threshold. Information for a general sink AGI cannot be passed through a repeater because its component sink and sources is not necessarily a 1:1 relationship, and so the AGI cannot be evaluated.

Large numbers of sources will not scale well, since each enrolled source will have some distinct representation within each domain. The protocol supporting the distributed manager architecture allows a repeating source to aggregate its represented sources, and specify an appropriate source description. However, many kinds of group application will not involve large numbers of sources.

Manager domains can be layed out in a number of ways, but always as a tree (*i.e.* having exactly one route between each pair of nodes). Domains may have increasing range, *e.g.* department, organisation, country *etc.*, where only the leaves have significant numbers of non-repeating sources and sinks, and parent domains merely connect siblings together. In the top-level domain, a manager will be forced into a particular location, even though its clients are widely distributed. This form has an advantage that it is simple to establish.

The other form involves domains of even size that form a tree with no distinct root. While the membership is evenly distributed across such domains, establishment of a branch to a new domain involves locating a nearby domain, and creating a connection to it. This requires more sophisticated services than those required by the hierarchical layout. Also, depending on how a channel's population migrates, adjacent domains may be connected through arbitrarily long routes between repeaters, that may eventually only have historical significance.

6.4 MC Implementation Evaluation

The implementation in this thesis consists of a manager that co-operate with other managers through repeaters. A factory is implemented to supply the address of a manager for a channel identified by a name, starting one if necessary. It can also contact a second factory to provide the address of a co-operating manager at a higher level, and then run a repeater in each direction between the parent manager and its own. No provision is made to tear down managers as they become redundant. The manager only supports root-initiated addressing, and implements no policies.

A `SunAudio` source, sink and repeater are implemented, and since the manager only deals with root-initiated addressing, the repeater mixes its streams (transferred via UDP) into one. Sinks do not allow the user to specify the acceptance of a source or sink.

A source, sink and repeater with no heavily loaded MCT characteristics (null-MCT) are also defined for timing purposes. A source displays the current time immediately before connecting to the manager and joining, while the sink displays the time it is asked to supply an address, and the time it receives its first data packet. The source sends a packet as soon as it receives a UDP address from the manager. The repeater similarly sends a first packet, so long as its sink has already received the first packet from the represented source.

All these clients must be supplied with a manager address (two for repeaters), and this can be obtained from a factory client which contacts the configured local factory to exchange a global name for a local address.

These programs allow an audio conference with no restrictions or integrity conditions to be established between participants. Management may be distributed, and automatically configured, while the conference may be identified by a global name.

The null-MCT clients allow timing experiments to be carried out between sources and sinks without concern for the varying overheads of real MCTs. Since sources and sinks may be on different machines, it is also necessary to determine the difference between the two clocks for absolute timing to be determined.

6.5 Distributed versus Centralised Performance

Since the aim of distributing the manager functions is to balance the load when their are many clients, timing runs have been made to establish the time from connecting a new source to connecting all of the current sinks in a null-MCT channel. This is interpreted as two times: the time each sink gets a getaddr_ind from the channel (since it used root-initiated addressing), and the time for an initial packet to arrive at the sink (sources and repeaters send a packet immediately to each address association as soon as it receives a useaddr_ind). This times are both measured from the source's initial attempt to join the channel, and a clock correction is applied to times measured at different hosts. Measured values are averaged over 20 consecutive readings.

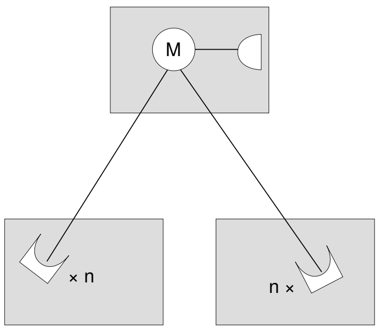


Figure 6.1: Processes for first run (A1)

The results from a channel using only one manager are compared with those from a channel with several managers. In each run, a single source on one host joins a channel with 2-26 (even numbers) pre-established sinks evenly placed on two other similar hosts. In the first run, referred to as A1, a single manager exists on the source host, and shown in figure 6.1.

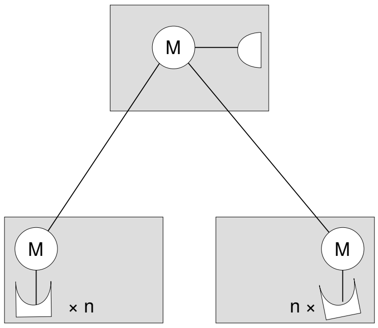


Figure 6.2: Processes for second run (A2)

In the second run (A2), the same manager exists, but only the source uses it directly. Two more managers, one on each of the other hosts, are used by the sinks on that host. This should allow connection establishment to occur in parallel, so the time to connect the last sink should be reduced. The positioning of processes for the second run is shown in figure 6.2.

In a third run (A3), the extra managers are placed in the root host, as shown in figure 6.3.

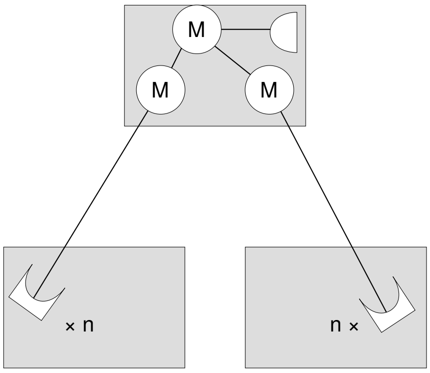


Figure 6.3: Processes for third run (A3)

In each run, the worst results are obtained, *i.e.* the time for full connectivity is measured. The results are listed in the tables in appendix D. The times for getaddr_ind arrival are illustrated in figure 6.4.

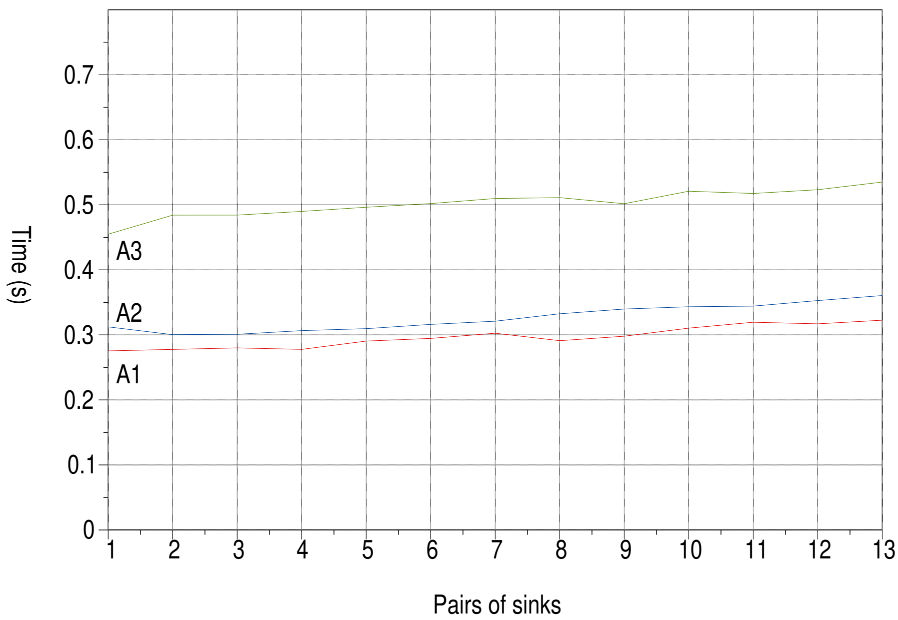


Figure 6.4: getaddr_ind times

It can be seen that placing several managers in the same host in A3 has caused additional delays, but having managers in separate hosts (A2) has only marginally increased times over A1. Although additional delays can be expected by placing additional components in the management, the object of these experiments is to measure scalability, so it is the gradient or shape of the graph that is more relevant, and no difference between A1 and A2 is obvious.

