Technical Report

UCAM-CL-TR-220
ISSN 1476-2986

Number 220

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

A distributed architecture for multimedia communication systems

Cosmos Andrea Nicolaou

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 1990 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Christ's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

Technological advances in digital communications and in personal computer workstations are beginning to allow the generation, communication and presentation of multiple information media simultaneously. In particular, the ability to support real-time voice and video makes a new range of advanced and highly interactive multimedia applications possible. These applications are not restricted to the computer industry, but extend to other technologically intensive industries which have some form of multimedia communication requirement. Such industries include medicine, conferencing, teaching, broadcasting, publishing and printing. Each of these application areas has its own particular set of requirements and makes corresponding demands on the computer systems used.

Such a wide range of application areas leads to a correspondingly large and diverse set of requirements of the systems used to implement them. In addition, the real-time nature of voice, and especially video, place heavy demands on the underlying systems. Many of these requirements and demands are not met by existing computer communication systems. This is due to the fact that the architectural models used to design and implement these systems were constructed before the technological advances making multimedia communication possible took place. As a result, existing multimedia systems have tended to concentrate on either low level implementation issues (e.g. communication networks and protocols) or on a single restricted application area, without paying any regard to their respective problems and requirements. The inevitable consequence is that there is a mismatch between the functions provided at the lower levels and those actually required by higher level applications.

This dissertation presents an attempt to overcome these problems by defining a new architecture for multimedia communication systems which recognises and supports a wide range of application requirements, in addition to satisfying the requirements made by the information media themselves. A thorough survey of existing multimedia systems was conducted in order to identify and understand the requirements made by both applications and information media and led to the formulation of a set of design principles. In recognition of the fact that any multimedia communication system is inherently distributed in nature, the architecture is presented as an extension of existing distributed systems.

The resulting architecture is called the Integrated Multimedia Applications Communication architecture (IMAC) and a prototype implementation of IMAC has been constructed and used to evaluate the utility and feasibility of the architecture and to identify its strength and weaknesses.

Contents

List of Tables

List of Figures

Glossary

The number of the page on which the term is introduced appears in parentheses after each term.

AAL ATM Adaptation Layer (14)

ADPCM Adaptive Differential Pulse Code Modulation (50)

ANSA Advanced Networked Systems Architecture (4)

ANSI American National Standards Institute (13)

ATM Asynchronous Transfer Mode (13)

B-ISDN Broadband Integrated Services Digital Network (2)

CCITT Comiteé Consultatif International Télégraphique et Téléphonique (11)

CCS Common Channel Signalling (12)

CFR Cambridge Fast Ring (22)

CIT Computer Integrated Telephony (13)

CMEX Continuous Media Extension to X (40)

CSCW Computer Supported Cooperative Working (33)

DARPA Defense Advanced Research Projects Agency (16)

DBP Dual Bus Protocol (22)

DCS Distributed Computing System (2)

DPC Distributed Processing Component (2)

DQDB Distributed Queue Dual Bus (22)

DRI Defense Research Internet (22)

EMCE Experimental Multimedia Conferencing Environment (35)

FDDI Fiber Distributed Data Interface (21)

FEC Forward Error Correction (52)

FTAM File Transfer and Access Management (16)

GEX Group EXecution protocol (176)

GOSIP Government OSI Profile (17)

HDTV High Definition TeleVision (7)

IDCM Integrated Digital Continuous Media (40)

IDL Interface Definition Language (108)

IMAC Integrated Multimedia Applications Communication architecture (v, 1)

IMC Information Media Component (2)

IRM Integrated Reference Model (25)

ISDN Integrated Services Digital Network (2)

ISLAND Integrated Services Local Area Network Development (29)

ISO International Standards Organization (15)

LCRI Local Channel Resource Interface (117)

LCR Local Channel Resource (115)

LSF Logical Synchronisation Frame (78)

LVC Lightweight Virtual Circuit (59)

MICE Modular Integrated Communications Environment (27)

MMCS MultiMedia Communication System (2)

MMDT MultiMedia DeskTop (2)

MPS Message Passing Service (176)

MSN Multi-Service Network (23)

ODP Open Distributed Processing (16)

OSF Open Software Foundation (171)

OSI Open Systems Interconnection (15)

PCM Pulse Code Modulation (50)

PSF Physical Synchronisation Frame (78)

PTM Packet Transfer Mode (14)

QoS Quality of Service (54)

QPSX Queued Packet and Synchronous eXchange (22)

RAVI Representation for Audio/Visual Interactive Applications (17)

REX Remote EXecution protocol (176)

RFC Request For Comments (23)

SDM Space Division Multiplexing (9)

STM Synchronous Transfer Mode (14)

ST STream protocol (22)

TCA Traffic Control Architecture (25)

TDM Time Division Multiplexing (10)

TWBNet	The Terrestrial WideBand Network (22)
UIC	User Interface Component (3)
UIMS	User Interface Management System (60)
UKC	University of Kent at Cantebury (32)
VM	Virtual Machine (175)
VTP	Virtual Terminal Protocol (16)

1

Introduction

This dissertation is concerned with the provision of an environment which facilitates the construction of advanced and highly interactive multimedia applications. In other words, the goal is to make the writing of multimedia applications easier than is currently the case. This desire is motivated by the observation that little or no practical experience and insight exist into what the practical uses, benefits, and pitfalls of multimedia communication will be. The only way to overcome this problem is to build a number of diverse applications for experimentation and evaluation; for this experimental process to occur it is first necessary to make the construction of the experimental applications easier.

The term media is used in this dissertation to refer to a variety of information forms, including text, graphics, voice, still video images and full motion video. The real-time nature of media such as voice and video makes rigorous demands of any computer system managing and implementing them. Multimedia communication implies that multiple media may be used *simultaneously* during communication; therefore, any multimedia communication application must be able to handle multiple media streams simultaneously.

It is the real-time nature of voice and video, coupled with the need to handle multiple streams simultaneously which are the primary source of the problems posed by multimedia communication. Multimedia communications applications intrinsically require some form of distributed computing environment. The services provided by existing operating systems, communications protocols and distributed programming tools (e.g. remote procedure call) predate the advent of multimedia communication and provide insufficient and inappropriate services for supporting multimedia applications. The divergence between the services provided, and those required by applications, is set to increase as more diverse, advanced and highly interactive multimedia applications are constructed.

This dissertation presents a new architecture, the Integrated Multimedia Applications Communication architecture (IMAC), which has been designed to accommodate multimedia application requirements as well the problems posed by the presence of multiple information media and communication. IMAC provides a coherent framework within which to design and implement all system components and thus avoid functional mismatches in the future. The IMAC architecture can therefore be used to guide the design and implementation of the applications support environment as well as the applications themselves.

1.1 Multimedia Communication Systems

Advances in networking and workstation[1] technology are making multimedia applications possible and increasing the desire for, and expectations of, such applications.

Advances in local and wide area digital communications networks enable the efficient transmission of multimedia traffic. The Integrated Services Digital Network (ISDN) promises to provide ubiquitous, wide area, digital communication capabilities. Although the ISDN is primarily designed to handle voice and data traffic, the higher speed Broadband ISDN (B-ISDN) is intended to carry video in addition to voice and data. A variety of local and metropolitan area networks operating in the 10-100Mb/s range can already carry multimedia traffic. This ability to communicate multimedia information is making a new range of information processing applications possible, and is stimulating the increasing use of computers in other technologically intensive industries which have some form of multimedia communication requirement.

The increasing power of workstations means that in addition to supporting a greater variety of advanced and highly interactive applications, they are increasingly able to support multimedia. However, the data and switching rates required by real-time video communication are still beyond the capability of most workstations. Therefore, such workstations can, at best, be used to control, rather than switch, video streams. Although less demanding media, such as voice, can be easily switched using current workstations, it may not be possible to transport the voice between workstations using existing protocols and networks such as TCP/IP and Ethernet. As a result a hybrid approach is often taken, whereby a workstation is used to control external hardware implementing media streams which it cannot directly implement itself. This leads to the idea of a "multimedia desktop" or MMDT. An MMDT is a collection of hardware and software components which implement and provide access to multimedia application and communication facilities. The partitioning between the media implemented within and without the workstation is constantly shifting and in the future it may be possible to implement all media within the workstation. However there will always be some delay between the introduction of new media and their integration into workstations; therefore the notion of an MMDT seems likely to persist.

The use of multimedia networks to interconnect MMDT's leads to the concept of a *Multimedia Communication System* or MMCS. An MMCS can be considered as being an extension of the traditional notion of a distributed computing system (DCS) to include the ability to generate, manipulate, communicate and present multiple, possibly real-time, media simultaneously. The next section discusses the components of an MMCS.

1.1.1 System Components

A multimedia communication system is composed of five functional components:

Information Media Component (IMC): implements the basic mechanisms for the generation and presentation of information media. Simple examples include a digital camera or microphone; a more complex example is a frame buffer with video input.

Communications Component: transports multiple information media from one place to another. It includes both the physical network and low-level access protocols in addition to higher level communications protocols. However, as discussed below, the boundary between the Communications component and DPC is often blurred.

[1] The term workstation is used to refer to a personal computer which supports a powerful multi-tasking operating system and programming environment.

Distributed Processing Component (DPC): provides the base set of services required by an application for distributed processing. This will usually be a superset of the functionality required for stand-alone processing. The provision of storage is also considered to lie within this component. The work of ANSA [**ANSA89b**] and the ISO Open Distributed Processing (ODP) standardisation effort represent significant efforts to define, design and implement such services.

Application Component: responsible for controlling and coordinating the lower level mechanisms to bring about some meaningful user interaction and communication.

User Interface Component (UIC); the interface between the application and the users of that application. The application and user interface components are separated in order to allow a single application to have a number of user interfaces. The application is concerned solely with functionality whilst the user interface deals with human computer interaction.

The partitioning of communication protocol functionality between the Communications and Distributed Processing components is not straightforward; in particular the lower levels of the OSI protocol stack are part of the Communications Component whilst the session and presentation layers are best considered part of the DPC. This separation is likely to vary for different protocol architectures and distributed operating systems. As a rough guideline, functions related solely to communication are placed in the Communications Component, whereas functions which have an impact on other system components are placed in the DPC. Typically the interface to communications protocols and associated resource management are considered to be part of the DPC.

It is not clear that storage should be entirely subsumed into the DPC, especially when the demands made on it by voice and video are so great. However, storage is a service required by applications and as such can be considered to lie in the DPC.

1.1.2 Functional Integration

There are two principal ways in which multimedia can be integrated into a DCS. The inclusion of additional media within an individual component is *Multimedia Integration*, whilst *Functional Integration* is concerned with the interoperation of system components. The research survey in chapter 3 shows that whilst considerable progress has been made on multimedia integration for one or two components in isolation, little attention has been paid to the need for functional integration. This can be seen by the fact that many projects have concentrated on multimedia integration for the IMC and Communication components alone (i.e. communications systems) or on the Application and User Interface components (i.e. stand-alone systems).

The lack of functional integration is the biggest problem facing the designer of an MMCS; currently there is a mismatch between the functions expected of one component by another and the functions actually provided. This leads to the situation where advances in one component are either stymied by previously unforeseen deficiencies in another, or are simply ignored and not used by the other components. The lack of functional integration makes writing multimedia applications inordinately difficult and in many cases the application writer is forced to implement or re-implement DPC level functions. This has led to the proliferation of ad hoc solutions to common problems. Clearly the goal of making application writing easier translates directly to one of *increasing* functional integration.

1.2 An Architectural Approach

In order to increase functional integration it is necessary to understand the requirements and demands system components make of each other and also to appreciate the relationships between components. This approach is often referred to as *architectural* and can be considered to define system building blocks and interfaces, design rules, guidelines and recipes for how to design specific systems. An additional benefit of an architecture is that it provides a framework within which related work can be evaluated and incorporated, and thus ensure that good use is made of such work.

The thesis of this dissertation is that such an architectural approach is effective in increasing functional integration and that this reduces the difficulties which are currently encountered in constructing multimedia applications. Moreover, sufficient research has been carried out to allow the identification of a set of common architectural principles and thus enable such an architectural approach to be taken.

1.3 The IMAC Architecture

As already noted, an MMCS can be viewed as a logical extension of a DCS to incorporate multimedia. Extending a DCS in this manner offers a number of potential advantages:

- a DCS provides a comfortable environment for writing distributed applications.
- reuse of a large amount of existing knowledge, infrastructure and software.
- easing the interoperation of multimedia applications with current and future distributed applications.

To ensure that these potential advantages are realised IMAC is based on an existing DCS architecture, namely the Advanced Networked Systems Architecture (ANSA), which is described in [**ANSA89b**][**ANSA89a**] and is briefly summarised in appendix A.

ANSA is a general architecture for distributed systems; IMAC is a specialised instance of ANSA catering specifically for multimedia. The implementation of IMAC is an extension of the ANSA Testbench [**ANSA90b**], which is itself a particular implementation of the architecture. In the long term the architectural concepts identified by IMAC will be incorporated into the ANSA architecture itself.

1.4 Outline

The approach taken to the work presented in this dissertation was to conduct a detailed survey of background and related work. This enabled the problems, requirements, inter-relationships and design principles of a Multimedia Communication System to be understood. This information was then used to guide the design of IMAC and the prototype implementation used to evaluate the utility and feasibility, and to identify the strengths and weaknesses of the architecture.

Chapters 2 and 3 present background and related work respectively, and chapter 4 discusses problems and requirements. The IMAC architecture is presented in chapter 5; the architecture is

informally defined and its design justified. The prototype implementation is described in chapter 7; chapter 8 describes how a complete application can be constructed using this prototype. Chapter 9 presents an evaluation of IMAC and its prototype implementation.

2

Background

This chapter begins by describing what is meant by the terms *multimedia*, *multimedia communication*, *multimedia integration*, *functional integration* and *multimedia desktop*. This is followed by a description of the background to this dissertation, in particular relevant standards work.

2.1 Multimedia

The term media is used to refer to a variety of information forms including, text, graphics, structured data (e.g. spreadsheet), voice, still video images and full motion video. That is, the unqualified term media is used to refer to information media as opposed to transmission media. Voice and video are used in a generic sense to refer to all encodings and standards used for these media, e.g. video may be of PAL television or High Definition TV (HDTV) standard. The term, *continuous* media, is often used to refer to real-time information media such as voice and video. This dissertation is primarily concerned with continuous media and wherever the term multimedia is used it implicitly includes continuous media. Section 4.2.1 discusses the implications of continuous media.

2.2 Multimedia Communication

Within this dissertation multimedia communication refers to interaction which explicitly allows for the *simultaneous* use of multiple information media. This is distinct from the ability to communicate using multiple media where only a single medium may be in use at any instant in time. These two interpretations are often confused. Wherever such confusion is possible, or emphasis of one over the other is required, the first usage will be qualified as being *true* multimedia communication.

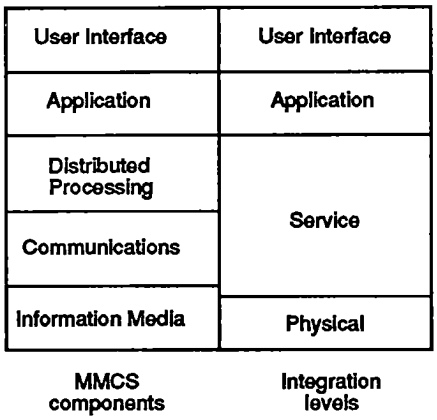

Figure 2.1: System Component and Integration Level Relationships

2.3 Multimedia Integration

The presence of multimedia places significant demands on each system component. An increase in multimedia integration for a given component implies that it is better able to meet the demands of some new media or multiple existing media. Multimedia integration can occur at the following four levels.

Physical Level integration: the multiplexing of multimedia traffic over a single piece of hardware or a single communication channel.

Service or System Level integration: the provision of a common service interface to the application for access to multimedia communication facilities and for communication between its constituent parts.

Application Level integration: the design and implementation of applications which can effectively manage, manipulate and communicate multimedia data.

User Interface Level integration: the presentation of multiple types of media to a human user.

The relationship between these integration levels and the MMCS components introduced in section 1.1.1 is illustrated in figure 2.1. Physical level integration occurs within the Information Media Component (IMC). Service level integration is concerned with the interfaces to the IMC, Communications Component and DPC, and is therefore shown as spanning the DPC and Communications Component to touch the upper layers of the IMC. The Application and User Interface levels map directly to their respective MMCS components.

Within the Physical Level it is possible to identify two independent types of integration, namely *hardware* and *network* integration. Independence means that a given level of integration of one type does not imply a corresponding level in the other.

2.3.1 Hardware Integration

MMDT functional components may be implemented on a single hardware component or be spread across several such components (see section 2.5). The fewer the number of hardware components,

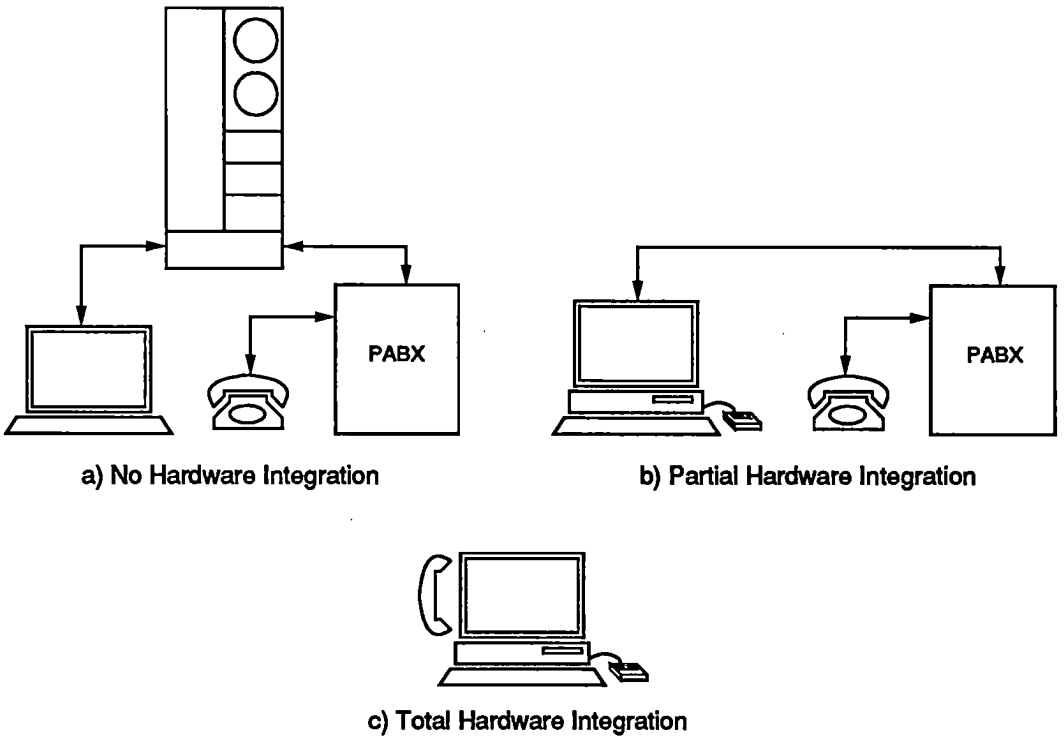

Figure 2.2: Degrees of Hardware Integration

the greater the degree of integration. Thus, there is a spectrum of hardware integration ranging from completely unintegrated hardware to fully integrated hardware. Figure 2.2 gives three simple examples showing the increasing integration of voice and data communication hardware. The first configuration uses a terminal, a telephone handset connected to a computer controlled PABX and a minicomputer running the application which controls the PABX and terminal. Each functional component of the MMDT is implemented using a different piece of hardware. The second configuration uses a workstation to implement both the user interface and the application components, thus removing the need for a minicomputer. The final configuration is an *integrated voice and data workstation* which incorporates the telephone handset directly. Increasing hardware integration offers potential reductions in cost and increased performance capable of supporting more advanced applications.