In fact, A1 has a linear fit of $y = 0{\cdot}0041x + 0{\cdot}2679$ and an exponential fit of $y = 0{\cdot}2648x^{0{\cdot}0645}$, while A2 has fits $y = 0{\cdot}0146x + 0{\cdot}3165$ and $y = 03143x^{0{\cdot}1604}$, and the single-manager architecture A1 appears better in both cases. However, it should be noted that this apparent disadvantage of distributed managers must be weighed against the advantage that joining and leaving of sinks involves management interactions that are kept as local as possible, and that many group applications will involve these changes to sink participation more often than additions and removals of sources.

Ideally, many more sinks should have been used to improve the statistical significance of the fits, but implementation limitations of the manager restrict it to 26 sinks, and this applies to the most concentrated structure of A1.

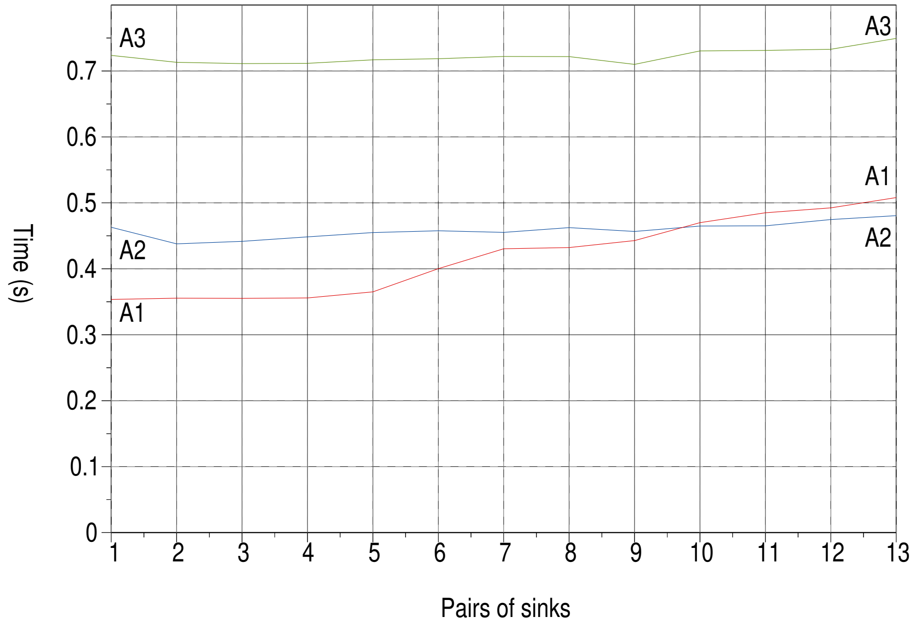


Figure 6.5: First data packet times

From the arrival times of first packets, it is clearer that the intermediate repeaters (A2) have scaled better, with the gradient almost flat, and being overtaken by A1 at around 20 sink. Since the repeaters are in the sink hosts, the first packet is transmitted to each host only once, and the second stage of transmission to the ultimate sinks is between processes on the same host, so delivery is quick.

Problems with these measurements include the use of a non-dedicated host at the root of the hierarchies (so measurements were made during a single, quiet period of use), and that additional managers are not placed on extra hosts, so their processing requirements must be met by taking resources from the sinks in A2 or from the original manager in A3.

7 Conclusions

This final chapter summarises the thesis and its contributions, and indicates further areas of research not covered by the thesis.

7.1 Thesis Summary

Chapter 1 introduced the thesis and its structure. It introduced the issues of multimedia storage and transmission, and multipeer transmission in computer networking. An outline of the Media Channel model was given, and its early architecture was restated, thereby motivating the present work.

Chapter 2 described the nature of multimedia data, and demonstrated its implications for its transmission in real-time and multipeer applications. Issues of the varying degrees of acceptable reliability, the ability for multimedia data to be scaled, filtered, mixed, and hierarchically organised for transmission, and the temporal interdependence of frames of media data for some types were raised.

Chapter 3 reviewed group and multipeer models, communications and management in the light of the requirements and observations of multipeer applications. Existing services and protocols, and their features, were discussed. Additionally, the Media Channel model was described, beginning with an outline of its service characteristics using the ISO multipeer taxonomy, and followed by service primitives more tightly defined than in early versions of the model. The MC service was redesigned to be independent of its underlying support mechanism, and allowed sinks to accept or reject connections from individual sources based on a combination of the source's identity and arbitrary MCT-defined descriptions. The session model was decoupled from its control through a channel, allowing for alternative mechanisms.

Chapter 4 discussed distributed architectures to support the MC service, and offered three alternatives to the architecture used in the original MC model, *i.e.* a single, centralised manager. One new architecture places management in client's end-systems, while the other two divides neighbouring clients into domains, with a manager in each. Managers co-operate with each other in neighbouring domains, with varying layouts.

Chapter 5 defined a protocol to support either of the distributed-manager architectures, and some of the important features of policy enforcement (participation and integrity conditions). It continued by describing the implemented components that could be used to generically support a channel with a given name (a co-operative manager, and a facility to form domain neighbourhoods automatically), additional client and system components to support an audio-conferencing channel, and analogous components for performance evaluation.

Chapter 6 evaluated the Media Channel service, its distributed architectures, its supporting protocol and implementation, and compared the evaluation to requirements identified earlier. Performance was also evaluated, giving predictions for global use.

7.2 Contributions of the Thesis

Improving on an early architecture for the Media Channel model, this thesis has designed and implemented a management system to control and support multimedia group communication. The first version of the model allowed a multimedia group communication or session to be expressed as several individual media channels (potentially M:N (virtual) multipeer communications) of potentially varied types. Channels operated independently of each other, and users participated in the session by participating in each of the relevant channels individually, but could make use of a session description (also distributed by a channel) to automatically configure their channel-participation programs. Information about channel associations that could be applied from end-systems (*e.g.* synchronisation, resource-sharing) could be supplied to sources and sinks also through the session description. The model adopted an approach of allowing applications to use their own choice of communication systems (which have varying addressing modes when used for multicasting), making the model applicable to many types of application.

The present model has improved on the first version by removing the assumption of full connectivity between sources and sinks (due to resource limitations), and brought in the extra parameters to the service needed to allow sinks to choose sources through arbitrary source/stream descriptions. Coupling this with policies associated with the channel, the value of individual multipeer connections within a channel, and of the channel as a whole, can be ascertained, and a soft integrity can be maintained in the distributed domain/manager architectures. The stream descriptions also allow some static or long-term state in the source to be supplied to sinks, which is important for some multimedia applications such as video, where some framing information would normally only be passed at the start of a transmission in the unicast case, but is needed in the multicast case for additional sinks that join some time later. It is also possible to change stream descriptions at any time, so the application can change that initial information as required, and sinks can re-assess their acceptance of a stream.

The new session model divides local configuration of sessions (*e.g.* synchronisation) from global configuration, which includes integrity conditions on the session in terms of the ATIs of its constituent media channel. Group sessions can still be supported through the use of session channels, allowing hierarchies of sessions to be defined, while allowing channels (and sessions) to belong to more than one session, making the model of temporary integration of sessions simple — participants of one session join a channel or another session (represented by a channel) when the session controller places the additional channel within the first session's description. However, other control systems for sessions are permitted.