This integration path will also be followed for video communication as workstation performance increases and allows the implementation of video directly. So far, only partial hardware integration has been achieved with MMDT's providing video communication; Pandora (section 3.7.4) and Palantir (section 3.7.3) are examples of such systems.

2.3.2 Network Integration

Several networks are described as integrated or multi-service, because they can support multiple types of traffic using the same infrastructure. As a result of early work on the integration of voice and data within the ISDN [**Gerla84**] three levels of integration have been identified, which are distinguished by the multiplexing technique used within each network component. Two principal multiplexing techniques are used: the first is Space Division Multiplexing (SDM) which uses a

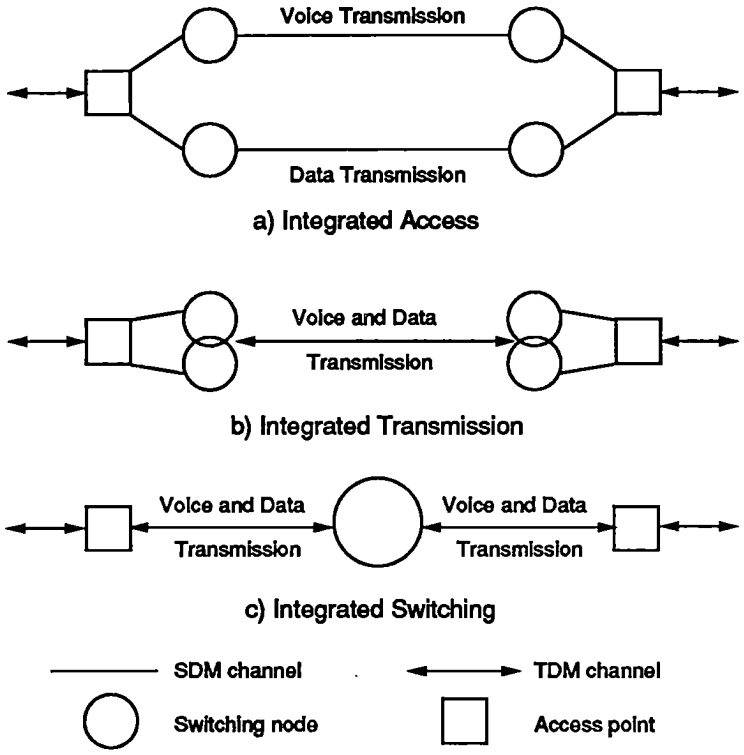

Figure 2.3: Levels of Network Integration

separate channel between communication endpoints. The second is Time Division Multiplexing (TDM) which shares a single channel between multiple endpoints over time; that is, each end point is allowed access to the channel for some period of time before the next end point is allowed access for some period and so on.

The three levels of integration identified are illustrated in figure 2.3:

Integrated Access: provides a single access interface for both voice and data traffic. Within the network this traffic may be carried over separate physical networks each of which has its own switching mechanism. User access is multiplexed using some form of TDM, whilst transmission and switching are multiplexed using SDM.

Integrated Transmission: uses a single physical network to carry both voice and data traffic, however the switching may still be handled separately. Here TDM is used for access and transmission, but SDM is used for switching.

Integrated Switching: uses a single switching mechanism to handle all types of traffic.

Increasing network integration can therefore be viewed as the increasing use of Time Division Multiplexing to multiplex different media types within the network. These levels of integration can be easily generalised to include other media in addition to voice and data. Note that, in general, these levels of integration are independent of one another. For instance, it is possible to have integrated transmission and switching without integrated access.

2.4 Functional Integration

Functional integration is concerned with the *co-operation* of system components to achieve some composite goal. The degree of integration is the degree to which system components co-operate to meet the goals set for the system as a whole. To increase functional integration it is necessary to devise a common abstraction which can be implemented across all components and thus provide a framework within which each component may identify the functions it provides and the functions it requires of other components.

2.5 Multimedia Desktop

Given the varying degrees of hardware and network integration possible it is dangerous to choose a single point in the integration spectrum and design applications specifically tailored to that point. The danger lies in designing a system which may be difficult or impossible to port to a new configuration or even to interwork with different configurations. The notion of a *Multimedia Desktop* is useful for avoiding such a pitfall. An MMDT represents a set of functions which can be accessed by application software independently of the physical and network configuration used to implement them. In particular, it must support the management of shared and distributed resources.

2.6 ISDN and Broadband ISDN

ISDN is the result of an international effort to produce a comprehensive set of standards for a global digital communications network capable of carrying multiple traffic types. This standardisation is being coordinated by the CCITT and has led to the I-Series Recommendations. The ISDN is designed to support a range of voice and non-voice (i.e. data) services within a single network. This integration is provided at the network access level, that is, a single point of access is defined for all traffic types supported. The CCITT refer to this access level integration as service integration; this is not the same as the notion of service integration introduced when discussing multimedia integration within an MMCS. The I.400 User-Network Interfaces recommendations deal with the definition of these access points.

2.6.1 User Network Interfaces

By definition an ISDN is recognised by the functionality it provides at its access point and not in any way by its internal architecture or implementation. The ISDN recommendations concentrate on the definition of these access points. A user network interface consists of a number of fixed bandwidth communication channels presented as an isochronous series of time slots; i.e. synchronous time division multiplexing. Three[1] types of channel are defined as follows:

Bearer (B) Channels: provide 64Kbits/s and are targeted at digital voice communication, they may also be used for data.

Higher (H) Rate Channels: higher bandwidth B channels offering 384Kbits/s, 1536Kbits/s and 1920Kbits/s; these are called H0, H11, H12 channels respectively.

[1] The E channel is not mentioned in this description as it is rarely used.

Data (D) Channels: either 16 or 64Kbits/s and are often called *signaling* channels since their intended use is for the management of connections on associated bearer channels.

The signaling protocol used on the D channel is defined in I.451 and is related to the existing CCITT Signaling System No. 7 defined in the Q.700 series recommendations.

Different channel combinations are used to offer different ISDN rates as follows:

Basic Rate: 2xB channels and a single 16Kbits/s D channel.

Primary Rate (2.048Mbits/s): 30xB channels and a single 64Kbits/s D channel; this rate is used within Europe.

Primary Rate (1.544Mbits/s): 22xB channels and a single 64Kbits/s D channel; this rate is used within the U.S.A.

The use of separate channels for call management and routing (control) and actual voice traffic (data) is generally referred to as *out-of-band signaling.* Within the context of the ISDN and circuit switched networks in general, such out-of-band signaling is referred to as Common Channel Signaling (CCS). Allowing user access to CCS, via the D channel, promises to bring some of the power and flexibility present in computer data communications to the wide area telephone network.

2.6.2 Service Capabilities

Two broad categories of service are defined by the CCITT (I.112):

Bearer services: provide the capability for the transmission of signals between user network interfaces. Bearer services are either restricted, in which case the data communicated may be modified in transit, or unrestricted, in which case no such modification occurs. Bearer services are defined for restricted circuit-mode 64Kbits/s speech and unrestricted circuit-mode 64Kbits/s which can support speech, X25 and other information streams which may be multiplexed onto the channel. A packet-mode bearer service is defined, as are virtual call and permanent virtual circuit services which allow for packetised data transfer over a virtual circuit.

Teleservices: provide a complete service capability, including terminal equipment functions, for communication between users according to protocols established by agreement between administrators. Teleservices use bearer services to communicate data and in addition provide higher level (OSI Layers 4-7) functions. Teleservices are less well developed than bearer services but are likely to include services such as facsimile, videotex and electronic mail.

It is possible to define supplementary services which in some way extend the functions provided by bearer and teleservices; these supplementary services can only be offered in association with their base services.

All services within the ISDN are characterised using the attribute based model described in I.130. There is a set of generic attributes and a set of attribute values which can be assigned to these generic attributes. This approach has the potential drawback that only services which can be characterised by the attributes defined in I.130 can ever be implemented. A specific set of services have been targeted for standardisation.

The ISDN is currently being implemented within the existing telephone network infrastructure with a gradual move to transmission level integration as more digital communication capability

becomes available. The emphasis of the ISDN is on improving telephony based services rather than on general purpose multimedia services; the term "Computer Integrated Telephony" (CIT) is often used to describe the ISDN. However the wide area digital communication offered by the ISDN is stimulating the desire for other multimedia services to be offered in the future.

2.6.3 Broadband ISDN

As a result of advances in opto-electronics it is possible to build very high speed networks running at hundreds of Mbits/s. In response to the this the CCITT and ANSI are attempting to standardise these networks as the Broadband ISDN (B-ISDN). The B-ISDN promises to support video to the same degree as the ISDN currently supports voice, with proposed bandwidths starting at 150Mbits/s.

In recognition of the limitations of the current telephone network it is anticipated that the B-ISDN will be entirely implemented using new communications technology which can support the wide variety of traffic required. The goal is to provide integrated switching and transmission of all traffic types. The mechanism targeted for achieving this integration is Asynchronous Transfer Mode (ATM) which is discussed in section 2.7.

The B-ISDN uses many of the principles defined for ISDN, including out-of-band signaling and attribute based service characterisation. The attribute set has been extended to more fully characterise video. Also two service classes have been identified:

Interactive services: include real-time communication, messaging and retrieval services.

Distributive services: deal with the distribution of information, in either a broadcast fashion without user presentation control (e.g. TV and radio) or as a multicast with presentation control (e.g. teletex).

Both service classes allow for *true* multimedia communication. As for ISDN a set of candidate services have been identified for standardisation.

The B-ISDN explicitly recognises the need for multimedia communication and advocates a "strongly structured approach" to such communication to ensure:

- flexibility for the user.
- simplicity for the network operator.
- control of interworking situations.
- commonality of terminal and network equipment elements.

It is worth noting that the first two are almost certainly in conflict and that strict adherence to the last two is likely to stifle innovation.

The work on B-ISDN is at a very early stage, especially with regard to multimedia, with most attention being paid to the network itself (i.e. application of ATM techniques).

2.7 ATM

The term ATM refers to a very general class of networks and is not restricted to the standards being drawn up by the CCITT and ANSI for specific implementations of ATM networks. Many existing and future networks will justifiably be described as ATM even though they do not conform to CCITT recommendations.

ATM has gained favour for implementing the B-ISDN because it represents a compromise between the Synchronous Transfer Mode (STM) used within the telephone network for digital voice transmission and Packet Transfer Mode (PTM) used for computer data communications. This compromise offers the following potential advantages:

- fine granularity bandwidth sharing.
- low and bounded delay.
- low and predictable variation in delay (jitter).
- ability to efficiently carry bursty and variable rate traffic.

The disadvantages centre on the technical difficulties encountered in building such ATM networks. In particular extensive hardware implementation is required to achieve the data and switching rates required. This forces the lower levels of the protocol stack to be as simple as possible and therefore amenable to hardware implementation, which in turn means that functions such as internetworking and routing must be pushed up to higher levels in the protocol stack. McAuley discusses these issues in more detail [**McAuley89**].

Current ATM networks can be recognised by the presence of three principal features:

- fixed cell size.[2]
- asynchronous access.
- bounded access time.

The CCITT and ANSI describe ATM as being "connection oriented", however there is currently a great deal of debate as to exactly what this means. The motivation behind this appears to be the desire to pre-allocate and effectively manage network resources at connection setup time. Guarantees can then be made with regard to per-connection quality of service which can be met for the duration of the connection. It is not yet clear if such a reservation mechanism will be defined as a core part of the standards currently being drafted for B-ISDN. The term *lightweight virtual circuit* is used throughout this dissertation, as it is in [**McAuley89**][**Leslie83**] to represent an end-to-end connection establishment without any implicit guarantee of *reliability*. Lightweight virtual circuits do not in anyway preclude the provision of a connectionless style service on top of ATM.

The service provided by ATM is intended to be useful to all traffic types, and as such must not be parametrised for any single service. An ATM Adaptation Layer (AAL) is used to provide a particular service, such as voice, video or data communication over the underlying ATM bearer service.

[2] The CCITT has defined the ATM cell size to be 53 bytes; 5 bytes of header and 48 bytes of user data.

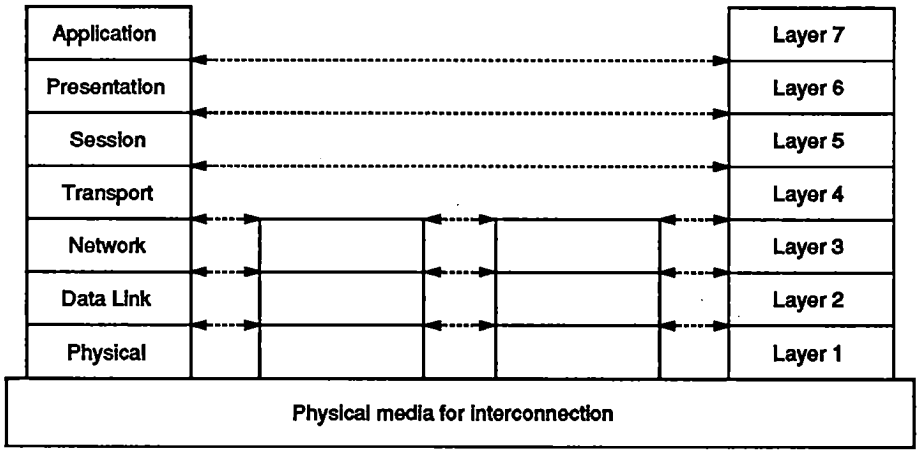

Figure 2.4: The OSI Reference Model

2.8 OSI Reference Model

The ISO Open Systems Interconnection (OSI) Reference Model defines seven layers as illustrated in figure 2.4. The primary aim of this model as defined by [**Day83**] is "to provide a framework for coordinating the development of OSI standards". These standards form the basis for the interconnection of heterogeneous computer systems to create *open systems*. The ultimate goal is that by fully specifying the function of each layer, and then by standardising these specifications it will be possible for a variety of manufacturers to produce conforming layer implementations which will interwork with each other.

The inherent danger in this approach is that if the standardisation takes too long then by the time conforming implementations come into existence the design assumptions and tenets made whilst defining the layer functions will have become invalid. Also, the desire for interconnection at all costs leads to a *lowest common denominator* approach to defining the layer functions, which may in the long term lead to functionally deficient standards. Similarly the fear of getting it wrong has led to vast array of complex standards offering a great number of options. The sheer number of standards (over 200) has forced procurement agencies to develop OSI profiles which specify a particular set of options to use for each layer, thus taking a vertical slice or plane through the OSI stack. The complexity of the options available means that there is no guarantee that one profile will interwork with one another.

The current ISO OSI standards are found lacking in a number of respects when applied to multimedia communication. These are a result of the differing and previously unforeseen requirements of multimedia communication:

- the insistence that the transport service provided be a reliable one; reliability is often prohibitively expensive for real-time multimedia communication.
- the presence of multiplexing in six out of seven layers;[3] multiplexing is a major source of jitter and performance loss.
- the sheer complexity of the protocols means that they are unlikely to provide the high levels of performance required for multimedia communication.

[3]The exception being the presentation layer, but for *architectural consistency* even it allows for demultiplexing using address selectors

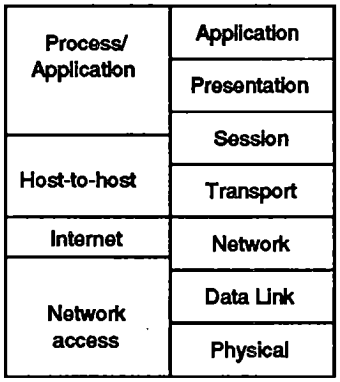

Figure 2.5: Internet Protocol Levels

Other deficiencies arise due to the delay in accommodating the technology required for multimedia communication; the primary example being the incorporation of ISDN, B-ISDN and ATM.

The standardisation process has been through two main phases; the first dealt with the lower four layers and lead to the full definition of the transport service. Whilst the second has been concerned with the session, presentation and application layers. As for ISDN, a very specific set of applications have been identified and targeted for standardisation, including:

FTAM File Transfer and Access Mechanism.

X.400 Electronic Mail.

VTP Virtual Terminal Protocol.

OSI-TP Transaction Processing.

Standardising a set of applications clearly cannot lead to a variety of diverse and powerful applications, but rather to a base set of essential tools. In recognition of this ISO have set up a new working group, ISO/IEC JTC1/SC21/WG7 for Open Distributed Processing (ODP) to investigate application requirements and to standardise a Support Environment for ODP (SE-ODP). ODP is discussed in section 2.11.

2.9 DARPA Internet

Clark [**Clark88**] describes the evolutionary approach taken with the DARPA Internet, whereby the design, implementation and evaluation of protocols took place before standards were set. The results of the evaluation phase often forced a redesign, and reimplementation. This philosophy is in stark contrast to that of OSI.

The Internet architecture defines four levels of protocol. These protocols along with their relationships to the OSI layers are shown in figure 2.5. Internetworking was a primary goal and is implemented within the network level by the IP protocol. A range of transport, or host-to-host, protocols may be used over IP, the principal ones being UDP which provides an unreliable datagram service, and a reliable byte stream protocol TCP. There is no requirement that TCP be layered on UDP, in fact TCP uses IP directly. The absence of strict layering leads to a hierarchical

architecture, which allows for redundant layers to be bypassed, thus offering a considerable increase in flexibility.

As with OSI a standard set of applications have been constructed, the principal ones being FTP (File Transfer Protocol), SMTP (Simple Mail Transport Protocol) and Telnet (remote login). More applications have been constructed using the Internet protocols than ISO, simply because more Internet implementations exist. In fact the vast majority of UNIX systems offer the Internet Protocols as standard.