The choice of implementation scales well for large numbers of sinks, and is flexible in allowing different communication systems to be used in the different domains that a channel covers. In a hierarchy of domains, local domains in which the participants are attached can make use of improved LAN multicasting facilities, with the repeater(s) acting as gateways to internet multicasting.

The interfaces between domains, as realised by the repeaters, can be used as locations for application-defined services that support heterogeneous multipeer communications such as filtering, and other services such as mixing of streams (to reduce bandwidth) and storing of current state (for new sinks), but equally, these interfaces can be made to carry only connection information, and importantly, interfaces in the same channel may be used in different ways for maximum flexibility in configuration of the channel.

In summary, the Media Channel model achieves the following:

- a channel supports a M:N interaction between distributed components of an application;
- it is configurable for a particular application;
- it can support monitoring of participation and enforcement of integrity conditions;
- the MC service can be supported by various mechanisms and architectures, including support for widely distributed and migrating management;
- co-operating applications are realised as several channels, and such sessions can be represented as and supported by a channel, allowing flexible management of multiple channels.

Configuration aspects of channels include:

- a MCT to identify the application;
- the underlying communication system between application components chosen by the application;
- application-defined stream descriptions;
- choice of multicast addressing mode.

A distributed implementation of co-operating managers connected by repeaters has these merits:

- repeaters support inter-manager communication, both for connection and integrity information;
- repeaters can support application-specific stream processing such as mixing and filtering;
- repeaters carrying data allow different multicast addressing modes to be applied either side of such a repeater, and provide a way of supporting root-initiated addressing;
- sinks joining or leaving a channel have only local effects with regard to addressing (*i.e.* when integrity information is not affected), so

scalability for sinks is improved.

7.3 Further Work

There are many shortfalls of the Media Channel model, its services, and implementation, and these are briefly outlined below.

Provision of current state to new sinks

If a continuous media stream is temporarily stopped, such as when a recorded video sequence is controlled, the source may choose to represent this by suspending transmission (since this may save resources), and relying on existing sinks displaying the last frame. In another situation, inter-coding may be used for some frames, and these will likely by interspersed with intra-coded frames to negate errors building up in transmission. If a sink joins in the middle of such video streams, it must wait for an intra-coded frame before it can render any useful image. In the second case, this will happen within a finite time, while in the first case, it must wait indefinitely.

When the time to wait is long, as defined by the application, a mechanism should exist that allows a newly joined sink to obtain the current state of a stream. Some multipeer systems may provide a mechanism for this, and so it would not be an issue for the management system. Nevertheless, it may be helpful to treat it as so, perhaps allowing a manager to ask an existing sink for the relevant data, on the demand of the new sink.

QoS-guarantee-based integrity

The integrity conditions enforced by the manager are based on the acceptance or rejection of a multipeer connection (synonymous with a source) by each sink. Sinks will not be told to connect in any way to a multipeer connection if its integrity condition is not met. Acceptance by a particular sink depends on the identification of the source, a description of the QoS required by the stream, and the sink's local resources. If the sink does connect, but subsequently finds that network resources are not sufficient, then problems may arise if it indicates this by rejecting the connection, as its addressing information would become invalid, and it may not be able to determine if the network becomes capable in the future without periodically accepting the connection with a mind to test it.

Instead, it may be preferable for the sink to indicate QoS resource failure (that it has detected from its service access point for an accepted connection) to the manager, which will apply this to QoS-guarantee integrity. If this fails, the connection is still considered to exist, but the clients using it are informed, and they may be able to place it in an idle state which allows remaining QoS capacity to be utilised while the connection's QoS reservation is still requested, and its clients can be informed when it becomes available. Even if such behaviour is not available from the service of the connection, a conference organiser may still be notified, and may make a manual decision to alter the nature of the conference. The Media Channel service must be extended to allow sinks to express QoS failure, but the infrastructure to support this, the passing of counters between managers, already exists.

Other implementations

The first architecture of those described in chapter 4 has been implemented, namely the domain hierarchy. Parts of that implementation (managers, repeaters, sources and sinks) are also applicable to the second architecture (the flat tree architecture). However, much more extensive support would be required to dynamically determine the topology of such domains, and to establish them. The third architecture, which we called per-client management, abandons many of the components of the first two implementations, and instead needs a reliable multicast transport to support the MC service.

Sink AGIs

The current protocol does not allow information about individual sinks to be passed through repeaters. Instead they are coalesced into simpler counters which are relevant to the evaluation of C-AGIs and ATIs, but this prevents evaluation of general sink AGIs. The protocol will need to be revised if such an integrity condition is considered important.

Appendix A: Distributed Processing Environment

There are a number of modules that make up this environment, written to support the implementation of the Media Channel (MC) Model. They are the transport modules, the object module, the named-service module, the data representation module, and the network-safe handle module.

A.1 Transport Module

This module defines *transport end-points* (TEPs) which are identified by transport addresses (a network address plus a port number). Two TEPs can be attached to each other, and a full-duplex stream of arbitrarty data can pass between them. Attachment of one TEP to another is set in the context of a *transport proxy* in each end-system, and a TEP will have a local proxy for each attachment. When a TEP is instructed to attach to a remote TEP, both TEPs will consult their respective configurations to generate a proxy to locally represent the remote TEP. The TEP and the proxy may be specialised for a particular purpose.

The API consists of some addressing types:

- `NAddr` — a network address;
- `TId` — a local transport-level handle (port number);
- `TAddr` — a transport address consisting of each of these.

It defines two classes to represent the TEP and transport proxy. The `TEP` interface is as follows:

```
class TEP {
  virtual TProxy *open_proxy(const TAddr &a) { return 0; }
  virtual void close_proxy(TProxy *p) { }

  virtual void bound() { }
  virtual void unbound() { }

protected:
  TEP();
```

```
public:
  virtual ~TEP();
  int bind(TId = 0);
  int is_bound();
  int unbind();
  TAddr get_name();

  TProxy *connect(const TAddr &);

  void collect(FileEventSet &);
  void process(FileEventSet &);
};
```

After creation, a `TEP` must be bound to a local address. The network component of this is available in the process environment, and the local identifier is supplied as an argument to `bind()`, with 0 meaning use any available value. A `TEP` can subsequently be unbound with `unbind()`, tested for its bind state with `is_bound()`, and the functions `bound()` and `unbound()` are invoked as this state changes. If bound, `get_name()` returns the full address of the `TEP`.

To attach the `TEP` to another whose address has been obtained, the address is supplied to the `connect()` call. The remote `TEP` will be contacted, and then each `TEP` will invoke its own `open_proxy()` as a factory to generate a `TProxy` representing a transport proxy. If the attachment is to be removed later on, the pointer to the relevant proxy is passed `close_proxy()`. These two functions are intended to be specialised for particular applications by creating and deleting objects of types derived from `TProxy`, below:

```
class TProxy {
  virtual void starting() { }
  virtual void stopping() { }
  virtual void activity() { }

  virtual int receptive() { return 1; }

protected:
  TProxy();

public:
  virtual ~TProxy();
  long send(const roblock &);
  long recv(const block &, int fill = 0);
  long discard(long);
```

```
  void disconnect();
  TAddr get_peer();
  TEP *get_ep();

  int is_running();
};
```

After being created by `TEP::open_proxy()`, the `TProxy` will be set up so that data can be streamed in blocks to the remote proxy using `send()`, and data from the remote proxy can be received using `recv()` (`fill` set means do not return until the supplied block is filled). `discard()` discards a number of octets. `starting()` is called just as the `TProxy` becomes connected, and `stopping()` is called just before attachment is terminated. `activity()` is called when a `recv()` would not block, and `receptive()` is called to ask if the `TProxy` is ready to receive data.

A.2 Object Module

The object module makes use of the transport module to define *object end-points*, (OEPs) which have object addresses (a transport address plus a 32-bit handle), and can be attached through *object proxies*. Several OEPs are bound to a TEP, using the handle to to distinguish them. Communication between these proxies consists of variable-length messages, *i.e.* block boundaries are preserved, and the transport proxies multiplex these messages when several pairs of attached OEPs span the same pair of TEPs.

The API defines basic types:

- `OId` — a local object handle;
- `OAddr` — an object address consisting of a `TAddr` and an `OId`.

It also defines classes `OTEP` and `OTProxy`, which are specialisations of `TEP` and `TProxy`, and must be used to connect objects together. Class `OEP` is an OEP represention with the following interface:

```
class OEP {
  virtual OProxy *open_proxy(const OAddr &a) { return 0; }
  virtual void close_proxy(OProxy *p) { }

  virtual void bound() { }
```

```
  virtual void unbound() { }

public:
  OEP();
  virtual ~OEP();

  int bind(OTEP &, OId = 0);
  void unbind();
  int is_bound();
  OAddr get_name();

  OProxy *connect(const OAddr &);
};
```

Like `TEP`, it can be bound to an address with `bind()`, unbound with `unbind()`, tested for bind state with `is_bound()`, informed of change to its bind state through its virtual functions `bound()` and `unbound()`. `get_name()` returns its full address when bound. It has a `connect()` method to attach it to a remote `OEP`, and configurable `open_proxy()` and `close_proxy()` functions to create and destroy objects of classes derived from `OProxy`.

```
class OProxy {
  virtual void starting() { }
  virtual void stopping() { }
  virtual void activity() { }

protected:
  OProxy();

public:
  virtual ~OProxy();

  OAddr get_peer();
  OAddr get_name();

  OEP *get_ep();

  long send(const roblock &);
  long send(const roblock [], size_t);
  long recv(const block &, int fill = 0);
  long discard(long);
  long remaining();
  void disconnect();
};
```

Like `TProxy`, `OProxy` has virtual functions `starting()` and `stopping()` to inform the derived class that the proxy is fully connected. `send()` sends a single message to the remote proxy, and can send a single block or several blocks as such a message. `activity()` is called when a message is ready to be received using `recv()` or discarding using `discard()`. `remaining()` indicated the remaining octets in the message, if it is to be received in parts, or to pre-allocate space for the message.

A.3 Named-Service Module

A.3.1 Model

The interface module provides communication of raw data. A user of the DPE may wish to send discrete messages of various types which provide a service. Also, a proxy (or any end-point of communication) may need to send or respond to messages of more than one type of service, and while message types within a single service can be uniquely numbered, end-points supporting several services need to negotiate with their peers the base numbers of the services they provide.

In this model, the end-point, which sends and receives data in streams, is encapsulated in a *service warden*, and uses the streams to negotiate with a warden at the other end. A warden also maintains a set of *service providers* and *service users*. A service is a set of message types (numbered 0 to *n*-1), and has a unique name. Service users generate messages of these types, while service providers can respond to them. The warden allocates a base number to each of its providers, so that each message type can have a number unique within the context of the warden, by adding its number to its service's base number.

When two wardens are first connected, they ask each other for specific services (using FIND messages), based on the users they each have. When requested a named service, a warden attempts to find it in its provider set, and returns the base number it allocated or an indication of failure (using a FOUND message).

When a FOUND message is received for a particular service, the warden notifies the service user of the base number of that service, which it will add to its message type numbers when sending USER messages.

A.3.2 Service

The user of the named-service module can instantiate a service warden and configure it with user-supplied data transmission facilities. It can then attach service users, which have names (the name of the service), and providers, which have names and lengths (the number of message types).

The user can write discrete messages through a service user, and the corresponding service provider can be configured to receive the messages. Messages have a type number, and arbitrary data of any length.

Service users are informed when a corresponding service provider is added to or removed from the remote warden. Disconnection implies removal of all providers.

A.3.3 API

The API consists of three C++ classes: `Service::Warden`, `Service::Provider` and `Service::User`.

A.3.3.1 Class Service::Warden

`Service::Warden` has the following public interface:

```
// in scope Service
  class Warden {
  public:
    Warden();
    virtual ~Warden();

    /* activation */
    int is_active();
    void activate();
    void deactivate();

    /* I/O */
    virtual long write(long msgtype, const octet_roblock &) = 0;
    virtual long read(const octet_block &) = 0;
    virtual long discard(long) = 0;
    virtual long remaining() = 0;
    virtual void find(const char *) = 0;
    void found(const char *, long);
```

```
    long search(const char *);
    void message_ready(long type);
  };
```

The user is expected to derive its own class from `Service::Warden`, supplying the virtual functions which support communication. A warden should be informed when it is connected to a peer by calling `activate()`, or when it is not connected by call calling `deactivate()`.

A warden should be informed when a FOUND message is received by calling `found()`, specifying the service name, and the base number. `search()` should be called to find the base number of a local provider attached to the warden. The warden will invoke `find()`, specifying a service name, to indicate that it wishes to send a FIND message for that service.

When a USER message is received, message_ready should be called, specifying the message type. The warden can then read so many bytes of the message by calling `read()`, discard so many bytes with `discard()`, and count the remaining bytes in the message by calling `remaining()`. The message can be read in parts if necessary. The warden can write an entire message at once using `write()`, specifying the message block and type.

A.3.3.2 Class Service::Provider

`Service::Provider` has the following public interface:

```
// in scope Service
  class Provider {
  public:
    Provider(const char *, long);
    virtual ~Provider();

    virtual void dispatch(int32 msgtype) = 0;
    long read(const block &msg);
    long discard(long);
    long remaining() const;

    void attach(Warden &);
    void detach();
    int attached() const;
    Warden *get_warden() const;
  };
```

The user is expected to derive its own class from `Service::Provider`, supplying the virtual function to process messages of various types.

When a provider is instantiated, it must be configured with a service name and the number of message types it recognises. It can then be attached to or detached from wardens with `attach()` and `detach()`. `attached()` indicates its current status, and `get_warden()` returns the warden to which it is attached.

`dispatch()` is invoked when a message of the specified type is ready to be received. `read()` reads a block from the message, `discard()` discards a number of bytes of the message, and `remaining()` returns the number of bytes left in the message.

A.3.3.3 Class Service::User

`Service::User` has the following public interface:

```
// in scope Service
  class User {
  public:
    User(const char *);
    virtual ~User();

    virtual void bound();
    virtual void unbound();
    int is_bound();
    long write(int32 msgtype, const roblock &msg);

    void attach(Warden &);
    void detach();
    int attached() const;
    Warden *get_warden() const;
  };
```

The user is expected to derive its own class from `Service::User`, supplying the virtual functions that inform it when a corresponding service provider is attached to the remote warden.

When a service user is instantiated, it must be configured with the name of the service it requires. It can then be attached to or detached from wardens with `attach()` and `detach()`. `attached()` indicates its current status, and `get_warden()` returns a pointer to the warden to which it is attached.

`bound()` is invoked when a service provider exists at the other end of communication. `unbound()` is invoked otherwise. `is_bound()` returns `FALSE` only after `unbound()` and before `bound()`.

While bound, the user can send whole messages of particular types using `write()`. The type number supplied is the number valid within the service — it will be transparently converted to the negotiated value before transmission. There is no negotiated value while unbound.