The Internet TCP/IP protocol suite provides much of the same functionality as that being standardised by ISO; indeed DARPA intend to adopt the U.S. Government OSI Profile (GOSIP) as an eventual replacement for the TCP/IP protocol suite. The U.S. GOSIP specifies a connectionless network service, whereas European and UK GOSIP's specify a connection oriented network service. It is not immediately obvious that the two profiles will be able to interwork efficiently.

2.10 RAVI

RAVI [**Oguet90**] is a proposed standard for the Representation of so-called Audio/Visual Interactive applications or AVI's; hence the acronym RAVI. This work is of interest because it is aimed at producing an application standard within the OSI framework and can be considered as extending OSI to provide application support. The anticipated application areas for RAVI include multimedia computer-assisted training, public information retrieval and transaction services as exemplified by tele-shopping.

The RAVI representation allows for the interchange of applications components themselves in addition to the data these applications manipulate and access. It consists of an interchange format for the data and a "formulation" describing the application body. This scheme allows for the easy propagation of application and information updates and is well suited to a commercial environment in which multiple organisations access the same applications and data. For instance RAVI provides a uniform means for television manufacturers to supply computer-assisted training applications for the maintenance of their televisions to a number of different servicing companies.

2.11 ODP

The ODP effort has two major goals, the first is concerned with establishing a single, consistent framework within which to express ODP requirements. The second is the standardisation of a support environment for these ODP requirements, within which ODP applications can be constructed. ODP is at a very early stage, however the ESPRIT ISA Project has been a major contributor to ODP and already has an architecture for ODP, namely the Advanced Network Systems Architecture (ANSA) and a prototype Support Environment for ODP, called the ANSA Testbench.

The Testbench is used as the base for the practical work carried out for this dissertation and is described along with the ANSA architecture in appendix A.

Neither ODP nor ANSA have yet made a serious attempt at supporting multimedia communication.

2.12 Distributed Computing Research

An early distributed system was the Cambridge Distributed Computing System [**Needham82**]. Since then work has been concentrated on Remote Procedure Call (RPC) [**Birrell84**] and in particular to the integration of RPC into existing languages [**Hamilton84**]. However RPC is often criticised for providing poor performance. Improved performance can be achieved with a highly optimised implementation as exemplified by Firefly RPC [**Schroeder89**]. Another possibility is to extend or modify the semantics of RPC to better match particular application requirements and thus to increase the performance achieved for a particular set of applications. For instance the *pipelined* or *streamed* RPC as advocated by Gifford [**Gifford88**], offers increased throughput for RPC's whose results are not immediately required and is particularly useful for communication with window or graphics systems. Gifford also identifies the need to synchronise operations on related pipes and provides an explicit, procedural, mechanism for doing so.

2.13 Summary

The ISDN provides access level integration for voice and data and is primarily aimed at telephony based applications. Within the context of such applications the ISDN has achieved a high degree of functional integration. The ISDN cannot be considered as providing true multimedia communication.

The B-ISDN is being designed to handle true multimedia communication from the outset and will provide integrated switching. However this work is at a very early stage. The out-of-band signaling techniques developed for ISDN are of general utility, as is allowing user access to signaling functions. These should ensure that the ISDN and B-ISDN are of use for building multimedia applications.

ATM is of fundamental importance to progress in multimedia communication since it provides the enabling communications technology. Its use in the B-ISDN will ensure that a wide area network capable of effectively handling multimedia communication will come into being over the next decade or so. Therefore any MMCS architecture must address the issues raised by ATM and allow for the effective exploitation of the advantages offered.

The main benefit offered by ISDN, B-ISDN and ATM is that of increased multimedia integration at the physical level. Targeting applications for standardisation is unlikely to encourage the experimentation required for the full potential offered by multimedia communication to be realised; neither is it likely to lead to generally useful tools which can used by other researchers in their experimentation.

OSI and the DARPA Internet have paid little or no attention to application requirements, and even less to multimedia application and communication requirements. They have concentrated on providing transport services across heterogeneous computer systems and networks. RAVI is a proposed standard which can be viewed as extending OSI to cover the application level; its principal application area to date has been that of computer assisted training.

ODP and ISA are addressing application requirements but have yet to tackle multimedia. Similarly research into distributed computing has evolved from message passing to RPC and various optimised styles of RPC. Again little or no attention has been paid to multimedia.

This chapter has examined the background to multimedia systems and found it to be lacking any effective support for the construction of multimedia applications. Most of the related work and standardisation efforts have concentrated on physical level aspects whilst the architectural work of OSI, DARPA and ODP has yet to pay attention to multimedia. A great deal of research is now

under way aimed at overcoming these shortcomings, which is described in the following chapter. The goal of this dissertation is to draw upon this research to construct an architecture which can better support multimedia applications.

3

Related Research

This chapter presents a survey of research related to this dissertation. A large number of projects are covered in brief in order to identify broad research directions. A smaller number of projects which are of direct relevance to this dissertation are discussed in more detail.

3.1 Multimedia Networks

A great deal of effort has been applied to the design and implementation of multimedia or multi-service networks. Some of these networks were specifically designed for multimedia traffic, whilst others are initially designed for high speed data communications and have subsequently been adapted for multimedia use. None of the networks described are in widespread use, although several have been adopted as standards and are becoming more common place.

3.1.1 FDDI I and II

FDDI (Fibre Distributed Data Interface) [**Ross86**] was originally intended as a data or packet oriented high speed local area network. A Timed Token Rotation protocol is used to bound the token latency. This scheme imposes a *single* upper bound on the token latency for all hosts attached to the network and thus makes it impossible to provide, efficiently, the low latency and fine granularity sharing required for real-time traffic at the same time as providing the higher throughput required by data traffic on the same network. FDDI II attempts to overcome this problem by defining four different traffic types (or priorities): isochronous channels, synchronous traffic, restricted and unrestricted asynchronous traffic. Isochronous channels effectively provide direct access to the network; packets are clocked into and out of the network using a single (usually external) clock. Synchronous traffic is guaranteed a maximum transmission delay of twice the timed token rotation time. The remaining types must simply wait for the restricted and unrestricted tokens respectively before being granted access to the network. FDDI II thus provides a mix of slotted and token ring media access protocols. Although FDDI II caters for multimedia traffic, its increased complexity means that complete implementations will not be available for some time yet

and are likely to be very expensive. Commercial implementations of the original FDDI ring are just beginning to appear.

3.1.2 Distributed Queued Dual Bus

The Distributed Queue Dual Bus (DQDB) network, formerly know as QPSX (Queued Packet and Synchronous Switch), is based on dual busses operating in opposite directions, thus allowing full duplex communication between each pair of nodes. DQDB provides separate isochronous and non-isochronous services which function independently of one another with the total capacity of the network being shared dynamically between them. DQDB is now embodied as the IEEE 802.6 standard.

3.1.3 Cambridge Fast Ring

The Cambridge Fast Ring (CFR) as described in **[Temple84][Hopper86]** is a slotted ring with a fixed 32 byte cell size designed to run at 100Mbits/s. Its high speed, small cell size (providing a fine granularity of bandwidth sharing) and fair access mechanism (effectively guaranteeing a minimum point-to-point bandwidth and a maximum access delay) make it suitable for multimedia traffic. The CFR is currently being used within the Pandora project (see section 3.7.4) to carry real-time voice and video.

3.1.4 Cambridge Backbone Network

The Cambridge Backbone Ring **[Greaves90]** is a fibre optic network designed to run in the 500 to 2000 Mbits/s range. It is an ATM style network designed from the outset to efficiently support voice and video traffic in addition to data. A slotted ring access protocol is used.

3.1.5 Terrestrial Wideband Network

The Terrestrial Wideband Network (TWBNet) is a wide area network spanning the U.S.A. from Boston to Los Angeles. It is based on commercially available T1 (1.792Mbits/s) telephone trunks with plans to upgrade to DS3 (40Mbits/s) lines in the future. TWBNet is an initial part of DARPA's Defense Research Internet (DRI). Bandwidth management is provided using the BBN Dual Bus Protocol (DBP) which is a type of Distributed Queue Dual Bus similar to IEEE 802.6. DBP includes extensions to support wide area networking and multimedia voice and video conferencing. Access to TWBNet is provided by IP and Stream Protocol (ST) (see section 3.3.2) gateways.

3.1.6 Fairisle

Fairisle, a joint project between the University of Cambridge Computer Laboratory and Hewlett Packard Laboratories in Bristol, aims to design and build a prototype for a 200Mbits/s ATM network, using fast packet switching techniques. Once built, the network is to be used for research into the management and control functions required by networks to support multimedia applications.

3.2 Digital Voice and Video

A great deal of work has gone into the design, simulation and implementation of protocols for real-time voice and video; the June 1989 issue of the IEEE Journal of Selected Areas in Communications was devoted to this subject. The emergence of ATM has lead to the increasing use of variable and so-called, hierarchical, encoding schemes which provide a constant quality service using varying amounts of bandwidth. This is in contrast to the majority of existing encodings and protocols which use a constant amount of bandwidth to provide a variable quality service. The variable rate encoding schemes usually have a minimum bandwidth requirement which must always be available and a higher bandwidth requirement which may be used if available, but whose absence will not adversely affect the quality of the service provided.

3.3 Communication Architectures

3.3.1 Unison and MSN

Project Unison [**Tennenhouse87**][**Tennenhouse89b**] demonstrated the use of ISDN to interconnect high speed local area networks such as Cambridge Rings, Cambridge Fast Rings and Ethernets. In particular the dynamic management of ISDN bandwidth using out of band signaling techniques (via the ISDN D channel) was investigated [**Harita89**].

The Unison protocol architecture was subsequently extended to support multi-service and internetworking requirements and led to the Multi-Service Network (MSN) architecture [**McAuley89**]. The MSN architecture is designed for efficient operation over high speed, ATM, networks as well as more traditional data networks and to allow for the subsequent hardware implementation of the lower levels of the protocol stack. Particular attention is paid to avoiding unnecessary demultiplexing and fragmentation as these introduce large amounts of jitter. The MSN protocols use the establishment of Lightweight Virtual Circuits to pre-allocate host and gateway resources in order to meet application specified requirements. The speed obtained for both local area and internetworking is directly attributable to this resource pre-allocation.

3.3.2 Defense Research Internet

The DARPA DRI is intended to support wide area multimedia applications; currently conferencing[1] is being supported over TWBNet. The Stream Protocol[2] (ST) is used to carry voice and video traffic. ST is at the same level as IP, but provides an explicit setup phase during which an application may specify its communication requirements. The ST gateways are then able to set up an appropriate route through the wide area network and to reserve sufficient gateway and network bandwidth to meet the application requirements. If insufficient resources are available the connection is refused.

Casner [**Casner90a**] describes how the problems of video clock synchronisation, end-to-end delay and packet loss are solved by TWBNet and ST. The need for clock synchronisation was removed by arranging for the receiving video codec to be run at a higher rate than the transmitting codec and for it to provide a means of indicating that no new sample is available. Thus for every clock cycle the receiving codec either plays back a newly arrived sample or is told that no sample has

[1] For a fuller description of computer conferencing see section 3.8.

[2] The original ST protocol is specified in IEN 119, [**Forgie79**] and has just been superceded by ST-II described in [**Casner90b**].

arrived and to do nothing. This approach also removes the need for receiver buffering and thus reduces end-to-end delay. The bandwidth reservation provided by the ST protocol helps reduce delay by pre-allocating resources and thus allowing fast packet forwarding within gateways. Packet loss due to buffer overflow is reduced by ST resource pre-allocation and by the use of forward error correction.

The Dual Bus Protocol used over TWBNet allows for dynamic creation of multicast groups with the underlying network providing packet replication and multi-site delivery. An application need only send a single packet to a multicast group and the network will replicate and deliver the packet to each group member. The source address in every multicast packet can be used to distinguish between streams originating at different sites.

3.3.3 Extending OSI

Salmony and Shepherd [**Salmony89**] propose that the OSI transport layer be extended to support multimedia by the inclusion of multi-channel synchronisation, a multicast facility and multi-connection management.[3] Two new concepts are added to implement synchronisation: namely *Synchronisation Markers* and *Synchronisation Channels.*

Synchronisation Markers are embedded in the data stream by the sender, so that the receiver can then compare markers from multiple streams and to buffer data until all the requisite markers are received before delivering the data. This is a simple scheme that may require extensive buffering and involves modification of the application data stream. However, it suffers from the fact that only a single, temporal, synchronisation scheme is possible and that the application has little or no control over this synchronisation.

Synchronisation Channels are intended to overcome these problems by providing an *out-of-band* means of specifying synchronisation relationships between multiple media. The synchronisation channel is used to instruct the receiver as to what order the data stream components are to be presented to the application. Such a scheme allows for greater synchronisation flexibility and avoids data stream modification, but introduces the greater problem of identifying and specifying data stream components and their inter-relationships.

This approach of extending the transport layer to support a particular set of anticipated application synchronisation requirements is in keeping with the OSI philosophy of identifying a restricted set of common applications which are then standardised. The example application used by Salmony and Shepherd is DARPA Multimedia Mail (see section 3.4.2) which allows for the *independent*, *sequential* or *simultaneous* presentation of mail document components.

3.3.4 Magnet

Magnet [**Lazar87**][**Lazar85**] was a test-bed for investigating the integration of multimedia traffic into a local area network environment. To efficiently support the varying mix of isochronous and non-isochronous traffic expected under practical use, an adaptive, 100Mbits/s, fibre-optic network was constructed. Adaptation is implemented by varying bandwidth and buffer allocation for each host in order to meet application requirements. An expert system instructs the low level bus controllers connecting hosts to the network as to how much bandwidth and buffer space to use.

A multiprocessor workstation, called EDDY, was built to experiment with the network. EDDY used separate service processors for processing voice, video and other network data; it also used

[3] The ability to establish and destroy multiple related connections simultaneously.

separate busses to interconnect the processors to each other and to the network. A variable rate, fixed quality video protocol was implemented to transport video between workstations.

An extension to the OSI model, called the Integrated Reference Model (IRM) [**Lazar86**] was designed. The IRM includes explicit representations for the resource management (M), connection management and control (C) and user data (U) information flows. IRM slices the seven layer OSI stack vertically to give three planes called the M, C and U planes. Layers in a given plane can, and do, communicate with adjacent layers in the other planes. Connection establishment is a two phase process, the first of which allocates resources for the second phase as well as subsequent user communication. The second phase allows for negotiation between sender and receiver to determine if the receiver wishes to accept the connection. Once a connection is established data may flow through the U plane. The separation into planes explicitly supports the use of *out-of-band* control techniques.

Magnet II [**Temple89**] is based on Asynchronous Time Sharing which separates traffic into different classes which expect a similar quality of service and prioritises them accordingly. Magnet II extends Magnet to the metropolitan area by using commercially available 45Mbits/s (T3/DS3) links to interconnect local area Magnet networks. A distributed, knowledge based Traffic Control Architecture (TCA) called Wiener [**Mazumdar89**] controls the hardware support for network adaptation.

The resource management and adaptation facilities of both Magnet and Magnet II are fundamental components of the network and as such have no control over higher level resource management.

3.4 Multimedia Documents and Electronic Mail

These projects allow for the creation and editing of multimedia documents; such documents typically include text, graphics, spreadsheets, voice and more recently video. Once created such documents may be transmitted asynchronously to other users using standard document and transport protocols. The absence of synchronous or real-time communication allows the use of existing network data oriented protocols (such as TCP). The storage of multimedia documents presents a number of difficulties arising from the voluminous nature of voice and especially video.

3.4.1 Agora

The Agora [**Naffah86**] project designed and implemented a multimedia document editor and mail system. The architecture of the Agora mail system was defined at the same time as the IFIP [**IFIP-WG6.579**] and CCITT X.400 [**CCITT-X.400**] models and is very similar to these, see figure 3.1.[4] This architecture defines a user agent providing user interface and application functions, a name server, a message server providing mail boxes (and therefore storage) for multimedia messages and a message transfer service for transmitting messages from one user agent to another. Within the message transfer system, the message transfer ends (MPE's) communicate with user agents or with other MPE's.

3.4.2 DARPA Multimedia Mail System

The DARPA Multimedia Mail Project [**Reynolds85**][**Postel88**] was a large project involving some ten organisations and produced two separate implementations of multimedia document editors.

[4] These figures are based on those in [**Naffah86**] and [**Reynolds85**] respectively.

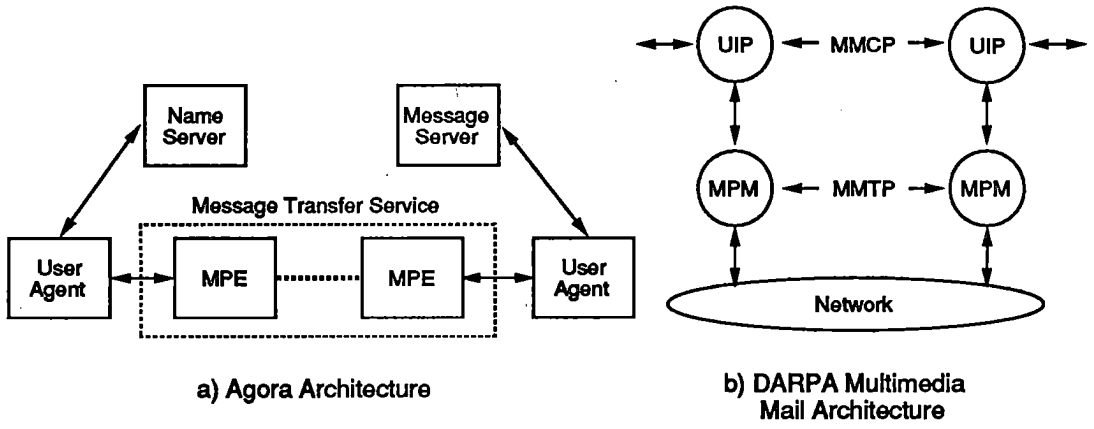

Figure 3.1: Multimedia Mail System Architectures

Common protocols were defined for document format [**Postel82a**] and transmission [**Postel82b**] to ensure interoperability. The architecture for these systems is illustrated in figure 3.1; UIP stands for User Interface Program, MPM for Message Processing Module, MMTP for Multimedia Mail Transport Protocol and MMCP for Multimedia Mail Content Protocol. The main difference between this architecture and that used in Agora is the absence of a name and message server; there is no need for a name server since the Internet already provides equivalent functionality and messages are stored in the host file systems.

The ISI Multimedia Mail Handler runs on Xerox 1108 machines and is written in Interlisp; the Diamond [**Thomas85**] editor was implemented on Sun workstations in C by BBN. Both systems allow creation, editing, transmission and management of multimedia documents, which may contain text, graphics, images, spreadsheets and digital speech. Within a document voice annotation is represented by an icon and voice caption, the user may invoke voice playback via the icon using a mouse. Voice editing is achieved using a graphical representation of the speech waveform and allowing the user to *cut and paste* voice segments using the mouse. Slate [**Lison89**] is a product version of Diamond marketed by BBN.