A.4 Data Representation Module

A.4.1 Model

This module converts typed, architecture-dependent data into untyped data suitable for transmission across networks to machines of other architectures. Types are defined to represent untyped data: octets (single bytes), octet blocks (arrays of octets), and octet buffers (blocks with a current pointer). It then defines types whose names describe their format (*e.g.* a 16 bit signed integer is called `int16`), and provides marshalling functions to load an octet buffer with a series of typed data, and corresponding unmarshalling functions to extract them again after transmission.

A.4.2 API

The API defines `uint8`, `uint16`, `uint32`, `uint64` as unsigned integers of the indicated bit widths. Similarly, `int8`, `int16`, `int32`, `int64` are signed integers. Floating-point types have names of the form `fp`*`MM`*`e`*`EE`*, where *`MM`* is the mantissa width and *`EE`* is the exponent width. A `string` type is also defined as a memory-managed, zero-terminated character string. Template types for memory-managed bounded and unbounded sequences, and arrays of parameterised type `T` are also defined: `bsequence<T>`, `sequence<T>`, `array<T>`.

The type `uint8` is used as an octet. An octet block type `block` refers to a read/write array of octets (`uint8 *`) with a particular size (`size_t`), while `roblock` describes a read-only array (`const uint8 *`). There are also read/write buffers (`buffer`) and read-only buffers (`robuffer`) which consist of a `block` or `roblock` and a current offset within the array.

Marshalling functions have the form:

```
buffer &type_marshal(buffer &b, type v);
```

where the argument `v` is written into the buffer at the current position, which is then incremented. If the buffer refers to a block pointing to `NULL`, nothing is written, but the offset is still incremented, allowing the required size of a block to be determined. Unmarshalling functions have this forms:

```
robuffer &type_unmarshal(robuffer &b, type &v);
```

which interprets the octets at the current position in the buffer as the coding for a *`type`* object whose value is written into `v`. The offset if incremented as with marshalling.

A.5 Network-Safe Handle Module

A.5.1 Model

This module provides facilities for mapping numeric identifiers, possibly transferred across a network and back, into internal pointers, ensuring validity.

A.5.2 Service

The user creates a database of network-safe handles, and can have handles allocated to user-supplied data. The handles can be manipulated across networks and returned, then mapped to the original data if it still exists. Handles can be de-allocated, making them invalid.

A.5.3 API

The template class `TypedNetHandle` is defined as follows:

```
template <class T>
struct TypedNetHandle {
  class DB {
  public:
    DB(int hashsize = 200);
    NetHandle::ID allocate(T *p);
```

```
    T *resolve(NetHandle::ID i);
    void deallocate(NetHandle::ID i);
  };
};
```

The user completes the template by supplying the user-data's type `T`, and can create databases of type `TypedNetHandle<T>::DB`. A pointer to a `T` can be allocated with `allocate()`, which returns a handle of type `NetHandle::ID` (a `uint32`). It can subsequently be de-allocated with `deallocate()`. Between allocation and de-allocation, the handle can be resolved with `resolve()` to give the original pointer.

A.6 Interface Definition

The modules described above may be combined with the help of an interface definition in an interface-definition language (IDL). This allows complex types to be expressed in a manner independent of programming language used, and an IDL compiler can then generate the code necessary to marshal and unmarshal data of those types using the facilities of section A.4 as primitives. The IDL also allows named interfaces to be defined, which consist of sets of named methods with arbitrarily typed arguments. The methods are unidirectional, similar to `oneway` methods in CORBA IDL [IONA 95], or `ANNOUNCEMENT` operations in ANSA [APM 92].

Primitive integer, floating-point and string types of the IDL are the same as those in section A.4. An unbounded sequence of *`type`* objects is expressed as *`type`*`<>`, while a sequence bounded by *`max`* is expressed as *`type`*`<`*`max`*`>`. An array of *`size`* *`type`* objects is expressed as a *`type`*`[`*`size`*`]`.

To define a new type and its marshalling functions, an IDL declaration of the following form is made:

```
define name type;
```

An interface has the following form:

```
interface name {
  methods
}
```

and each method has one of these two forms:

```
mname(arguments)
out-mname in-mname(arguments)
```

Interfaces and type definitions can be collected into named modules which simply place the names in a particular namespace:

```
module name {
  type-definitions
  interfaces
}
```

Complete names consist of their dot-separated components, so a type `X` in module `Y` is referred outside `Y` as `Y.X`. Modules can also appear inside other modules. Generated C++ code uses the scope operator `::` to separate components, and modules are encoded into `namespace`s or `struct`s.

For each interface, the IDL compiler will generate a class *`name`*`::Producer` derived from `Service::User`, and gives it an additional function for each method, with that method's arguments. The function will marshal the arguments into a block, and pass this through `Service::User::write`, specifying a number to identify the method within the interface. The name of the function is either *`mname`* or *`out-mname`*.

Correspondingly, a class *`name`*`::Consumer` is derived from `Service::Provider`, and has one extra function per method called *`mname`* or *`in-mname`*. It has the same arguments as the method, and these have been unmarshalled from a block received by *`name`*`::Consumer::dispatch` which completes the pure virtual function of `Service::Provider`. `dispatch` is supplied the method number as an argument, reads the remaining octets into a temporary block, and uses the method number to decide what arguments to unmarshal from the block, and to select one of the IDL methods to invoke.

A class `ServiceProxy` is derived from both `Service::Warden` and `OProxy`, and their virtual functions are hooked into each other by implementing the FIND, FOUND, and USER messages as structured messages passed through `OProxy::send` and `OProxy::recv`. This combined class can be bound through an `OEP` to a remote `OProxy`, and at the same time, service providers and service users can be attached to it. The service users will attempt to find their named services in the remote proxy, and successful responses will cause those users to be `bound()`, indicating that the services provided by the remote proxy are of the right type.

The users and providers attached to the `ServiceProxy` are derived from `Service::User` and `Service::Provider` respectively, as provided by the IDL compiler.

Appendix B: Event Management

Programs used in the implementation of the Media Channel (MC) Model must monitor communication end-points for incoming messages, and must also respond to timed events. The following modules have been written to meet these requirements in a consistent and generic way.

The alarm module handles timed events, while the file event module handles asynchronous events on sockets and files.

B.1 Alarm Module

B.1.1 Model

The alarm module provides facilities to manage timed events (alarms). A program will usually maintain a single database of alarms, and make this available to various unrelated components of the program that have timing needs. The components will share the database, managing their own alarms, and not interfering with each others'. The alarm database is frequently consulted to check for any pending alarms.

B.1.2 Service

The user may create an alarm database, and set or cancel alarms in the database at will. The database may be polled to determine which alarms are pending, causing an action on an object to be triggered.

B.1.3 API

The classes `Alarm` and `Alarm::DB` to represent an alarm and a database respectively.

B.1.3.1 Class Alarm

The class `Alarm` has the following public interface:

```
class Alarm {
public:
  enum status { OFF, OK, PENDING };

  class DB;

  static DB main;

  Alarm();
  virtual ~Alarm();
  virtual void raised();
  void set(const Instant &when, DB &db = main);
  void cancel();
};
```

The user is expected to derive its own class from `Alarm`, supplying a function to be invoked when the alarm action is triggered, and placing user-defined data in the derived class.

An `Alarm` can exist inside or outside a database. To place it in a database, to go off at a specific time, `set()` should be called specifying the time and the database to use (by default, a static database `Alarm::main` is used). It can later be cancelled, whether it has been triggered or not, by calling `cancel()`. The virtual function `raised()` is called if the alarm is triggered.

B.1.3.2 Class Alarm::DB

The class `Alarm::DB` has the following public interface:

```
// in scope Alarm
  class DB {
  public:
    DB();
    ~DB();

    status poll(Instant &next);
    status poll_diff(Instant &next);
    void process();
  };
```

The class `Alarm::DB` holds a set of objects of base type `Alarm` in chronological order. Calling `process()` will trigger any currently pending alarms (doing nothing if there are none).

The polling functions, `poll()` and `poll_diff`, return a status value indicating the state of the next alarm to be triggered: `Alarm::OFF` means that there are no alarms (*i.e.* the time to the next alarm is infinite); `Alarm::PENDING` means an alarm is currently pending (late); `Alarm::OK` means the next alarm to be triggered is in the future.

When there is an untriggered alarm (either in the past or the future), the argument `next` contains information on how long to wait until that alarm. For `poll()`, that information consists of an absolute time; for `poll_diff`, it is a delay from the current time. For a pending alarm, the absolute time is now, or the delay zero.

B.2 File Event Module

B.2.1 Model

Unix processes deal with most external communication through file descriptors, which are variously available for input and output, and errors may occur on them. A process can remain idle, and release processing time to other processes by going to sleep, making it unable to monitor file descriptors for input, output and error events. Therefore, a Unix process has the option to specify which type of events on which sockets should force it to resume execution, and a timeout may be specified so that the process will resume anyway.

The file event module allows specifications of which descriptors to monitor to be made, and makes use of the alarm module to calculate the shortest necessary timeout.

B.2.2 Service

The user can create an event specification and modify it. This specification can then be used as the criterion for re-awakening when the user process sleeps. After re-awakening, the specification now indicates which of the monitored events actually happened.

B.2.3 API

The class `FileEventSet` is defined to represent an event specification.

B.2.3.1 Class FileEventSet

The class `FileEventSet` has the following public interface:

```
class FileEventSet {
public:
  FileEventSet();
  void reset();
  int await(Alarm::DB & = Alarm::main);

  void set_input(int fd);
  void clear_input(int fd);
  int test_input(int fd) const;

  void set_output(int fd);
  void clear_output(int fd);
  int test_output(int fd) const;

  void set_error(int fd);
  void clear_error(int fd);
  int test_error(int fd) const;
};
```

By default, a `FileEventSet` specifies that no file descriptors are to be monitored. The specification can be changed by using `*action_type*()`, where `*action*` is either `set` (meaning enable monitoring) or `clear` (meaning disable monitoring), and `*type*` is the event type: `input` `output` or `error`. Calling `reset()` returns the specification to the default state after use.

`await()` makes the program sleep until at least one file descriptor event occurs, or an alarm is due (processing all pending alarms in that case), as specified by the alarm database.

After `await()`, `test_*type*()` may be called to test if a particular event occurred.

B.3 Utilisation

The classes `Alarm::DB` and `FileEventSet` can be used in a single threaded process to orchestrate independent components that must respond to synchronous and asynchronous events. A single `Alarm::DB` is instantiated (the static `Alarm::main` can be used for this), and separate components are instructed to use it to register all their timed events as objects of base class `Alarm`.

The main body of the program then iterates through a loop consisting of an event collection phase, a sleeping phase and a processing phase. In the collection phase, a `FileEventSet` object is reset to its default state, then each component is asked to specify which asynchronous events it wishes to respond to, by modifying the `FileEventSet`. In the sleeping phase, `FileEventSet::await()` is called, passing the alarm database as an argument. This phase is ended when an asynchronous event as specified in `FileEventSet` occurs, or a synchronous one as specified by the alarm database (any pending synchronous ones are processed now). The `FileEventSet` object now contains an indication of the asynchronous events that occurred, and in a similar manner to the collection phase, the processing phase involves offering each program component the modified `FileEventSet`, allowing the component to test for the events that it wanted to monitor.

Appendix C: Interface Definitions

This appendix summarises the interfaces used between a manager and a client (the interfaces in the module `MC`), and between a factory and a client (the interfaces `FactoryServer` and `FactoryClient`). They are expressed in the form of interface definition language (IDL) files, intended to be fed into the code generator of the DPE in appendix A.

C.1 Manager-Client Interfaces

```
module MC {
  define Ref uint32;
  define CounterId uint16;
  define CounterVal int32;
  define CounterVals CounterVal<>;

  define UserAddr uint8<>;
  define Manager object_address;

  define QoSInfo uint8<>;
  define Agent object_address;
  define Agents Agent<>;
  define SourceScore struct {
    Agent source;
    CounterVals scores;
  };
  define SourceScores SourceScore<>;
  define SourceSpec struct {
    Agents sources;
    QoSInfo qosinfo;
  };
  define SourceSpecs SourceSpec<>;

  interface SourceManager {
    join_req join_ind(string mct; Agent agt; QoSInfo qos);
    joinrpt_req joinrpt_ind(string mct; Agent agt;
           SourceSpecs qos; Manager mgr);
    leave_req leave_ind();
    setqos_req setqos_ind(QoSInfo qos);
    setqos_req setqos_ind(SourceSpecs qos);
```

```
    setscore_req setscore_ind(SourceScores scores);

    conn_rsp conn_cfm(Ref mgr, usr);

    getaddr_rsp getaddr_cfm(Ref mgr; UserAddr addr);
  }

  interface ManagerSource {
    conn_req conn_ind(Ref mgr);
    noconn_req noconn_ind(Ref usr);
    setqos_rsp setqos_cfm(Ref usr; Agents okay);
    setscore_rsp setscore_cfm(SourceScores scores);

    getaddr_req getaddr_ind(Ref usr);
    useaddr_req useaddr_ind(Ref usr; UserAddr addr);
  }

  interface SinkManager {
    join_req join_ind(string mct; Agent agt);
    joinrpt_req joinrpt_ind(string mct; Agent agt; Manager mgr);
    leave_req leave_ind();

    conn_rsp conn_cfm(Ref mgr, usr);
    setqos_rsp setqos_cfm(Ref mgr; Agents okay);
    setscore_rsp setscore_cfm(Ref usr; SourceScores scores);

    getaddr_rsp getaddr_cfm(Ref mgr; UserAddr addr);
  }

  interface ManagerSink {
    conn_req conn_ind(Ref mgr; SourceSpecs qos);
    noconn_req noconn_ind(Ref usr);
    setqos_req setqos_ind(Ref usr; SourceSpecs qos);
    setscore_req setscore_ind(Ref usr; SourceScores scores);
    getaddr_req getaddr_ind(Ref usr);
    useaddr_req useaddr_ind(Ref usr; UserAddr addr);
  }

  interface ManagerManager {
    set_counter counter_set(Agent agt; CounterId id; CounterVal val);
  }
}
```

C.2 Factory-Client Interfaces

```
interface FactoryServer {
  getmgr_req getmgr_ind(string name);
}
```

```
interface FactoryClient {
  getmgr_rsp getmgr_cfm(string name; object_address addr);
}
```

Appendix D: Timing Results

Sinks/2	A1	A2	A3
1	0.275389	0.312355	0.454838
2	0.277710	0.300353	0.484179
3	0.280009	0.300954	0.484244
4	0.277760	0.306586	0.489915
5	0.290488	0.309624	0.496317
6	0.294617	0.316174	0.501987
7	0.302584	0.321076	0.509768
8	0.291128	0.332482	0.511090
9	0.297998	0.339790	0.501753
10	0.310406	0.343290	0.520976
11	0.319463	0.344295	0.517506
12	0.317042	0.352810	0.523259
13	0.322622	0.360537	0.535029

Table D.1: getaddr_ind times (s)

Sinks/2	A1	A2	A3
1	0.353553	0.462993	0.723595
2	0.355473	0.437851	0.713126
3	0.355202	0.441649	0.711166
4	0.355816	0.448434	0.711577
5	0.365046	0.454975	0.716885
6	0.400207	0.457678	0.718572
7	0.430398	0.455233	0.721998
8	0.432229	0.462457	0.721781
9	0.442807	0.456644	0.709992
10	0.469991	0.464870	0.730378
11	0.484968	0.465168	0.731162
12	0.492426	0.474699	0.732905
13	0.508014	0.480620	0.749354

Table D.2: First packet times (s)

References

[Almeroth 96]
Kevin C. Almeroth, Mostafa H. Ammar
Collecting and Modeling Multicast Group Behaviour Data in the Internet's Multicast Backbone (MBone)
Networking and Telecommunications Group, College of Computing, Georgia Institute of Technology, August 1996.