Each multimedia document has a number of constituent components which are in some way related to each other. These relationships must be maintained when the document is presented to a user. Document components can be labelled as being *independent*, *simultaneous* or *sequential*. Text and associated annotating voice will be simultaneous, whereas an image and associated text will be sequential. There is no synchronisation between simultaneous components: both are started at the same time and allowed to run independently of one another, assuming that there will no loss of synchronisation between them. Given that real-time playback only involves the local host this approach works well in practice.

3.4.3 Minos

Minos concentrated on the presentation of, and information extraction from, multimedia documents [**Christodoulakis86b**][**Christodoulakis86a**]. A client-server architecture is used, with a central server implementing the multimedia object store (multimedia documents are composed from such objects) connected to a series of client workstations by an Ethernet. The client workstations are decoupled from the real-time limitations of the network by taking local copies of objects to be edited and played back in real-time and then accessing them from local storage.

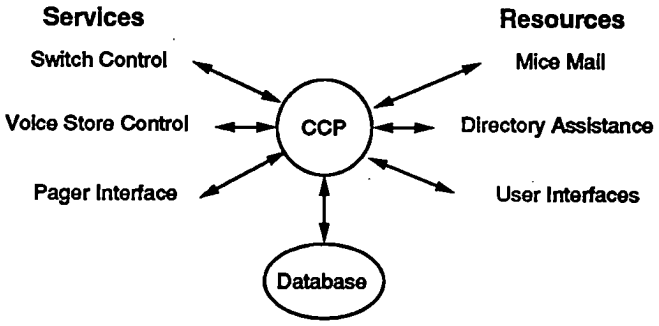

Figure 3.2: MICE Architecture

3.5 Centralised Architectures for Telephony

This section describes a number of projects which have used computers to control and supplement traditional telephony services. They have typically made use of PABX's which allow call management to be controlled by computer. This is in contrast to PABX manufacturers providing new and more sophisticated functionality such as voice storage and mail directly within the PABX. The use of workstations allows the provision of user customisation and additional functionality (in particular databases) which cannot be conveniently provided using a telephone handset. Schmandt [**Schmandt89b**] argues this point in more detail and presents two applications, Phonetool and Rolotool, illustrating the advantages gained.

All of the systems described below exhibit a low degree of hardware and network integration, using separate hardware and networks for voice and data communication.

3.5.1 BerBell

The BerBell [**Redman87**] telephone switch was amongst the first to allow for computer control, via a serial line, of its operation. BerBell provides two programming interfaces to its call management functions: a C language interface, and a proprietary interpreted language called BERPS. Users may then either choose to use pre-existing applications or to construct their own.

3.5.2 MICE

The Modular Integrated Communications Environment (MICE) [**Herman87**] was designed to allow for fast prototyping of, and experimentation with, new telephony based services. Figure 3.2[5] illustrates this software architecture. A central control process (CCP) and associated database is used to store service configurations and to instantiate the services in response to user requests. A simple interprocess communication system allows agent processes providing services to communicate with the CCP and for the CCP to communicate with the processes managing the telephone switch and associated hardware. A variety of services were provided including simple voice mail and paging services. These made use of the voice storage and retrieval functions provided by the PABX and a speech synthesiser to convert stored voice back into speech. DynaMice [**Root86**] used a powerful workstation with bitmapped display to provide a very informative, direct manipulation style of interface to the underlying call management services.

[5] This figure is based on that appearing in [**Herman87**]

MICE has recently been extended to incorporate ISDN access and terminals [**Chow90**]; this work has centered on the construction of a protocol gateway connecting the existing MICE network with the ISDN.

3.5.3 Computer Integrated Telephony

DEC's Computer Integrated Telephony (CIT) [**Strathmeyer87**] is another example of a centralised architecture; in this case the ISDN provides call management functions via D channel signaling. Strathmeyer identifies the need for increased functional integration between the application and communication components, with the ISDN viewed as the principal enabling technology. CIT and similar applications are set to proliferate as ISDN becomes more widely available.

3.6 Distributed Architectures for Audio

Audio is used to refer to a wider range of services than those implied by telephony. These additional services are made possible by the use of a distributed architecture, whereby control, generation, presentation, storage and communication functions are distributed over a number of physically separate system components.

3.6.1 Etherphone

The Etherphone project [**Swinehart83**] built custom telephone handsets implementing a real-time voice protocol over a 3Mbits/s Ethernet. Each handset has sufficient intelligence to be controlled over the Ethernet using the Cedar RPC protocol. Applications typically run on user workstations and communicate with Etherphones and other servers using Cedar RPC. Sophisticated call management operations and programming interfaces are provided by a central *Voice Control Server* [**Swinehart87**]. Text documents may be annotated with voice; such voice annotations are identified by surrounding a text character with a distinctive shape and may be played back, inserted or deleted using on-screen menus. A voice editor allows annotations to be displayed graphically using a "capillary tube" representation, within which speech appears black and silence as white. A play back cue, or cursor, moves along the capillary tube in time with the voice. Cut and paste editing operations are used to edit the voice. A *Voice Storage Server* designed to support voice editing was also implemented [**Terry88**]. A *Text-to-Speech* server is used to convert text strings to voice which can then be played back via an Etherphone.

The Etherphone project paid particular attention to architectural and system support requirements [**Swinehart88**] and developed a *Voice Systems Architecture* to meet the following goals:

Completeness: to be able to specify the role of system components (e.g. telephone transmission, switching and associated network services) in supporting the range of applications required, e.g. computer controlled telephony and voice recording, storage and editing.

Programmability: to allow for existing applications to be modified and extended. Simple applications should be easy to construct and complex ones possible. A fault in one application should not have an adverse affect on other applications and users.

Openness: by defining component functions rigorously it should be possible to replace individual components with re-implementations of the same functions.

A five layer reference model (similar in spirit to OSI) is defined as follows:

Physical Layer: represents the physical transmission medium (OSI Layer 1).

Transmission Layer: provides real-time voice encoding and transport protocols, as well as non real-time control protocols. Nothing is said about the relationships between the real-time and associated control protocols with regard to performance and synchronisation requirements. This layer corresponds to OSI Layers 2, 3 4 and 6, but not 5.

Conversation Layer: provides a uniform approach to the establishment and management of voice connections, called *conversations*, between services. It also deals with the distribution of conversation state and associated state transitions across interested parties. All communication between system components are mediated by this layer. The conversation layer is analogous to OSI layer 5.

Service Layer: identifies useful voice related services such as telephony, voice playback, recording and storage, speech recognition and synthesis. This corresponds to OSI layer 7.

Applications Layer: contains the client applications that make use of lower layer functionality.

The conversation layer is of crucial importance, and a sound design and implementation of this layer must be produced if any practical implementations of the architecture are to meet the goals set. Within the current Etherphone system the Voice Control Server constitutes a centralised conversation layer implementation; this centralisation is an implementation, rather, than an architectural constraint. The Etherphone system has recently been extended to handle video (see section 3.7.1) without requiring any major modification and thus validating many of the ideas used in its design.

Although the Etherphone architecture has demonstrated its utility, it fails to address the problems of heterogeneity and synchronisation as described in chapter 4. The entire system has been implemented within the extremely powerful, but homogeneous, Cedar programming environment using a single instance of voice hardware, namely the Etherphone. Synchronisation between related media, such as the playback cue used for voice editing and the real-time voice being played back is *open loop*; that is, no feedback is provided by the voice stream to ensure that the cue is kept in step. For instance, if the voice connection breaks or is delayed the cue will continue moving along the capillary tube as if nothing had happened. Similarly the assumption is made that sufficiently high bandwidth, low delay and predictable jitter are provided by the Ethernet (which is physically separate from the Ethernet used by the rest of Xerox PARC) for voice communication and control of this communication to proceed in real-time.

3.6.2 ISLAND

The Integrated Services Local Area Network Development (ISLAND) [**Ades87**][**Calnan87**] project investigated the transmission, storage and manipulation of real-time voice in a distributed computing environment. ISLAND used the Cambridge Ring [**Wilkes79**] local area network and made extensive use of the Cambridge Distributed Computing System (CDCS) [**Needham82**]. Voice communication was provided by custom built Ringphones interconnected by a Cambridge Ring. A new protocol was developed for voice transport between Ringphones; the existing Single Shot Protocol (a primitive RPC protocol) was used to control the Ringphones over the network. The ISLAND Ringphones were designed to be as simple as possible (to increase reliability), and to be controlled by remote software. In this way the controlling software could be implemented using all the facilities provided by the CDCS, whilst the Ringphones could be optimised for both hardware and software reliability. A distributed, replicated, software implementation of a PABX (the Exchange) was constructed [**Want88**] with each component of the Exchange executing on a separate network server to provide a high degree of fault tolerance. A voice editor [**Calnan89**] allowed voice to be edited using only the Ringphone handset. To make the use of the handset practical

the voice was automatically structured into a series of silence delimited phrases with play back, insertion, deletion and replacement operations occurring at the phrase level.

The ISLAND architecture originally outlined in Ades' thesis [**Ades87**] and subsequently refined by Calnan [**Calnan87**] partitions the system into components rather than layers; this decomposition is based on the low levels of hardware integration possible for voice at the time. Media specific *Terminals* or *Devices* (such as Ringphones) represent the most basic components in the system. Such Devices are usually used in combination under higher level control, provided by a *Conductor*, which furnishes applications with an integrated view of the underlying system. Manipulation and control of a single medium is provided by a *Translator*; Translators are usually only used directly by Conductors in order to satisfy application requests. The use of Conductors is not mandatory and it is possible to access Devices and Translators directly. Testing of new Devices and Translators is usually carried out independently of a Conductor and once the new Devices and Translators are working the Conductors can be updated to manage the new functionality.

A Conductor is defined to provide its client applications with device and location independence. A Conductor mediates all access between physical devices and applications and maps from the logical device interface it presents to applications and the underlying physical device interfaces. Access to network servers (such as for voice storage) is also via a Conductor, thus providing the same degree of independence for servers as for devices. It is also common for servers to use a Conductor to provide device and location independence with regard to its clients. Conductors are also responsible for managing multiple related physical devices in order to achieve a common goal; for instance a single operation to play back a previously recorded voice file requires the Conductor to instruct the server to play back the voice *and* a Ringphone to listen to the voice stream to be played back. Resource management is also provided by Conductors, for example access to a single physical device from multiple applications is mediated by a Conductor.

A Translator is charged with providing interfaces for the control and manipulation of a single medium and defines the interfaces between real-time and non real-time system components. Translators perform the following functions:

1. Convert between real-time and non real-time encodings, e.g. between the small packet sizes used between Ringphones and the much larger blocks required by the voice storage server.
2. Provide an interface for the control of a medium, in particular for capture and display.
3. Provide an interface allowing the controlled medium to be manipulated.

Translators are intended to be designed to provide a range of interfaces from which applications can choose the one best suited to their needs. Conductors are responsible for the insertion of appropriate Translators. Returning to the example of voice playback, the Conductor will insert a Voice Translator between the storage server and Ringphone as illustrated in figure 3.3.

The Exchange provides a mechanism whereby other applications and servers may gain and relinquish control of individual Ringphones. Thus, the Exchange and all other system components are able to co-exist within the same environment.

The ISLAND architecture uses the Conductor to manage heterogeneity and to provide an integrated view of an un-integrated hardware base. ISLAND demonstrated that voice could be effectively integrated into a distributed computing environment. However no attention was paid to the synchronisation of related media. A great deal of effort went into the design and implementation of the Ringphones and hardware for the server machines in order to ensure that they could meet the demands made by real-time voice. Similarly the Cambridge Ring network provides guaranteed point-to-point bandwidth and access delay.

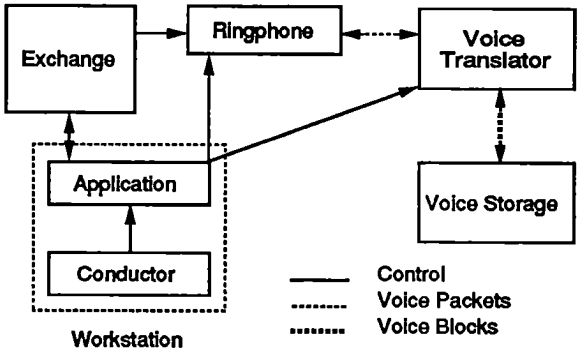

Figure 3.3: Use of Conductors in ISLAND

3.6.2.1 MIT Audio Server

A similar project at MIT [**Schmandt88**] used an IBM XT equipped with plug-in cards for telephone interfaces, voice digitisation and recognition, as a per workstation audio server. These audio servers were controlled by a low speed serial line from a powerful workstation. Real time voice transport was implemented using analogue telephone lines. This infrastructure formed the basis for a number of other projects including the Conversational Desktop and Pitchtool.

In this system the serial line, and access to it, constituted the main performance bottleneck for implementing real-time control of the audio server's functions. To overcome this, operations requiring real-time performance have two variants; the first is used to *prepare* for a subsequent operation, thus allowing buffers to be created and filled from disk, before the second, *continue*, operation is issued. Examples are *prepare_play* and *play*, *prepare_record* and *record*. This scheme minimises the delay between the continue operation being issued and it taking effect with the result that it appears instantaneous. A second drawback was encountered in the situation where two identical operations were required in quick succession; the problem being that the second operation could not be prepared until the first was completed. To overcome this a queue for prepare operations is provided for each operation type.

3.7 Video

The following projects show that the multimedia integration of video is following the same evolutionary path as that taken by voice.

3.7.1 Etherphone Video

The Etherphone project has recently been extended by the addition of a central video switch, an analogue distribution network and a central analogue video mixer to display up to four video images on a separate colour monitor associated with each workstation. The video switch and mixer are under computer control and identical editing facilities are provided for video as for voice. The incorporation of video required a minimal amount of software to drive the new hardware and involved no major changes to the existing architecture and servers.

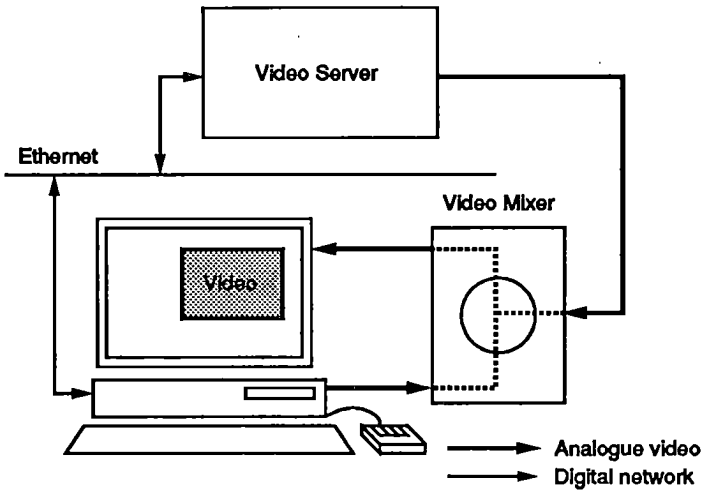

Figure 3.4: Palantir Architecture

3.7.2 MUSE

The Athena MUSE project [**Hodges89**] set out to build a system which would reduce the "time and skill" required to construct multimedia teaching and learning applications. A centralised architecture is used, with a central video server (using video discs), a campus wide cable TV network and workstations fitted with Parallax boards. The Parallax board accepts an analogue video input and then digitises this directly into its frame-buffer, giving the workstation direct access to digital video. In addition some of the workstations have local video disc players. MUSE provides a general mechanism for representing temporal relationships between multiple media, e.g. text used to annotate video must be rewound and fast forwarded in time with the video. Rather than relating such streams to each other, an independent timer is used to drive both of them, which allows for the addition and removal of media without disturbing existing relationships.

3.7.3 Palantir Project

The Palantir project at the University of Kent at Canterbury (UKC) is building a *video server* which is intended to provide a variety of video services. A video server manipulates analogue video, but is controlled via a digital network. Each workstation is equipped with a *video mixer* which accepts analogue video from the server and mixes it directly into the workstation's monitor output. Currently Ethernet is used for the control network and an analogue video network and switch are used to route video to and from, the video server and workstations. Figure 3.4 outlines this system architecture; note that the analogue network switch is not shown. The server and mixer allow external control of the placement of video windows on the display and are designed to allow interworking with standard workstation window systems. UKC plan to use the ANSA Testbench to implement the control and management interfaces to the video server and network.

3.7.4 Pandora

Pandora [**Hopper90**], a collaborative project between the University of Cambridge Computer Laboratory and Olivetti Research Limited, uses a distributed architecture. Each workstation is

equipped with a complex multimedia peripheral, called *Pandora's Box*, which implements real-time voice and video communication. The Cambridge Fast Ring is used to provide real-time voice and video transport between boxes. Internally, a Pandora Box is a structured as a set of separate devices which are interconnected using a central switch, both to each other, and to network source and sink devices; the workstation controls the interconnection of devices. Pandora Boxes currently contain as many as five Inmos Transputers controlled via a 20Mbits/s Transputer Link, and use an analogue mixer to display video on the workstation monitor under workstation control. The controlling workstation software is charged with managing connections between boxes. The X11 window system is used and has been extended via the protocol extension mechanism, to provide a programming interface to the video capabilities of the Pandora Box. The design and implementation of the controlling software is at an early stage.

Applications built so far include a simple "video-phone", and a video mail system supporting video storage and limited video editing.

3.7.5 Lancaster Distributed Multimedia Research Group

The Distributed Multimedia Research Group at the University of Lancaster are investigating several aspects of distributed multimedia systems, including the construction of a powerful "Multimedia Network Interface" multi-processor for providing real-time voice and video communication between general purpose workstations. This network interface consists of a number of Inmos Transputers, and is connected to the workstation via a 20Mbits/s Transputer Link. This network interface is not only charged with implementing voice and video communication, but also implements an X server and an instance of the ANSA Testbench for control of the interface's voice and video devices.

A network emulator is also being constructed to provide real-time voice and video communication between the network interfaces and also to allow experimentation with a range of different styles of network. The initial configuration is for ATM, with FDDI and ISDN support to follow.

Other on-going work includes an investigation of the requirements made by the provision of transactions in a multimedia system and an investigation into the heterogeneity requirements made by multimedia systems.

Much of the Lancaster work is being carried out within the framework of the ANSA architecture and use is being made of the ANSA Testbench.

3.8 Computer Supported Cooperative Working

Computer Supported Cooperative Working (CSCW), or Computer Conferencing as it is sometimes called, is concerned with using computers to allow physically distant individuals to cooperate on a shared piece of work or information. Such cooperation typically takes the form of providing access to, and manipulation of, a "shared workspace", within which participants have a consistent view of the same information. The information media commonly supported are text, graphics, bitmaps, still images; voice is most often provided via external means (e.g. via a PBX). Two main problems are encountered in providing such a shared workspace: ensuring that all participants see the same consistent view of the workspace and providing effective management of access to the workspace for manipulation (often referred to as *floor control*).