[Altenhofen 93]
Michael Altenhofen, Jürgen Dittrich, Rainer Hammerschmidt, Thomas Käppner, Carsten Ktuschel, Ansgar Kückes, Thomas Steinig
The BERKOM Multimedia Collaboration Service
Proc. 1st ACM Multimedia Conference, Los Angeles, 1993, pp. 457-463.

[APM 92]
ANSAware 4.0 Application Programmer's Manual
Architecture Projects Management Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK.

[Atwood 96]
J.W. Atwood, O. Catrina, J. Fenton, W. Timothy Strayer
Reliable Multicasting in the Xpress Transport Protocol
Proceedings of the 21st Local Computer Networks Conference, Minneapolis, Minn., October 13-16, 1996.

[Ballardie 95]
Tony Ballardie, Jon Crowcroft
Core Based Tree (CBT) Multicast; An Analysis of Multicast Routing Architectures
Technical Report `ieee-ton95-cbt.ps.Z`, Department of Computer Science, University College London.

[Birman 89]
K. Birman, A.K. Marzullo
ISIS and the META Project
Sun Technology, Summer 1989, pp. 90-104.

[Berners-Lee 94]
T. Berners-Lee, L. Masinter, M. McCahill (Eds)
Uniform Resource Locators (URL)
Network Working Group RFC 1738, December 1994.

[Berners-Lee 96]
T. Berners-Lee, R. Fielding, H. Frystyk
Hypertext Transfer Protocol — HTTP/1.0
Network Working Group RFC 1945, May 1996.

[Borenstein 93]
N. Borenstein, N. Freed
MIME (Multimedia Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies
Network Working Group RFC 1521, 1993.

[Braden 94]
R. Braden, D. Clark, S. Shenker
Integrated Services in the Internet Architecture: an Overview
Network Working Group RFC 1633, July 1994.

[Campbell 92]
A. Campbell, G. Coulson, F. García, D. Hutchison
A Continuous Media Transport and Orchestration Service
ACM SIGCOMM '92, Maryland USA.

[Davies 92]
N. Davies, G. S. Blair, G. Coulson, N. Williams
A Programmer's Guide to the Base Services Platform
Internal Report, Distributed Multimedia Systems Group, Lancaster 1992.

[Deering 89]
Steve Deering
Host Extensions for IP Multicasting
Network Working Group RFC 1112, August 1989.

[Deering 95]
S. Deering, R. Hinden
Internet Protocol, Version 6 (IPv6) Specification
Network Working Group RFC 1883, December 1995.

[Diot 97]

Christophe Diot, Walid, Dabbous, Jon Crowcroft
Multipoint Communication: A Survey of Protocols, Functions and Mechanisms
IEEE Journal on Selected Areas in Communications, Vol. 15, No. 3, April 1997.

[Eriksson 94]

H. Eriksson
MBONE: The Multicast Backbone
Communications of the ACM 37, 8 (August 1994), pp. 54-60.

[Fielding 97]

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee
Hypertext Transfer Protocol — HTTP/1.1
Network Working Group RFC 2068, January 1997.

[Floyd 97]

Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L.
A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing
IEEE/ACM Transactions on Networking, December 1997, Volume 5, Number 6, pp. 784-803.

[García 95]

Francisco García, Andreas Mauthe, Nicholas Yeadon, David Hutchison
QoS Support for Video & Audio Multipeer Communications
Lancaster University, Internal Report, January 1995.
`ftp://ftp.comp.lancs.ac.uk/pub/ISO_QOSF/ps.format/24N9502.ps`

[GNU]

GNU's Not Unix
`http://www.gnu.org/`

[Gupta 93a]

Amit Gupta, Wendy Heffner, Mark Moran, Clemens Syzperski
Network Support for Realtime Multi-Party Applications
Tenet Group, Computer Science Division, Department of EECS, University of California at Berkeley & International Computer Science Institute, Berkeley.

[Gupta 93b]
Amit Gupta, Mark Moran
Channel Groups: A Unifying Abstraction for Specifying Inter-stream Relationships
Technical Report TR-93-015, Tenet Group, Computer Science Division, University of California at Berkeley & International Computer Science Institute, March 1993.

[Gupta 95]
Gupta A, Howe W, Moran M, Nguyen Q
Resource sharing for multi-party real-time communication
Proc. IEEE INFOCOM'95.
`http://tenet.berkeley.edu/tenet-papers.html`

[Handley 93]
Mark T. Handley, Peter T. Kirstein, M. Angela Sasse
Multimedia Integrated Conferencing for European Researchers (MICE): Piloting Activities and the Conference Management and Multiplexing Centre
Research Note RN/93/22 Dept. Comp. Sci. UCL, London.

[Handley 94]
Mark Handley, Stuart Chapman
Specification of the MICE Conference Management and Multiplexing Centre Version 2.0
Dept. Comp. Sci. UCL, London, Internal Report, January 1994.
`ftp://cs.ucl.ac.uk/mice/reports/draft-cmmc-spec-2.ps.gz`

[Handley 95]
M. Handley, I. Wakeman
CCCP: Conference Control Channel Protocol: A scalable base for building conference control applications
Proc. of SIGCOMM 95.

[Heijenk 94]
G. J. Heijenk, X. Hou, I. G. Niemegeers
Communication Systems Supporting Multimedia Multi-user Applications
IEEE Network, January/February 1994, pp. 34-44.

[Hinden 95]
R. Hinden, S. Deering (Eds.)
IP Version 6 Addressing Architecture
Network Working Group RFC 1884, December 1995.

[Hofmann 96]
M. Hofmann
Adding Scalability to Transport Level Multicast
In Proc. Third International COST 237 Workshop, Barcelona, November 1996.

[InfoZIP]
InfoZIP Home Page
`http://www.cdrom.com/pub/infozip/`

[IONA 95]
Orbix Programmer's Guide
IONA Technologies Ltd, 8-34 Percy Place, Dublin 4, Ireland.
`http://www.iona.com/`

[ISO 93a]
ISO IEC JTC 1
Information Technology — Digital Compression and Coding of Continuous-Tone Still Images
International Standard ISO/IEC IS 10918, 1993.

[ISO 93b]
ISO IEC JTC 1
Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media up to about 1.5Mbit/s
International Standard ISO/IEC IS 11172, 1993.

[ISO 94]
ISO IEC JTC 1
Information Technology — Generic Coding of Moving Pictures and Associated Audio
International Standard ISO/IEC IS 13818, 1994.

[ISO 95]
ISO/IEC JTC1/SC6
Draft text on the subject of “Multipeer Taxonomy”
ISO/IEC JTC1/SC6 N9477, March 17,1995.

[ITU-T 93]
ITU-T H.261
Video Codec for Audiovisual Services at p×64kbits
International Telecommunications Union Telecommunications Standardisation Sector, ITU-T Recommendation H.261, March 1993.

[Kirstein 97]
Peter T. Kirstein, R. Bennet
Recent Activities in the MERCI Conferencing Project
Proceedings JENC8, 1997, pp. 923-1..11.