Centralised architectures (see figure 3.5) use a single *conference server* to accept input and output from participants and then to multicast this to all other participants. Such an architecture is

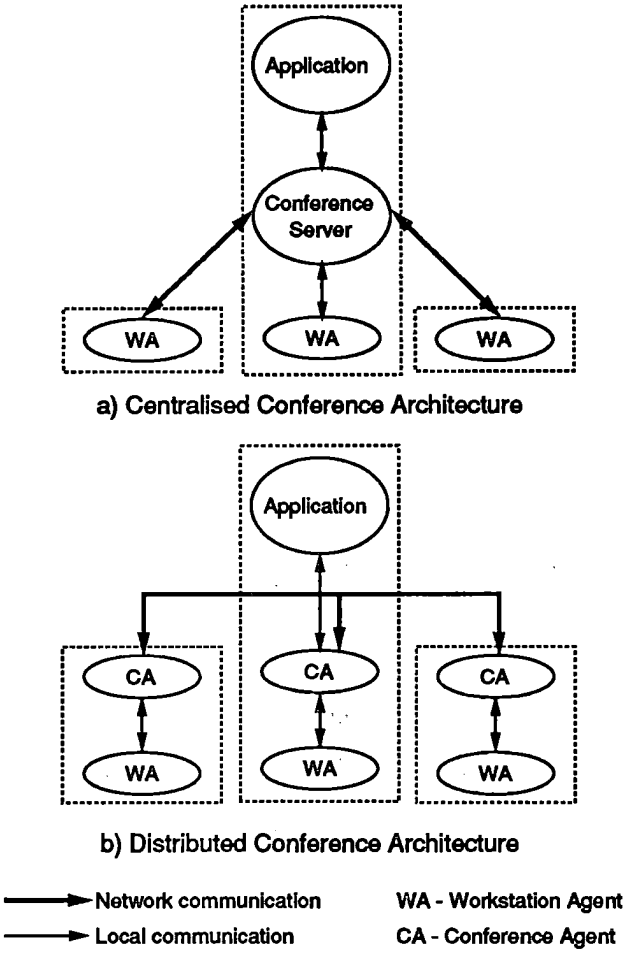

Figure 3.5: Centralised and Distributed Conference Architectures

straightforward to implement; however the conference server is an inherent performance bottleneck and is likely to lead to poor interactive response. It also doubles the network traffic; all messages are sent twice, once to the server and then again to all other conferees. Distributed architectures (see figure 3.5) attempt to overcome these drawbacks by replicating much of the conference server functionality at each participant's workstation. Each workstation runs a *conference agent*, whose primary function is to multicast updates generated locally to all other agents and to accept such updates from other agents. This approach improves performance and reduces network traffic at the expense of the increased complexity required to maintain consistency. The term *workstation agent* is used to collectively refer to a workstation's window and input/output system.

3.8.1 RTCAL and MBlink

The RTCAL and MBlink systems developed at MIT [**Sarin85**] represent early CSCW systems; both systems used external voice channels to augment the shared workspace provided. RTCAL implemented a shared calendar and diary for scheduling group meetings. MBlink investigated the low level issues involved in replicating a bit mapped display across multiple workstations.

3.8.2 EMCE

SRI constructed an Experimental Multimedia Conferencing Environment (EMCE) [**Aguilar86**] as part of a larger project to build a Command and Control Workstation [**Poggio85**] for the U.S. Navy. EMCE was implemented on Sun workstations, interconnected by a 10Mbits/s Ethernet, each equipped with an Adams-Russel Speech Processing Peripheral controlled via a serial line. Speech is encoded using Linear Predictive Code Modulation at 2400bits/s. To facilitate the construction of complex, distributed software an object oriented approach was taken.

Only one conference participant may speak at a time, with voice packets being multicast to all other conferees. A voice activated, collision sensing, floor control scheme was used. As soon as the current floor holder's workstation receives a voice packet form another workstation it gives up the floor. If the floor is free and two speakers try to take the floor at the same time they both give it up. This policy was found to be effective for small, two to ten participant, conferences over a low delay local area network.

To maintain synchronisation between the pointer position and speech, pointer coordinates and speech are combined into a single stream which is multicast to all other conferees. Due to the relatively long delays involved in displaying graphics the EMCE designers decided not to tightly synchronise such graphics operations with speech; this assumption is no longer valid for high performance workstations which are capable of real-time or near real time graphics update and animation. All outgoing information from each workstation contains a timestamp and object or host identifier, thus allowing the receiver to distinguish between information sent at the same time by different hosts. Incoming packets are buffered (up to the some maximum delay to allow for packet reordering) and sorted in order of transmission time before being forwarded to the application. This ordering forms the basis for maintaining a consistent view of the shared workspace.

Four types of network traffic were identified as requiring different communication services:

- control commands for conference management.
- real-time interactions, e.g. voice communication and graphics updates related to voice communication (e.g. pointer movement or dragging a window).
- non real-time interactions; text and graphics updates not related to voice communication.
- bulk data transfer, for files and large graphics operations.

The suggested mapping of these traffic types onto communications services is as follows: a) control and non real-time interactions to a reliable packet protocol, b) real-time interactions to a low delay circuit channel and c) bulk data transfer to a wideband satellite channel.

3.8.3 Lantz's Conferencing System

Lantz [**Lantz86**][**Lantz87**] argues for a greater degree of integration between conferencing systems and existing applications, programming methodology and system's software. Lantz claims that this increased integration should allow for the use of existing, unmodified, applications within conferences, the incorporation of material and information created outside of the conference into it and the ability to develop applications without being forced to be aware of the presence of the conferencing system.

An experimental system was constructed using a distributed architecture and running over the V-System [**Berglund86**]. The experimental prototype did not support voice communication, rather

an external channel was used. Each workstation ran a conference agent which is responsible for mediating all communication between the workstations agent and the application being used in the conference. The conference agent takes advantage of the V-System's message passing nature and intercepts all messages passed between the application and workstation agent. For the workstation with control of the floor, the conference agent multicasts updates to all other conference agents; all the other agents simply buffer any input requests and wait for the agent with the floor to multicast the input to them. A central *Conference Manager* was used to control the conference and to implement floor control. The workstation agent required minor modification to provide the conference agent with sufficient information. Care had to be taken when floor control was transferred to avoid a loss of synchronisation due to input requests issued before the change of floor being erroneously satisfied after control had been transferred; the solution adopted was simply to abort all outstanding input requests on a change of floor control.

3.8.4 MMConf

BBN's MMConf [**Crowley89**] takes an almost identical approach to Lantz, except that MMConf runs over UNIX. Conference agents are called managers and the workstation agent is replaced by the SunView^{TM} window system. Each application is relinked with a new system library which passes all input/output events to the conference manager which is then able to multicast or buffer them as necessary. Users request floor control by taking some action within the applications window (pointer movements do not count as actions). This results in a call to the conference manager requesting the floor, which is then multicast to all other managers. On receipt of a floor control request the conference manager with the floor passes it on to all its applications which may then either, tidy up any internal state before relinquishing control, or refuse to give up control. This scheme works well when there is little contention for control; in practice the external voice channel helps to reduce such contention.

MMConf identified a large number of problems encountered with maintaining synchronisation and consistency within a distributed architecture. However, on closer examination many of problems described are not entirely due to distribution and are the result of the low level at which replication is implemented (i.e. window system events), the lack of integration between the conferencing system and the system's software supporting it and the lack of any user accessible distributed processing support for sharing common resources and synchronising access to these resources. Some of the problems described are briefly summarised below:

1. inconsistent data files at any site rapidly lead to a loss of synchronisation.
2. non-deterministic applications - some applications are either inherently non-deterministic, others as a result of implementation error.
3. user customisation of key bindings and other software options leads to misinterpretation of events multicast from a workstation with one set of bindings to another with a different set.
4. timing dependencies; many operations are faster on one kind of workstation than on another and if no means of synchronising on the termination of *all* such operations is provided then synchronisation is rapidly lost. Window scrolling was found to be particularly problematic and was disabled.
5. race conditions between events synchronised within the window system and events occurring outside of the window system (which affect conference state) can lead to different event orderings on different machines, and on the same machine, at different times.
6. differing application versions were found to lead to rapid synchronisation losses even if the differences appeared to be minor.

7. interference with conferencing by other applications. For instance an application which grabs full screen control (i.e. locks out all other applications) may well lock *all* the conference participants screens if executed on the conference floor holder's workstation.

8. adding a new member to an existing conference requires establishing the current conference state for the new participant; a possible, but inelegant solution is to playback all of the window system events made within the conference up to the current time.

9. lack of feedback led to the use of external voice communication to establish that all conference participants did have the same view of the workspace. Users were often heard to ask each other what the current state of their display was.

3.8.5 Medical Applications

Karmouch et al [**Karmouch90**], describe a comprehensive multimedia system for use in a hospital radiology department. Multimedia documents (stored in a database), incorporating X-ray images and voice annotation (provided by a PBX), are used to represent patient records. A two party conferencing facility is provided for on-line communication between radiologists and doctors, and provides simultaneous access to the same patient record. Floor control is implicitly passed between participants whenever mouse input occurs.

The system was evaluated using a seven week in-hospital trial. The results of these trials showed that the system could be made more useful by decreasing the time taken to digitise and communicate the X-ray images, by improvements in image contrast rather than resolution and by the provision of the ability to view multiple images simultaneously. The conferencing facility was used nine times during the trial and was found to be effective and easy to use.

These results illustrate that whilst multimedia systems are becoming practical, there is a strong requirement for *qualitative* improvements. It is difficult to foresee the features and requirements that the users of these systems value the most. In Karmouch's system the absence of the ability to view two X-ray images simultaneously was an unanticipated problem, whilst the tool provided for measuring angles in X-ray images was found to be of little use because it did not accurately model the way that a radiologist would perform the same measurement.

3.8.6 Floor Control

Floor control is an important component of any conferencing system and the style of floor control required is highly dependent on the style of conference being supported. Floor control policies or protocols lie along a spectrum defined by the degree of agreement required to change who has control. At one end there are implicit policies which automatically change control of the floor whenever a user takes some action which is an inherent component of his or her work. For example, whenever a user speaks or moves the mouse cursor. At the other extreme there are explicit policies that only transfer control in response to a request, (e.g. a special phrase which is recognised as requesting the floor or selecting an icon). In an explicit scheme there is some notion of a floor controller (either software or human) which is responsible for arbitrating between multiple requests. Between these two extremes, lie any number of hybrid policies which vary in the degree of interaction required to transfer control. The style of communication which can be supported by these policies ranges from the very informal (implicit policy) to the formal (explicit policy).

Ideally it should be possible to choose the most appropriate policy for the particular communication required; e.g. an implicit policy for an informal technical meeting or an explicit policy for a board meeting. However, in practice the policy choices available are constrained by the underlying

communication network and in particular by the communication delay imposed by this network. A local area network with low delay and a multicast facility is well suited to supporting an implicit policy. A wide area network with much larger delay can only support an explicit policy. The advent of high speed, low delay, wide area networks will remove this restriction.

Although implicit policies offer the potential for informal and more social communication than explicit policies they suffer from a number of disadvantages. Implicit policies do not scale well; as the number of participants increases the conference becomes chaotic as floor control is switched too rapidly for the participants to keep up. Conflict arises when two or more participants take some action which implicitly requests control of the floor at the same time; some way of deciding who should be granted control is required as is some form of synchronisation to ensure an orderly change. Completely implicit policies can only support a small number of users.

Implicit policies may be modified or extended to provide better scaling characteristics. A simple extension is to only transfer control if the current floor holder has been idle for some time. To ensure synchronisation some form of atomic multicast protocol, as advocated by Birman [**Birman87**], may be used. Note that both of these mechanisms introduce additional delay which if too large may destroy the implicit nature of the conference and lead to the adoption of a more explicit protocol between the conference participants. Any multicast algorithm must be carefully designed to remain *stable*, in terms of the computation and communication resources it requires when faced with a large number of rapid floor control requests.

3.9 VOX Audio Server

The VOX Audio Server [**Arons89**][**Schmandt89a**] is designed to integrate voice and video into standard workstation environments and user interfaces. The system architecture is intended to satisfy the following goals:

Sharing: allow multiple applications to share the same audio hardware.

Routing: provide dynamic creation of routes or connections between devices.

Real-time: be capable of handling real-time audio events.

Device independence: hardware heterogeneity must be hidden from applications wherever possible.

Extensibility: allow for unexpected uses of audio, the incorporation of new hardware and the integration of video.

The resulting architecture has been heavily influenced by the X Window system; a VOX audio server runs on each workstation with network transparent connections between the server and its clients. A *workstation manager* is intended to provide a workstation wide resource management policy and operates in an analogous manner to window system managers. Figure 3.6 illustrates this structure.

At its lowest level, VOX defines *Logical Audio Devices* (LAUD's, pronounced louds) which represent physical devices such as microphones, speakers, audio mixers and recorders. LAUD's have audio ports which may be "soldered" together to build *Compound* Logical Audio Devices or CLAUD's (clouds). Resource management is provided by allowing LAUD's to be mapped and unmapped from their physical devices; when mapped exclusive access is granted to the device. Client applications may request that devices be mapped, but only the workstation manager is able to force such mappings to occur. Figure 3.7 illustrates the construction of an answering machine CLAUD using

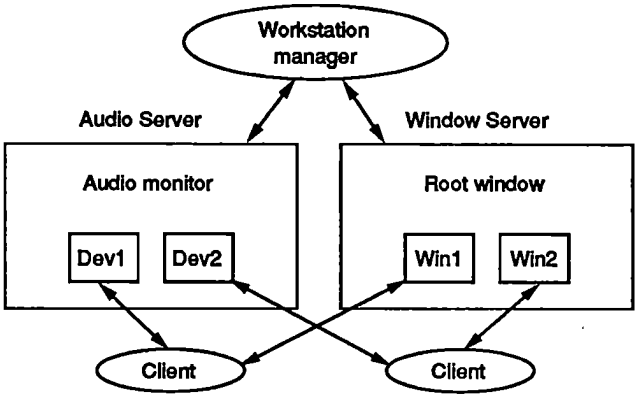

Figure 3.6: VOX Architecture

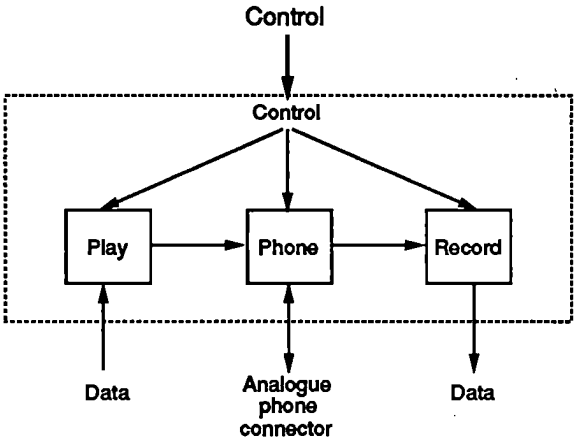

Figure 3.7: Answering Machine CLAUD

separate audio play, telephone and record LAUD's. All interaction with LAUD's and CLAUD's is achieved via queues of input and output events. To attain real-time performance the same mechanism of having separate prepare and continue operations (input events in VOX) as used for the MIT audio server (see section 3.6.2.1) is used again. The VOX server is responsible for maintaining the synchronisation, or rather, the relative ordering of events. This is achieved by multiplexing all input events from a CLAUD's component LAUD's into a single time stamped event queue and by de-multiplexing output requests within the server.

The VOX architecture appears to be primarily intended for the control of audio (and video) streams and devices which are implemented outside of the controlling workstation.[6] This approach is dictated by current technological limitations; however these limitations are set to disappear in the future and the VOX architecture does not offer any means of using the additional flexibility and finer control which will then become available.

[6]The current hardware platform transmits voice using analogue technology with a crossbar switch used to implement device interconnection.

3.10 DASH

The DASH [**Anderson90b**] project is concerned with the integration of real-time or *continuous media* (CM) into distributed computing systems. DASH assumes that all information media used, including continuous media, will be handled in entirely digital form and that continuous media will flow along the same data paths as all other data in the system. This approach, called Integrated Digital Continuous Media (IDCM), allows general purpose processing power to be applied to continuous media and offers greater flexibility as a result. Clearly IDCM assumes a very high degree of hardware *and* network integration. The project can be split into two broad components: the first deals with the communication of CM over digital computer networks, whilst the second is concerned with the construction of applications and the integration of CM into the existing system's environment.

3.10.1 DASH Resource Model

The DASH resource model [**Anderson89**] forms the basis for supporting continuous media by providing a mechanism which pre-allocates resources for real time use and guarantees sustained real-time performance. In this model, the system components which handle CM are decomposed into resources: a CPU and its scheduler, networks, host interfaces and low level protocols are all treated as resources. The unit of work for a resource is a message and continuous media consist of a stream of such messages. Such a CM stream is unidirectional, with a source and sink, and may use multiple resources between the source and sink. Resource reservation is provided by the establishment of sessions; sessions may involve a single resource or they may be end-to-end sessions which concatenate a number of basic sessions (i.e. sessions for a single resource) to provide end-to-end resource allocation.

The interface to the resource model is provided by resource managers which accept requests for resources for a specified message size, message rate and maximum end-to-end delay. The resource manager then attempts to satisfy the request using a two phase protocol. The first phase proceeds from the source to the sink and obtains a series of basic sessions satisfying the request parameters for each resource specified. The second phase proceeds in the reverse direction and attempts to optimise the resource allocation. On completion of the second phase either an end-to-end session has been created which guarantees the requested requirements, or if sufficient resources were unavailable at any stage, the session is refused.

A similar scheme is used by Ferrari [**Ferrari90**] for establishing real-time channels in a wide area network. Requests for channel establishment include parameters for minimum packet inter-arrival time, maximum packet size, delay bounds and packet loss rates. These parameters are then used to establish an end-to-end channel in the same way as that described above for DASH.

3.10.2 DASH Architecture and CMEX

DASH proposes to integrate multimedia communication into existing systems such as Mach, TCP/IP and X11. The Session Reservation Protocol (SRP) [**Anderson90a**] may be used in conjunction with TCP/IP to implement the DASH resource model. Of particular interest is the Continuous Media Extension to X (CMEX) which proposes to extend the X11 Release 4 server implementation to handle real-time voice and video and to provide synchronisation of related streams. CMEX uses an identical structure of physical, logical and composite logical devices as that used in VOX.