[Kim 95]
K. Kim, S. Simpson, D. Hutchison, A. Scott
Transition to High-Speed Networks — SuperJANET Experience
In proc. Twelfth International Conference On Computer Communication, Seoul, August 1995.

[LaLiberte 95]
Daniel LaLiberte, Michael Shapiro
The Path URN Specification
IETF Internet Draft `draft-ietf-uri-urn-path-01.txt`; expires 17th January 1996.
`http://union.ncsa.uiuc.edu/~liberte/www/path.html`

[Le Gall 91]
Didier Le Gall
MPEG: A video compression standard for multimedia applications
CACM, vol. 34, no. 4, April 1991, pp. 46-58.

[Levergood 93]
Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, Lawrence C. Stewart
AudioFile: A Network-Transparent System for Distributed Audio Applications
In. Proc. USENIX Summer Conference, June 1993.

[Liou 91]
Ming Liou
Overview of the p×64kbit/s Video Coding Standard
CACM, vol. 34, no. 4, April 1991, pp. 59-63.

[Mauthe 94]
A. Mauthe, D. Hutchison, G. Coulson, S. Namuye
From requirements to services: group communication support for distributed multimedia systems
Proc. 2nd Int. Workshop, IWACA'94, Heidelberg (Berlin: Springer), pp. 266-77.

[Mauthe 95]
A. Mauthe, G. Coulson, D. Hutchison, S. Namuye
Group Support in Multimedia Communication Systems
In Proc. Second International COST 237 Conference on Communications, Copenhagen, Denmark, November 1995, pp. 1-18.

[Mauthe 96]
A. Mauthe, D. Hutchison, G. Coulson, S. Namuye
Multimedia group communications: towards new services
Distrib. Syst. Engng 3 (1996) pp. 197-210.

[Mauthe 97]
A. Mauthe, N. Yeadon, D. Hutchison, F. García
Quality of Service Filtering in an Internet Environment
Internal Report MPG-97-25, Lancaster.

[McCanne 95]
S. McCanne, V. Jacobson
vic: A flexible framework for packet video
In Proc. of ACM Multimedia '95, November 1995.

[McDysan 94]
David E. McDysan, Darren L. Spohn
ATM Theory and Application
McGraw-Hill Series on Computer Communication, (ISBN 0-07-060362-6?) 1994.

[Namuye 96a]
S. Namuye, S. Simpson, D. Hutchison
An Architecture for Flexible Multimedia Group Management Services
Proc. of Third Communication Networks Symposium, Manchester Metropolitan University, 1996, pp. ??.

[Namuye 96b]
S. Namuye
An Architecture for Flexible Multimedia Group Management Services
Ph.D. Thesis, Lancaster University, 1996.

[Pasquale 92]
J. Pasquale, G. Polyzos, E. Anderson, V. Kompella
The Multimedia Multicast Channel
Proc. of Third International Workshop on Network and Operating Systems Support for Digital Audio and Video, San Diego, November 1992, pp. 269-278.

[Pasquale 93]
J. Pasquale, G. Polyzos, G. Anderson, V. Kompella
Filter Propagation in Dissemination Trees: Trading off Bandwidth and Processing in Continuous Media Networks
Proc. NOSSDAV'93, Lancaster, November 1993, pp. 259-268.

[Rezende 95]
J.F. Rezende, A. Mauthe, S. Fdida, D. Hutchison
M-Connection Service: A Multicast Service for Distributed Multimedia Applications
Proceedings of Multimedia Transport and Teleservices, International COST237 Workshop, Copenhagen, Denmark, November 1995.

[Sasse 95]
M.A. Sasse, R. Bennett
Multimedia Conferencing over the Internet — The MICE Project
Library & Information Systems Briefings, Issue 58, March 1995.

[Shulzrinne 96]
H. Shulzrinne, S. Casner, R. Frederick, V. Jacobson
RTP: A Transport Protocol for Real-Time Applications
Network Working Group RFC 1889, January 1996.

[Sollins 94]
K. Sollins, L. Masinter
Functional Requirements of Uniform Resource Names
RFC 1737, MIT/LCS, Xerox Corp, December 1994.

[Strayer 95]
Tim Strayer (Ed-in-Chief) et al
XTP 4.0 Specification
XTP 95-20, Santa Barbara, 1995.

[Syzperski 93a]
Clemens Syzperski, Giorgio Ventre
A Characterization of Multi-Party Interactive Multimedia Applications
Technical Report TR-93-006, The Tenet Group, International Computer Science Institute & Computer Science Division, Department of EECS, University of California at Berkeley.

[Szyperski 93b]
Clemens Syzperski, Giorgio Ventre
Efficient Multicasting for Interactive Multimedia Applications
Technical Report TR-93-017, The Tenet Group, International Computer Science Institute & Computer Science Division, Department of EECS, University of California at Berkeley.

[Szyperski 94]
C. Szyperski, G. Ventre
Efficient Support for Multiparty Communication
In Proc. of Multimedia Transport and Teleservices, International COST237 Workshop, Vienna, Austria, Nov. 1994.

[Topolcic 90]
C. Topolcic (Ed.)
Experimental Internet Stream Protocol, Version 2 (ST-II)
Network Working Group RFC 1190, October 1990.

[Toutain 95]
François Toutain, Laurent Toutain
Network Support for Multimedia Communications Using Distributed Media Scaling
Proc. of Second International COST 237 Workshop, Copenhagen, November 1995, pp. 138-157.

[Turletti 93]
Thierry Turletti
H.261 Software Codec for Videoconferencing Over The Internet
INRIA Research Report N°1834.

[Vin 92]
Harrick M.Vin, P.Venkat Rangan
System Support for Computer Mediated Multimedia Collaborations
In Proc. CSCW92, November 92.

[Wallace 91]
G. Wallace
The JPEG Still Picture Compression Standard
Communications of the ACM, Vol. 34, no. 4, April 1991, pp. 30-44.

[Watson 97]
A. Watson & M. A. Sasse
Multimedia Conferencing via Multicast: Determining the Quality of Service Required by the End-User
Proceedings of the 1997 International Workshop on Audio-Visual Services over Packet Networks (AVSPN), 15-16 September 1997, Aberdeen.

[Wilde 96]
E. Wilde, M. Nanduri, B. Plattner
A Transport-Independent Component for a Group Session Management Service in Group Communication Platforms
Proc. of European Conference on Multimedia Applications, Services and Techniques, Louvain-la-Neuve, Belgium, 1996 (pages?).

[Wilde 96a]
E. Wilde
Specification of GMS System Protocol (GSP) Version 1.0
Technical Report TIK-Report No. 19, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, September 1996.

[Wilde 96b]
E. Wilde
Specification of GMS Access Protocol (GAP) Version 1.0
Technical Report TIK-Report No. 15, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zürich, March 1996.

[Wilde 97]
E. Wilde
Group and Session Management for Collaborative Applications
Doctor of Technical Sciences dissertation, Swiss Federal Institute of Technology, Zürich, 1997.

[Yeadon 96a]
Nicholas Yeadon, Andreas Mauthe, Francisco García, David Hutchison
QoS Filters: Addressing the Heterogeneity Gap
Proc. European Workshop on Interactive Distributed Multimedia Systems and Services, Berlin, Germany, March 1996. Lancaster Internal Report MPG-96-09.

[Yeadon 96b]
Nicholas Yeadon
Quality of Service Filters for Multimedia Communications
Ph.D. Thesis, Lancaster University, May 1996

[Zhang 93]
L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala
RSVP: A New Resource ReSerVation Protocol
IEEE Network, vol. 7, Sept. 1993, pp. 8-18.