Two encoding abstractions are defined: namely *strands* and *ropes*. A strand represents a single

information medium stream, and multiple strands may be interleaved to create ropes. Synchronisation of related streams is achieved by interleaving multiple strands into a single rope which is then transmitted over a single communication channel; the receiver de-multiplexes the rope to obtain the individual strands. Alternatively, strands transmitted over multiple channels may be synchronised to a real-time clock and played back in lock step.

Whilst offering the potential for increased functional and multimedia integration, CMEX suffers from a number of practical and architectural disadvantages. The principal practical disadvantage centres on the fact that X servers are already very complex and very large pieces of software, adding yet more complexity is a major undertaking; in particular CMEX will almost certainly require a multi-threaded X server. Strand and rope formats must be understood by CMEX servers if they are to be synchronised, therefore the addition of new formats requires the modification of the server. In addition the synchronisation provided by the server is largely outside of application control, and offers a restricted set of synchronisation operations. These issues are discussed in more detail in section 4.2.4.3.

The CMEX server is currently at the design stage and a suitable hardware base for DASH has yet to be identified and installed.

3.11 Extending UNIX

Leung et al [**Leung90**], have built a multimedia communication system by extending UNIX to provide system level support for such communication. This approach allows the use of existing UNIX applications from within multimedia conferencing applications and provides a uniform programming paradigm for the construction of multimedia applications. Two abstractions are provided; one for communication and the other for application programming. Both are then integrated into a single, consistent programming model. An experimental 100Mbits/s fast packet switch is used to provide data and real-time voice transport between the UNIX workstations.

3.11.1 Communication

Multimedia virtual circuits are used for transporting multimedia data between hosts and are responsible for maintaining the temporal synchronisation between related media streams. Synchronisation is achieved by multiplexing the related media streams or channels (e.g. data and voice) over a single multimedia virtual circuit. Thus, there is a two level structure of multiple, separate, media channels multiplexed over a single virtual circuit with temporal synchronisation being maintained between the constituent channels. Each channel has a service class which describes its average and peak bandwidth and a priority which represents its delay requirements. A high priority channel will be granted resources such as network buffers and network access ahead of a lower priority channel. However, this two level structure means that resource management must be applied to virtual circuits as well as individual channels. This is implemented by the simple rule that the priority of a virtual circuit is the highest priority of its constituent channels. Given that a virtual circuit group consists of all the virtual circuits of the same priority the following scheme is implemented:

- *strict priority* is applied between virtual circuit groups; that is, a lower priority group cannot be transmitted whilst a higher priority group has yet to be transmitted.
- within a virtual circuit group, each virtual circuit's constituent channels are granted network access in a round-robin manner and thus provide temporal synchronisation.

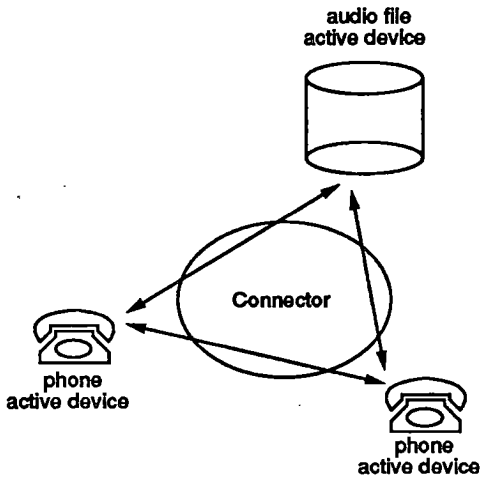

Figure 3.8: Recorded Telephone Conversation

This scheme is extended to allow the optional provision of flow control; for instance a voice channel would not be flow controlled, whereas a data channel would. When channels are combined into virtual circuits the flow control scheme may be extended to *all* of the constituent channels and therefore if one channel is stopped then so are all of the others in the circuit. Flow control is implemented using a dynamic window scheme, whereby the window size may be dynamically altered based on current resource availability. Another aspect of multimedia virtual circuits is that channels may be added to, and deleted from, existing circuits, and that the service class and priorities may be dynamically modified.

3.11.2 Programming

The observation is made that multimedia applications often require multicast (one to many) and multidrop (many to one) communications. In some cases the multicast must be ordered across all recipients; that is, *all* such recipients receive multicast data in the *same* order. Finally applications often need to add new connections, modify existing ones and destroy redundant connections.

The notion of a *connector* is introduced to satisfy these requirements and is closely modelled on the UNIX *pipe* facility. Unlike pipes, connectors, may have more than one sink or more than one source. Connectors are used to interconnect UNIX process in a similar fashion to pipes, with the added provision of multicast and multidrop communication. Processes access connectors using the standard UNIX system calls `open`, `read`, `write` and `close`. If one process writes to the connector then *all* reading processes will receive the data, similarly if several processes write to the connector then all reading processes will receive the data sent by each process in the order that it was sent. To overcome the inconsistency present in UNIX between communication between processes (provided by pipes) and communication between processes and devices (provided by input/output redirection) the idea of an *active device* is introduced which allows all multimedia devices to be treated as processes. The use of active devices means that connectors may be used in a uniform fashion to connect all multimedia devices and processes. Figure 3.8 illustrates how connectors and active devices can be used to construct a recorded telephone conversation.

The multimedia virtual circuits described above appear as active devices. Connectors are used to connect local devices to a circuit which then traverses the network to a remote machine where

the same circuit is connected to the receiving device; in this way an end-to-end channel may be established.

The active device interface provides a single input and output interface similar to that of a UNIX process. In order to allow more complex devices and processes to be implemented this interface may be extended on a per device or per process basis. The active device interface is considered to be the *base class* and devices or processes may extend this base class by adding new interfaces as required.

A high-level programming language called Non-Procedural Language (NPL) provides an event-driven programming model. Applications consist of a number of event-handlers which are invoked when the corresponding event occurs; these event-handlers are then able to create and manipulate virtual circuits and devices. Each host runs a *User Interface* and *Call Controller* component which implement a simple session protocol to allow the creation of new calls between users on separate hosts. A series of user interfaces, varying in sophistication, are provided to the system. A basic interface is provided for placing and managing calls, an advanced interface allows the creation of new services using existing NPL components and finally it is possible to create new NPL programs.

3.11.3 Extending UNIX Summary

Multiplexing of related channels onto a single circuit provides ordering of the data flowing along the constituent channels. However, this does not provide synchronisation with respect to absolute time: the resource management and scheduling policies applied to channels must provide such synchronisation. It is claimed that ordering provided by the multiplexing helps provide temporal synchronisation. This scheme suffers from many of the same disadvantages as those discussed above for CMEX; these issues are discussed in more detail in section 4.2.4.3.

The connector and active device model appears to provide a clean and powerful programming model for constructing multimedia applications.

3.12 Summary

A great deal of effort has been applied to increasing multimedia integration at the network, and low level protocol levels. Either ATM or some form of dynamic bandwidth allocation is being used to physically support multimedia traffic, whilst protocol architectures are moving towards resource management with pre-allocation of resources to provide guaranteed, sustained real-time performance. Similar effort has gone into the construction of workstation and associated peripheral hardware which can support real-time multimedia communication; again the major issue is the degree of application control and flexibility provided by these systems.

Voice systems have followed an evolutionary path from centralised to distributed architectures, both in terms of hardware and software. The same path is also being followed for video. Most progress at the application and user interface levels has been made by voice systems; whilst video related work has been concentrated at the physical level. The Etherphone and ISLAND projects both provided real-time voice communication over digital networks and implemented sophisticated applications; this level of sophistication has yet to be reached for video systems. Projects providing multimedia document and multimedia mail facilities have avoided the need for real-time voice communication by using asynchronous communication, whilst many other projects have used analogue transmission for voice. There is great variation in the degrees of hardware and network integration present in multimedia systems and there is a similar variation in the system architectures used. The inevitable delay between greater integration being possible and it becoming commonplace, and continuing

technological progress dictates that this situation is unlikely to change. Therefore, whilst the DASH IDCM approach is attractive, it is unlikely to become practical and common place for some years yet.

Although architectures such as those used in VOX and CMEX are distributed, the distribution is provided at a coarse granularity. That is, the unit of distribution is the VOX or CMEX server and the client application programs, this has the disadvantage that even minor extensions to the system's functionality require updating the servers and client applications.

A common feature of all the architectures presented is the provision of some means of separating real-time from non real-time system components. This separation is typically fixed and fundamentally affects the design of application programmes. Such separation is either forced entirely by performance constraints or results from the observation that real-time environments are generally very poor for program development whereas rich development environments (such as UNIX) are equally poor for real-time operation. Leung's work is a notable exception which attempts to integrate multimedia communication into the rich programming environment provided by UNIX.

Most systems do not provide any means of synchronising related media and have ignored the problem by using an *open loop* approach. Where synchronisation has been provided its nature has been predetermined and its operation largely outside of application control; there is little separation between synchronisation mechanisms and synchronisation policies inevitably leading to a restrictive system.

Finally the lack of functional integration is clear with projects either concentrating entirely on physical and service level issues or on application and user interface issues without paying any regard to one another.

4

Problems and Requirements

The nature of existing computer systems presents a series of *problems* which must be overcome in order to make the support of multimedia possible. In addition, the nature of multiple information media, and in particular continuous media, have *requirements* which must be satisfied by the underlying computer systems if such media are to be successfully supported. The distinction between problems and requirements provides a framework within which to discuss the issues relevant to multimedia.

4.1 Problems

Each component of a Multimedia Communication System encounters its own particular set of problems in addition to any problems common to multiple components. Recall that an MMCS has five components as follows:

- Information Media Component (IMC).
- Communications Component.
- Distributed Processing Component (DPC).
- Application Component.
- User Interface Component (UIC).

The following discussion concentrates on the IMC, Communication and Distributed Processing components, as these represent the most significant obstacle to progress in the Application and User Interface components.

4.1.1 Information Media Component Problems

Continuous media such as voice, and especially video, require much higher data rates than those supported by existing systems. Although compression may reduce these rates, it will often be necessary to support multiple simultaneous connections, increasing the aggregate rate further. Multimedia communication also demands flexible connectivity, both within individual, and between separate, workstations. Common examples include connecting a camera to a screen or network output, or a network input to screen output.

Current workstations are predominantly uniprocessor machines with bus based access to memory and input/output devices. Unfortunately, buses represent an inherent bottleneck in any system design. This bottleneck is rarely reached in current systems because bus accesses are typically bursty and caching techniques may be used to reduce bus latency and access. However, continuous media connections are often long lived, requiring sustained high data rates and are therefore more likely to saturate a bus. During such a connection little or no bandwidth may be available for other computation - potentially stalling the rest of the system. Alternatively, other computation may stall the continuous media stream in unpredictable ways and therefore make it impossible to provide the guaranteed performance required for such media.

In many existing multimedia systems, these problems are avoided by off-loading much of the multimedia processing to separate, specially designed, and often complex, peripherals or front ends. A good example is the Pandora Box (section 3.7.4), which is designed to be controlled by a separate workstation and is described as a *multimedia peripheral.* In such systems care must be taken to ensure that the control path between the controlling workstation and the peripheral provides sufficient flexibility and sufficiently low latency to allow for the effective control of the peripheral. Any latency measurements must be taken on an end-to-end basis; that is, from the user of the system all the way to the low level hardware devices. The large amount of diverse software involved in such a data path makes it difficult to provide a reliable bound on the latency introduced.

To summarise, existing workstation architectures are poorly suited to supporting the sustained, possibly multiple, high data rate connections required for multimedia. Whilst off-loading the processing to a peripheral avoids some of these problems it requires that care be taken to provide effective control of the peripheral.

4.1.2 Communications Problems

Existing communications networks generally fail to provide the high bandwidth, low latency and communication guarantees essential for multimedia communication. As seen in section 3.1, a new generation of networks is being designed and implemented with the explicit goal of overcoming these problems.

Established protocols, protocol architectures and their implementations, predate the emergence of multimedia communication and inevitably suffer from the same problems as the underlying networks. The complexity of these protocols makes these problems much harder to overcome at the protocol level than at the physical network level and consumes substantial amounts of processing time for each network packet transmitted and even more so for each packet received. A final problem associated with excessive complexity is that predicting the performance of protocols and offering the performance guarantees required for continuous media is, at best, extremely difficult. Factors contributing to this complexity include: coping with heterogeneous physical networks, providing a high degree of reliability, standards conformance, excessive multiplexing and internet routing.

The increase in network speed, from 10Mbits/s to 100-500Mbits/s, coupled with the decrease in network packet size (53 bytes for CCITT ATM) outweighs the increases in CPU speeds. This entails a net *decrease* in the amount of processing time available between potential network accesses. Therefore, the amount of protocol processing must be reduced or it will prove impossible to make full use of the increased network capacity. Another important consideration is that as network speeds increase the end-to-end delay, expressed as data bits in transit, as opposed to time, also increases. This has the effect of lengthening any end-to-end feedback loops designed to provide flow control; the information available for making flow control decisions will be out-of-date with respect to the current state of the network. The only way of overcoming these problems is to reduce protocol processing time and end-to-end delay, by implementing as much of the protocol in hardware as is practical. The complexity of existing protocols such as TCP/IP and OSI makes such hardware implementation impractical and as a result a new series of protocols are currently being designed with the explicit goal of being amenable to hardware implementation.

The principal drawback associated with hardware implementation is the lack of flexibility available once the hardware has been implemented. The large investment required to realise such hardware implementations dictates that a great deal of research be carried out to validate the design of the protocol before committing to hardware.

The presentation delay (i.e. OSI layer 6) is often a significant delay in many protocol implementations. The obvious way to reduce this delay is to improve the quality of the presentation layer implementation. Another option is the use of pipelining to overlap the presentation of one packet with the transmission of the next one.

Unfortunately, multimedia communication by definition requires multi-service protocols and out of necessity requires high speed networks. McAuley [**McAuley89**] discusses these issues in more detail and presents a new protocol architecture and associated implementation designed to overcome these problems.

4.1.3 Distributed Processing Problems

The problems posed by the distributed processing component fall into two categories: hardware and protocol heterogeneity, and inappropriate resource management policies.

4.1.4 Heterogeneity

Heterogeneity is one of the oldest problems in computer science and is usually solved by providing a level of indirection, in the form of a common interface, between the heterogeneous hardware or software and the controlling software. Such a solution works well if the underlying system differs in detail rather than fundamental functionality; even so, it is often necessary to provide an escape mechanism to allow access to implementation specific functions. The heterogeneity found in multimedia systems is further increased by the varying degrees of hardware and network integration, and the diverse range of multimedia devices and networks used.

The interfaces provided for managing hardware and network heterogeneity are often separate and different in style. For instance, Berkeley UNIX provides a file based interface for managing hardware heterogeneity and a separate, specially designed interface, Berkeley sockets, for managing protocol heterogeneity. Unfortunately, the often close relationship between multimedia devices and communication protocols, and the desire to be able to freely interconnect multimedia devices, requires that any scheme for managing hardware heterogeneity also be used for protocol heterogeneity.

There is an inevitable performance penalty associated with managing heterogeneity which existing systems assume is worth paying. However, this may not be true for multimedia systems in which great care is required to reduce the performance overhead to an acceptable level, otherwise system implementors will simply bypass any such mechanisms provided.

4.1.5 Resource Management

The other major problem encountered is that the resource management policies used by general purpose operating systems for CPU time, buffer management and network access are designed to provide fairness rather than guaranteed real-time performance. Such policies make it impossible to construct real-time multimedia applications. Whilst this is reasonable for shared machines, it is much harder to justify for a high performance workstation which has a single user for the vast majority of the time. As a result, the rich programming and run-time environments offered by such systems cannot be used by multimedia applications. Similarly the real-time operating systems which can provide the performance demanded by multimedia only provide very primitive programming and run-time environments. Some form of compromise must be found between rich programming and run-time environments on the one hand, and real-time performance on the other.

4.1.6 Common Problems

There is a fundamental tradeoff between placing functionality as near the hardware as possible for performance reasons, and for keeping it near the application and user for generality. This tradeoff is particularly evident in multimedia communication systems which inherently require high levels of performance at the same time as sufficient flexibility to experiment with new applications.

The most obvious instance of this tradeoff is that between heterogeneity management and performance. To date most multimedia systems have opted for performance, and have ignored heterogeneity issues, with the result that these systems are very difficult, or impossible, to port to new hardware and software platforms.

Representing and maintaining the relationships between multiple media is a problem faced by all components and is discussed in more detail in section 4.2.8.

4.1.7 Problems Addressed by IMAC

IMAC and its prototype implementation address a subset of the problems discussed above. A uniform interface for managing hardware, protocol and multimedia heterogeneity is provided, as is a means for representing and maintaining the relationships between multiple media. Care is taken to ensure that the performance penalty associated with the use of the heterogeneity interface is kept to a minimum.

The remaining problems associated with workstation design, operating system and protocol design, and resource management are not tackled by IMAC; however, section 9.3 presents requirements for the design and implementation of future systems to avoid these problems.

4.2 Requirements

This section discusses the requirements made by continuous media and the requirements made by multimedia on each component of an MMCS; particular attention is paid to synchronisation.

4.2.1 Continuous Media Requirements

Some information media inherently consist of a continuous sequence of symbols which have strict timing dependencies between symbols. Such media are described as *continuous* and the term *stream* is used to refer to the sequence of symbols constituting such a medium. By this definition voice, video and graphical animation are continuous, whilst text, images and graphics (excluding animation) are not.

The digitisation of continuous media may take many forms:

1. sampling the original stream at regular intervals and generating a fixed size sample representing the stream's state at the instant it was sampled.
2. sampling at regular intervals but generating samples of varying size.
3. sampling at irregular intervals and generating fixed or variable sized samples.

If there is physical separation between the point of generation and the point of presentation, then the samples must somehow be communicated from one point to the other. There are a number of ways in which a continuous media stream may be packetised for transmission over a communication network. If a single channel is used then any combination of fixed or varying sample sizes, transmitted at fixed or varying time intervals may be used. Alternatively, multiple channels may be used, whereby a single media stream is split into several sub-streams, each of which may be transmitted over a separate communication channel; some form *hierarchical* encoding is required to allow a single stream to be split into multiple sub-streams. Although the use of variable sized samples, varying transmission intervals, and multiple communication channels appears to contradict the initial definition of continuous media, this is not the case since the continuous nature of the media need only be apparent at the source and sink of the media stream. The use of such variable rate and hierarchical encodings considerably complicates the timing constraints imposed on the communication system, since the constraints must vary in line with the variation in the encoding used.

The term *isochronous* is often used as a synonym for continuous. However, within this dissertation an important distinction is made between the two. Continuity is viewed as a fundamental property of a medium, whereas the term isochronous is used to refer to how a communication medium is being used; that is, a single continuous medium may be isochronous when used in one way but merely continuous when used in another. Thus, isochronous media are a special case of continuous media, with the added requirement that they be synchronised with respect to real-time. For example, video communication between two users is isochronous, whilst a video stream which is being recorded to disc need not be synchronised to real-time and is therefore only continuous.

Real-time synchronisation is discussed further in section 4.2.9, whilst section 5.2.3.4 gives a precise definition of isochronous media as used within IMAC. The following sections discuss the requirements imposed by continuous media in detail.

Protocol	End-to-end Delay	Packet Rate	Packet Size	Compression
Etherphone	40*ms*	50/s	160 x 8 bit samples[5]	silence suppression
ISLAND	2-5*ms*	500/s	16 x 8 bit samples[6]	none

Table 4.1: Example Voice Protocols

4.2.1.1 Basic Requirements

The requirements made by continuous media are often summarised as:

1. low latency
2. high bandwidth

These requirements are stated in relative terms with respect to existing communication requirements, the remainder of this section quantifies these terms.

The CCITT defines encoding standards for voice at 32Kbits/s[1] and 64Kbits/s,[2] with a 16Kbit/s standard under development. Although these rates appear modest, the 8 bit sample size they require dictates a latency of 125μs between successive samples. It is possible to relax this latency requirement by aggregating several voice samples into a single packet for network transmission at the risk of reducing voice quality and robustness.[3]

The Etherphone [**Swinehart83**] and ISLAND [**Ades86**] voice protocols are summarised in Table 4.1. Both protocols use standard 64Kbits/s voice Codecs, an 8 bit sample size, and 125μs between samples; each network packet carries a number of 8 bit samples. Both protocols give acceptable voice quality communication over a local area network using custom built phone hardware, and represent an accurate indication of the bandwidth and latency requirements of real-time voice. The lower delay achieved by the ISLAND voice protocol is largely due to its smaller packet size, lack of compression and encryption, and the use of the Cambridge Ring as opposed to the Ethernet.

For video, the demands are far greater, with required bandwidths ranging from 4.6Mbytes/s for ISDN video phones,[4] to 20 MBytes/s for broadcast video and 144 MBytes/s for HDTV. The use of compression techniques promises to reduce these requirements, but only at the expense of introducing additional latency.

At the time of writing the Pandora Box (see section 3.7.4) can generate, transmit over the CFR, and present real-time video in one of four resolutions.[7] Video frames are captured at 128x120 resolution, with 8 bit pixels. DPCM coding is used to reduce the 8 bit pixels to 4 bits and thus halve the data rate. Each frame may be optionally sub-sampled to a resolution of 64x60, and before being displayed a frame may be interpolated to double the screen resolution. The various combinations are summarised in table 4.2.

[1] CCITT G.711 PCM coding.

[2] CCITT G.721 ADPCM coding.

[3] The loss a single network packet entails the loss of several voice samples.

[4] CCITT H.261.

[5] Representing 20*ms* voice.

[6] Representing 2*ms* of voice.

[7] Pandora video data was supplied by Alan Jones of Olivetti Research Ltd.

Sub-sampling	*KBytes/Frame*	*Interpolation*	*Resolution*
Yes	1.875	No	64x60
Yes	1.875	Yes	128x120
No	7.5	No	128x120
No	7.5	Yes	256x240

Table 4.2: Pandora Video Options

In the current implementation, 12.5 frames are transmitted each second, giving data rates of either 23.43Kbytes/s (187.5Kbits/s) and 93.75Kbytes/s (750Kbits/s). A new implementation of the network interface is under construction which will allow transmission of either 25 frames/s at the currently supported resolutions (187.5Kbytes/s), or 12.5 frames/s at 256x240 resolution (375Kbytes/s). These figures are intended to give a general idea of the data rates required for video, they are not entirely accurate in that they do not account for protocol headers.

4.2.2 Additional Requirements

In addition to the basic requirements discussed above, there are a number of other requirements which have a fundamental impact on the design of an MMCS.

1. timeliness: continuous media data is only valid for a bounded period and once this time has elapsed the data is useless.
2. variations in delay, commonly referred to as *jitter*, must be bounded to some statistical limit such as the ninety-ninth percentile,[8] and if possible, minimised.
3. high reliability is not of primary importance since voice and video are inherently tolerant of small amounts of error. However, if the medium is compressed or being recorded then the tolerable communication error rate will be decreased.
4. media and application specific error recovery strategies. Different media require different action to be taken in the face of errors. For instance, some media, primarily voice, have no meaning if stopped or delayed, whilst other media may degrade to a discrete representation; e.g. video degrades to a still image. This particular requirement is called ***continuity.***
5. synchronisation: the fact that samples are generated and presented at regular intervals means that some form of synchronisation must be maintained between the source and sink.
6. *guarantees* must be provided for latency, bandwidth, jitter and timeliness to ensure that the continuous nature of the media is maintained.
7. all of these requirements and guarantees must be met on an end-to-end basis; that is, from the point the media is generated to the point it is presented, regardless of the number of stages present in the path from generation to presentation.

The following discussion initially concentrates on the requirements of single continuous media streams, before moving on to multiple, related, continuous media streams.

[8] i.e. 99% of all packets experience less than some stated amount of jitter.

4.2.2.1 Timeliness and Latency

The notion of timeliness imposes an upper bound on the maximum acceptable latency and dictates that any data exceeding this latency is of no use. This simple observation means that late data is as useful as lost, or corrupted, data and can, therefore, be ignored. Similarly, there is no point in using acknowledgements to detect lost, corrupted or delayed packets since any retransmission on a timeout of such an acknowledgement will almost certainly arrive too late to be of any use. As a result of the timeliness requirement, any protocol design for continuous media should avoid the use of retransmission schemes. Such schemes also increase bandwidth requirements.

The primary use of re-transmission is to provide reliability. Therefore, other means must be used to provide any required level of reliability. An alternative method is Forward Error Correction (FEC) which introduces sufficient redundancy into the data transmitted to allow the receiver to recover from lost or corrupted samples. Unfortunately, FEC inherently increases bandwidth requirements and latency. The fact that voice and video, especially when presented to humans, are inherently error tolerant makes the provision of high levels of reliability unnecessary. For voice an error rate of 1%, provided each error burst is shorter than $4ms$, is often quoted as acceptable; the acceptable error rate for video is entirely dependent on the encoding and compression algorithms used.

The timeliness requirement suggests that low latency is of greater importance than reliability for continuous media; therefore, reliability is not a goal for such media.

The end-to-end path between generation and presentation can be subdivided into a series of delays as follows:

- packetisation delay; the time taken to generate the sample and transfer it to a network buffer.
- network access delay; the time the network buffer spends waiting for access to the network.
- transmission delay; the time taken to transmit the sample over the network.
- receiver buffering delay; the time spent buffering the sample at the receiver.
- presentation delay; the time taken to present the sample.

These delays will vary depending on the media and communications network used and are largely the result of contention for shared resources such as CPU time, memory and network access. On a LAN the packetisation and receiver buffering delays are likely to dominate.

Aggregating several samples for transmission in a single network packet increases the packetisation delay, but is necessary if network and communications protocols are too slow to provide the lower latency required for smaller, more frequent, samples. The sample size also affects the network utilisation and larger samples may be used to increase network utilisation. Therefore, there is a tradeoff between the sample size, decreased packetisation delay and increased network utilisation.

The Etherphone voice protocol employed a large packet size to increase network utilisation at the expense of an increased packetisation delay. The $40ms$ of end-to-end delay is broken down as follows:

$20ms$: packetisation delay.

$5ms$: encryption and Ethernet transmission.

$5ms$: software delays.

$10ms$: receiver buffering to remove jitter.

The dominant delays are for packetisation and receiver buffering, which together account for 75% of the total delay. The large packetisation delay reinforces the claim that presentation delays are a major communication cost.

4.2.2.2 Jitter

Jitter refers to the statistical variation in delay introduced by the presence of queues in computers and computer networks. Jitter may be reduced by the use of buffering at the receiver. Incoming samples are added to the tail end of a list of already received samples. Samples are removed from the head of this list at regular intervals, timed using the receiver's local clock. In this way delay variations are absorbed by variations in the size of the list. Such a buffering scheme, although reducing jitter, increases delay. Therefore, the amount of jitter allowed must be traded off against the latency requirements. Given a reliable estimate of the jitter present in the system it is possible to decide how much buffering to use (i.e. the maximum size of the list) and the latency that this buffering introduces. For such a scheme to work effectively the jitter must be bounded; if this is not the case then there is little that can be done to remove jitter. Removing jitter using receiver buffering assumes that the dominant source of jitter is the network and that the presentation of the media does not, in itself, introduce substantial amounts of jitter; in practice this is a reasonable assumption.

With the exception of receiver buffering, each source of delay described in the previous section is also a source of jitter. The delay introduced by receiver buffering is intended to reduce jitter. The largest and most studied source of jitter is that introduced by buffering for network access and network transmission. Clock rate variations are an additional source of jitter; if the generator's clock rate varies then so do the intervals at which at which it generates samples. Clock rate variations are discussed further in section 4.2.2.4.

4.2.2.3 Continuity, Media and Application Specific Error Recovery

It must be possible to take action to preserve continuity in the face of lost, corrupted or delayed samples. For video it is often possible to redisplay the previous frame (assuming it is still available) and so degrade to a still image if full-motion video can no longer be maintained. Voice is more problematic since it has no meaning if stopped. If only a few samples are lost it is possible to play "white noise" to fill in for the lost samples without introducing a noticeable degradation in quality. However, if the voice stream is broken or suspended for more than a short period of time, $10ms$ say, then there is little that can be done, other than to notify the user of the problem.

Steinmetz [**Steinmetz90**] also identifies the continuity problem and suggests a synchronisation mechanism called *partial blocking* which allows for the specification of some alternative action to be taken in the face of communication errors and suspension.

Continuity and partial blocking are just *one example* of a media specific error recovery strategy. The precise strategy required will be media and application specific. Some applications may be prepared to tolerate far greater error rates than others; for instance, a voice and video editor may be tolerant of increased communication delay whilst requiring increased reliability. Therefore, some means is required for allowing the application to control the action taken on errors.

4.2.2.4 Single Stream Synchronisation

An implicit assumption made so far is that each sample transmitted over the network contains sufficient information to detect lost and re-ordered samples. This information is required to maintain

the ordering of samples and in conjunction with meeting timeliness and jitter constraints, allows synchronisation between the source and sink to be maintained.

Another source of synchronisation problems is differing clock rates at the source and sink. If the source clock runs faster than the sink's clock then the source will generate samples faster than the sink can present them and this will eventually lead to a loss of synchronisation. The sink can monitor the rate of incoming samples and adjust its clock rate to match that of the samples and hopefully that of the source. This only ensures that the source and sink run at the *same* rate; it does not detect if both clocks are running at an incorrect rate. One approach is for each sink to periodically communicate with a source which is known to have a highly accurate and stable clock and thus allow the sink to synchronise its clock to the correct rate. Clock rate variations are almost entirely due to temperature variation and could be largely eliminated by the use of temperature compensated clocks.

The synchronisation requirement for a single stream depends on the timeliness and jitter constraints being met, on sufficient information to order samples, and on the maintenance of the same, constant, clock rate at source and sink.

4.2.2.5 End-to-end Guarantees

Continuous streams demand that their timing constraints be met for the entire duration of the stream; that is, stream requirements must be guaranteed. In order to provide guarantees a means must be found for expressing the guarantees required. Simple guarantees include minimum bandwidth, maximum latency and maximum jitter. In order to more accurately model the performance and behaviour of a real-time stream a more complex set of parameters, such as those used by DASH, is required. These parameters constitute a Quality of Service (QoS) and are commonly referred to as QoS parameters. The simplest means of providing such guarantees is to pre-allocate the necessary resources and ensure that these resources are not re-allocated for the duration of the connection. There is a need for negotiation between the guarantees requested and those which can be efficiently provided.

The end-to-end argument [**Saltzer84**] states that many communications problems can only be solved, or solved efficiently, on an end-to-end basis. This is especially applicable to multimedia communication involving human participants as the final communication end-points. All requirements and guarantees must be on end-to-end basis if the human users are to be able to effectively communicate. Therefore, QoS must be extended beyond networks and communication protocols to include operating systems, applications and user interfaces. This extension of QoS underpins the increased functional integration provided by the IMAC architecture. The use of QoS in IMAC is described in section 5.2.7.

4.2.2.6 Multiple Streams

The discussion so far has centered on the requirements of a single continuous media stream; however, this dissertation assumes that multiple streams will be used simultaneously. The use of multiple streams introduces two more requirements:

1. guarantees must be met in the face of multiple streams.
2. related streams may need to be synchronised with respect to each other.

The first requirement is not unique to continuous streams; guarantees must be met regardless of all other system activity. Therefore, any resource management scheme must cover *all* system resources

and not just network and protocol resources.

The synchronisation required for multiple streams is discussed in detail in section 4.2.8.

4.2.2.7 Summary

The continuous media requirements discussed above can be categorised as follows:

- those requirements which are related to the implementation of communications protocols and can be hidden behind the interface to these protocols. These include reducing latency, bounding jitter and increasing bandwidth.
- those which have wider implications and must be explicitly represented in the protocol interface. These include support for the synchronisation of multiple streams, media and application specific error recovery and QoS specification and negotiation.

The first category contains requirements for the design of communication protocols to support multimedia communication. This dissertation is primarily concerned with the construction of multimedia applications and therefore concentrates on the second category.

4.2.3 Information Media Component Requirements

The primary requirement of the IMC is that its constituent devices provide an effective interface for controlling and monitoring the behaviour of the information media supported. Sufficient flexibility and status information must be available to support a wide range of applications over the same IMC devices.

4.2.4 Communication Component Requirements

The continuous media requirements discussed in section 4.2.1 must be satisfied by the communications component. The communications component must also satisfy any communications related application requirements. It is argued that adherence to the following design guidelines will lead to a communications component satisfying the requirements made of it.

- reduce complexity.
- minimise multiplexing.
- support QoS specification.
- provide guaranteed performance.

As already stated in section 4.1.2, the complexity of existing protocols is a serious obstacle to their use over the next generation of high speed networks and to their use for multimedia communication.

4.2.4.1 Minimising Multiplexing

Multiplexing is an essential part of any communications system and protocol; however, the cost incurred by multiplexing is high and often neglected. The primary function performed by multiplexing is the sharing of communication resources. It allows for a single communication channel

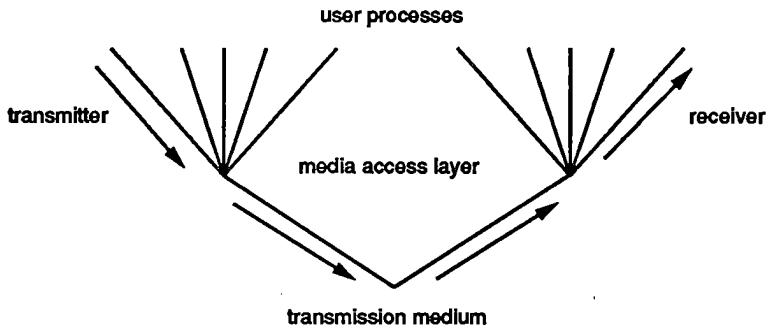

Figure 4.1: Minimal Multiplexing

to be shared between several higher level communication channels or user processes. This leads to a tree structure in which internal nodes represent a communication channel and leaf nodes a user process. The root node represents the transmission medium. On the transmission side, data flows from the leaf nodes to the root of the tree; at the receive side, data flows from the root to the leaf nodes.

If the arrival of communication requests is asynchronous and the available communications resources finite then there will inevitably be contention for the available resources. Under these conditions the form of multiplexing provided is referred to as being asynchronous and the following discussion assumes such asynchronous multiplexing. Synchronous multiplexing occurs when the arrival of communication requests is synchronous and can be controlled so as to avoid contention. A major consequence of asynchronous multiplexing is that the associated resource contention introduces variations in the time taken to process each communication request and therefore variations in the communication delay for each request. Such delay variations are referred to as *jitter*.

There are two points at which multiplexing *must* be provided. Namely, between hosts to allow a common transmission medium to be shared by these hosts and within each host to allow the points of attachment to the transmission media to be shared by an inevitably larger number of user processes; i.e. at the media access layer. Figure 4.1 illustrates the structure of this minimal amount of multiplexing.

Traditional, layered protocol architectures, as exemplified by OSI, introduce additional multiplexing beyond these minimal requirements. OSI requires multiplexing at six of its seven layers; the exception being the presentation layer, but even here it is suggested that multiplexing be implemented for "architectural consistency". The increased complexity of OSI multiplexing can be seen by comparing Figure 4.2 to Figure 4.1; given that each node is a source of jitter it is clear that the OSI stack is bound to introduce large amounts of jitter. Such excessive multiplexing is based on the assumption that communications channels are an expensive resource and must, therefore, be shared at every layer. Another argument made for multiplexing is that it preserves the functional independence of each protocol layer. The use of multiplexing in this manner allows the system to be scaled to support a large number of channels. As discussed in section 4.2.4.3 multiplexing is also often used to provide synchronisation between multiple, related, information media streams.

4.2.4.2 Multiplexing Disadvantages

Multiplexing inevitably introduces additional complexity and, as seen in section 4.1.2, complexity is a serious problem for multimedia communication. The disadvantages of layered multiplexing are

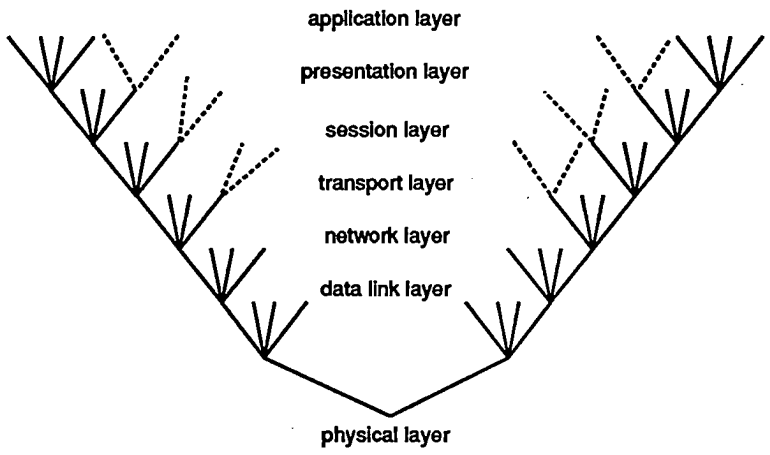

Figure 4.2: OSI Multiplexing

discussed in greater detail by Tennenhouse [**Tennenhouse89a**].

In a layered implementation each layer is commonly implemented as a separate thread of control and a synchronisation overhead is incurred when data passes from one layer to another; this synchronisation is an additional source of jitter and leads to even greater implementation complexity.

If several channels are multiplexed onto a single channel then traffic and delay variations in one channel may adversely affect the performance of the others; this is referred to as *performance cross-talk*. The presence of jitter increases the likelihood of such cross-talk, as each channel is more likely to experience sudden bursts of traffic. If the amount of jitter is large, then it may also lead to cross-talk between non-multiplexed channels. Cross-talk makes it impossible to provide communication guarantees.

Excessive multiplexing makes it very difficult to meet application specific communication requirements or QoS. If several higher level channels, each with their own QoS requirement, are multiplexed onto a single lower level channel, with a single QoS, then the QoS requested for the higher level channels is likely to be compromised. If several layers of multiplexing are used, then the original communication requirements may be completely lost by the time the data reaches the network. Such a loss of QoS has not been a problem for existing systems since they assume that all applications have the same communication requirements. This is no longer true, both from the application point of view, and for the emerging multi-service networks, which have been explicitly designed to offer a range of communication services. The compromise of QoS in this manner is referred to as *QoS cross-talk*.

Thus excessive multiplexing, and the jitter, performance and QoS cross-talk this generates, effectively destroy any notion of application specific communication requirements, communication guarantees and the opportunity to take advantage of advances in networking.

4.2.4.3 Multiplexing for Synchronisation

Synchronisation of multiple, related, continuous media streams is often provided by multiplexing the separate streams onto a single, order preserving, communication channel. The DASH CMEX server (section 3.10.2) and Leung's work (section 3.11.1), both use multiplexing for this purpose. Such a synchronisation scheme is conceptually simple and straightforward to implement. However,

in addition to the problems associated with the use of multiplexing discussed above, there are a number of other disadvantages related solely to the synchronisation of multiple media:

- a great deal of additional complexity is required to multiplex different types of media (e.g. voice and video) since they will use different, complex, and possibly compressed encodings.
- increased complexity may lead to excessive delay; therefore, although the related streams are synchronised with respect to each other, they may lose synchronisation with respect to real-time. That is, they may not be synchronised with respect to the human users of the system.
- the ability to apply media specific hardware assist is lost; video compression hardware may not be easily applied to both voice and video. The alternative of multiplexing pre-compressed streams introduces yet more complexity.
- multiplexing all media onto a single stream removes the opportunity for communicating each media stream over the communication network or channel best suited for it. The communication requirements for voice and video are very different and using a single channel for both is likely to be inefficient; similarly, different continuous media streams using the same network may require differing QoS.
- by definition, multiplexing serialises all of the streams being multiplexed and therefore removes any opportunity to use parallelism to increase performance.
- the media which can be synchronised (i.e. multiplexed), and their synchronisation relationships (i.e. the way they are multiplexed), are determined by the communication systems implementation. Adding new media and synchronisation schemes requires changing the communications component.
- the application has only limited control over the synchronisation of the multiplexed media.
- the ability to use application level knowledge to influence, and possibly, relax the synchronisation requirements is lost.
- all of the media to be multiplexed must either originate from, or be passed through, the same network source in order to be synchronised.

Given the immaturity of multimedia applications it is unlikely that a common set of synchronisation schemes can be devised and implemented within the communication system which can satisfy all potential application requirements. The disadvantages described centre on the lack of flexibility offered by this scheme, both in terms of the scope provided for experimentation at the application level, and for its inability to use application and media specific requirements and information to efficiently manage communication resources.

A commonly used justification for the use of multiplexing for synchronisation is that given the performance of current systems there is no alternative but to use multiplexing. Any designs making such an assumption should take care to allow for subsequent reduction in the use of multiplexing as system performance inevitably improves and thus increase the flexibility available to the application level.

4.2.4.4 Advantages of Using Multiple Channels

The use of separate channels to communicate separate information media offers a number of potential advantages as follows:

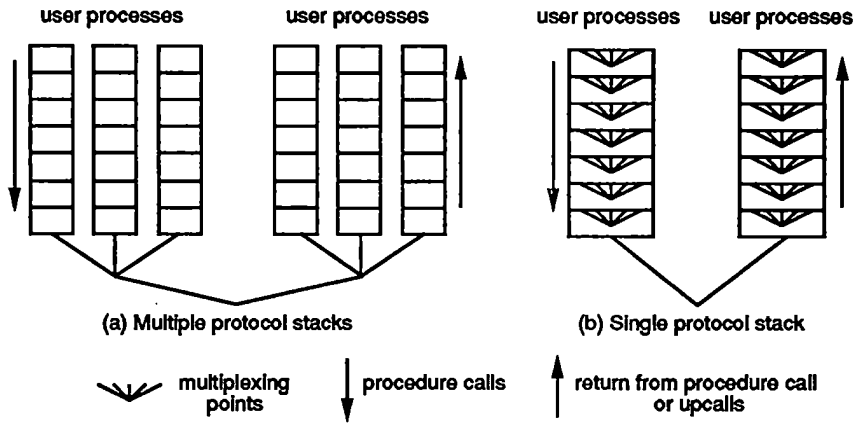

Figure 4.3: Multiple Versus Single Protocol Stacks

- separate channels may be processed and transmitted in parallel, thus increasing performance.
- allows the use of channels best suited to each medium, and thus allows for more efficient resource utilisation.
- media specific hardware is more easily applied to a single medium than to multiple, multiplexed media.
- allows the possibility of relaxing communication requirements based on application and media specific knowledge and requirements.
- preserves the modularity and separation of information media.

4.2.4.5 Guaranteed Quality of Service

In order to provide QoS and guarantee that a given QoS is maintained, some means is required for associating the QoS with the required communications channels and resources. Therefore, some form of end-to-end connection must be established which provides the resources to satisfy a given QoS. The notion of a *Lightweight Virtual Circuit*[9] (LVC), used in the UNIVERSE [**Leslie83**], MSN [**McAuley89**] and UNISON [**Tennenhouse89b**] architectures meets this requirement, in addition to minimising multiplexing. Other protocol architectures, such as DASH and MAGNET, use the same approach coupled with the pre-allocation of resources to provide guaranteed communications performance.

Both the MSN architecture and UNISON restrict multiplexing to the data link layer and by so doing effectively give communicating peers a private *instance* of the protocol stack. This is in contrast to OSI where all communicating parties share the same, single protocol stack. Figure 4.3 illustrates the difference between these two approaches and shows the points at which multiplexing is implemented. The first approach may require more memory but is computationally more efficient than the second, which conserves memory at the expense of poor performance and excessive jitter. In order to take full advantage of minimal multiplexing it is necessary to have a means of specifying the multiplicity of available protocol stacks and for identifying the protocol stack which best meets the application's communications requirements. The QoS facilities provided by IMAC, described in section 5.2.7, directly address these requirements.

[9] Often called an *association*.

4.2.5 Interactivity

The notion of interactivity provides a useful characterisation of the degree of real-time interaction required by a particular style of communication. For instance, a telephone conversation involves real-time interaction between the communicating parties, whereas an electronic mail message is asynchronous and involves no real-time interaction. Interactivity can be measured in terms of the delay between, or the amount of data communicated by, one party before the other party has an opportunity to reply. The amount of delay or data is expressed as a proportion of the total duration or data communicated. For the telephone conversation the delay is essentially instantaneous (given human reaction times), whilst for electronic mail the delay is the entire duration of the communication. Thus the degree of interactivity may be expressed as the inverse of the delay or data proportion, giving ∞ interactivity for the phone conversation and 0 for mail.

The amount of delay introduced by communication channels will affect the degree of interactivity which can be provided. A high delay wide-area network will reduce the interactivity of any digital voice communication implemented over it.

4.2.6 User Interface Requirements

There is a strong drive within the User Interface Management System (UIMS) research community towards more concurrent user interfaces and UIMSs which support this concurrency [**Lantz87**]. This drive is motivated by the general belief that concurrent input is a natural way for users to interact with computers [**Buxton86**][**Buxton85**][**Hill86**], and by the desire to build direct manipulation interfaces as described by Schneiderman [**Schneiderman83**]. Direct manipulation interfaces are characterised by concurrent input and the provision of timely positive feedback in response to user actions. Hudson [**Hudson87**] and Tanner [**Tanner87**] explore in detail the demands made on a UIMS by this type of user interface. An important requirement is that the feedback provided should appear instantaneous to the user, thus imposing a maximum response time in the order of 10-40ms (human perception threshold time).

Feedback must be provided, not only in response to the local users actions, but also in response to remote users actions and in response to errors. The error feedback generated must reflect the error in some meaningful fashion to the user, thus avoiding the situation where a user is left to stumble across the error in the normal course of his or her communication.

As a simple example consider the situation where a user is running the X window system. This user has a terminal connection to a remote machine, if the remote machine crashes no feedback is given; rather the user is left to determine that the remote machine crashed based on its lack of response. This is largely a result of the fact that the communications protocol used does not generate any indication that it is having difficulty communicating with the remote machine. This may not in itself seem a great hardship for the user. However, if more complex conferencing applications which support communication with multiple users using multiple media are to be built, then the provision of positive feedback becomes essential.

There is a strong requirement to allow users to customise their user interface to personal taste. As found by MMConf (section 3.8.4) such customisation must be carefully managed to avoid interference between conference participants. Similarly, users may wish to present the same data in different ways, e.g. one user may wish to see some data as a pie chart, whilst another prefers a bar graph. Again such functionality must be part of the initial design as it is very difficult to retrofit to an existing user interface.

4.2.7 Application Requirements

Application requirements fall into two categories: firstly, those which are common to all applications, but are exacerbated by the inclusion of multimedia, and secondly, those which are a direct result of multimedia. In the first category the requirements are as follows:

- increased productivity for the application writer. The primary tool for achieving such an increase is the re-use of modular application components.
- extensibility: the ability to add new functions in an incremental manner, whilst preserving the maintainability of the application.
- portability: applications should be capable of running over a wide range of platforms.
- distribution: distributed applications require heterogeneity management, efficient communication and effective handling of communication failures and partial failure. Effective error handling often includes a notion of graceful degradation.

Any architecture intended to make writing applications easier must address all of these requirements. However, the immaturity and lack of experience with multimedia applications means that common application components and concepts have yet to be identified and as a result there is little scope for component re-use. Multimedia applications may need to be extended to handle new, previously unforeseen media, in addition to media specific enhancements. Applications should also be easily extended to take full advantage of technological progress. Clearly, restrictive communications component design (e.g. using multiplexing to implement synchronisation) will make such extension difficult.

The heterogeneity present in a multimedia communication system is even greater than that found in a traditional distributed system. As seen in chapters 2 and 3 this heterogeneity occurs across hardware and software, and also in system configuration; i.e. not only are many different hardware and software components used, but they are used in different ways. Multimedia applications are often distributed and must therefore manage communication and partial failures. Failure handling is of primary importance in any interactive application and even more so for a conferencing application involving two or more participants.

Graceful degradation refers to an application's ability to offer restricted functionality in the face of partial failure; for example a conferencing system providing voice and video communication could degrade to voice only communication if video communication is lost.

4.2.7.1 Multimedia Related Application Requirements

The introduction of multimedia introduces a number of additional application requirements. Most information media are sufficiently different to one another to warrant special treatment; in particular communication and error handling requirements are media specific. These requirements also change from one application to another as media are used in different ways for different purposes. Continuity (section 4.2.2.3), is one such example. In some conferencing systems voice is used to control the floor and is the primary medium. Alternatively, a multimedia document editor may use voice as a secondary medium to recite annotations and may be replaced by a text annotation if voice is unavailable or not desired. The Etherphone project uses a different voice protocol for interactive real-time voice communication to that used for recording. The recording protocol requires additional reliability but is tolerant of greater latency.

The QoS parameters used to specify these requirements should be declarative rather than procedural; i.e. they should say *what* is required rather than *how* it is to be provided. They should

also be specified using application level terms and concepts, rather than communication level ones. For instance it should be possible to request a high quality voice channel rather than directly specifying the encoding, bandwidth, delay and jitter characteristics required to realise high quality voice communication. In this way, it is possible to shield the application from the details of how the desired communication is implemented. The mappings between the application level and communication QoS parameters must be maintained in some application component. The use of procedural parameters will lead to non-portable, non-reusable application components which are tied to particular implementations of particular protocols over particular platforms.

Given that a single QoS request may be satisfied in a variety of ways and that the QoS actually provided may influence the behaviour of the application, it is necessary for the application to be able to determine the QoS used to satisfy its original request. For example, a request for voice communication may be satisfied using a terrestrial network or using a satellite link, the large delay introduced by the satellite link is likely to affect the behaviour of the application, especially with regard to the degree of interactivity and synchronisation it provides. The application must therefore, be able to determine whether the satellite link is being used or not.

Given that a major use of an MMCS will be for personal communication there is a requirement to name end users rather than physical locations or machines. For example, the current telephone system and even electronic mail systems, name *locations* which are easily accessible by the intended user. This works well if the user does not frequently move from one place to another. It would be far more convenient to name the user directly and have the system take care of identifying the physical location of the user on demand. In this way, if a user a, moves from location x to location y, then the name a can be used to contact a at location y, or indeed any other location user a chooses to visit. This is called *user addressing.*

The final multimedia specific application requirement is the synchronisation of related media streams and is discussed in detail in the following section.

4.2.8 Multiple Stream Synchronisation Requirements

It is possible to identify three broad categories of synchronisation which are important for multimedia applications:

- event synchronisation: the synchronisation between the application and certain, well defined, events or states, (including errors), in the execution of the system.
- stream synchronisation: the synchronisation between the end-points of a single media stream and the synchronisation between multiple, related media streams.
- synchronisation with respect to real-time.

The first category is concerned with allowing for the effective control of system components, whilst the second deals with the synchronisation of multimedia communication. The third is concerned with assuring that related actions within the system occur at the *same* absolute time, and also that the synchronisation of operations within the system is synchronised with respect to external system events and in particular the user's notion of time. The following discussion assumes that the components being controlled execute in parallel and are asynchronous with respect to each other. The term *controlling application* may refer to a single thread of execution, or it may consist of multiple execution threads distributed across several machines.

4.2.8.1 Event Synchronisation

In order to allow for the effective control of system components it is necessary to identify points of interest in the execution of the components to be controlled. These points, referred to as *event synchronisation points*, provide an opportunity for the controlling application to synchronise its execution with that of the components being controlled. The following examples illustrate some likely uses of event synchronisation.

Consider an application which uses a remote camera as a video source and arranges for this video to be displayed in a window on a local workstation. Both the camera and the window system providing the local window execute concurrently with respect to each other and with respect to the application. The application must arrange for the window to be displayed *before* the video begins to appear and for the video to stop *before* removing the window. The appearance of the window represents a synchronisation point, i.e. the application must wait for this point to be reached before allowing any video to be displayed. Similarly, the application must wait for the video stream to cease being displayed *before* removing the window; the display of the video ceasing constitutes another synchronisation point. Note that the synchronisation point is defined to be when the video has actually ceased to be displayed, rather than when the camera has been told to terminate the video stream. This implies that the synchronisation point is reached within the local workstation window system rather than within the camera. The use of synchronisation points in this manner avoids the race condition between the camera being told to start and the window appearing, and between the camera being told to stop, the video stream actually stopping and the window being removed.

It is also useful to treat errors as event synchronisation points as this allows the controlling application to synchronise with the component reporting the error and thus determine the nature of the error. By so doing, the important application requirement of effective error handling may be satisfied. Finally, it will often be necessary to associate a timeout with any awaited synchronisation point in order to detect the failure of the component responsible for the synchronisation or any communications failures.

4.2.8.2 Stream Synchronisation

Section 4.2.2.4 discussed the synchronisation of a single stream and identified the likely causes of a loss of synchronisation as: lost or corrupted data, clock rate variations and jitter. This section is primarily concerned with the requirements for synchronising multiple, related streams.

It is possible to define a synchronisation spectrum ranging from no synchronisation at one extreme to a very high degree of synchronisation at the other. The degree of synchronisation refers to the amount of skew which may be tolerated between two synchronised streams, if this skew is exceeded then the streams are no longer synchronised. In a discrete digital system, skew is best represented as the granularity at which synchronisation is implemented; a fine granularity implies a small amount of skew is tolerated, and as granularity increases so does the amount of tolerable skew. In other words, the smaller the sample size, the greater the degree of synchronisation. Sample sizes, and therefore, granularity, can be measured in terms of the amount data present in each sample, or the amount of time it takes to create (i.e. digitise) each sample. In an interactive system, time is often the most useful measure of synchronisation. Absolute synchronisation can never be achieved. However, if synchronisation is implemented at the finest granularity supported by the system in question, usually at the hardware sample level, then it will be referred to as being *complete*. Figure 4.4 illustrates this spectrum with increasing synchronisation from left to right.

The style and granularity required are entirely application dependent and any proposed synchronisation scheme must support a diverse range of synchronisation requirements. As already shown

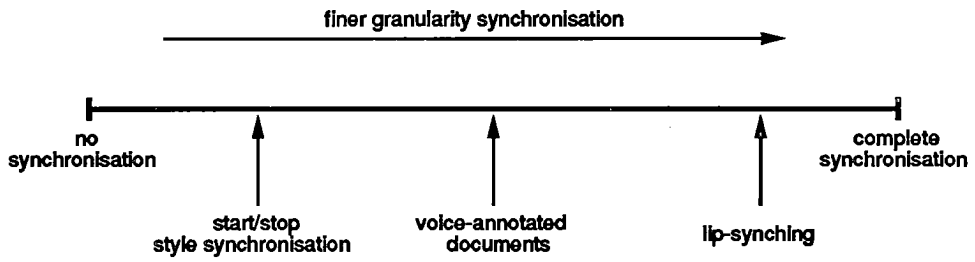

Figure 4.4: Synchronisation Spectrum

in section 4.2.4.3 multiplexing cannot provide the level of flexibility required.

"Lip-synching" is the most obvious example of multi-stream synchronisation and requires synchronisation to within $\pm 100ms$ between the voice and video streams. Other examples include synchronising voice and cursor movement in a conferencing application, or voice and the graphical display of that voice in a voice editor. Multimedia document systems and editors must synchronise displayed text and graphics with any annotating voice. The degree of synchronisation required varies widely between these applications and also different users may prefer different levels of synchronisation for the same application.

In order for any multi-stream synchronisation to be implemented there must be a way of relating the contents of one stream to the contents of another. Therefore each stream must exhibit some form of structure which can be used to determine the granularity of the synchronisation required. Video is typically structured as *frames*, whilst voice is often treated as a series of silence delimited *talk-spurts*. The structure chosen for a particular stream should allow for subsequent synchronisation with as wide a range of other information media as possible.

Given that streams are structured, it is then necessary to relate components of one stream to components in another. For non-isochronous streams a simple sequence numbering scheme can be used. Isochronous streams require that the sequence numbers be extended to include some notion of time; this in turn may require the provision of a synchronised clock across all the stream sources and sinks.

The principal causes of a loss of multi-stream synchronisation, in addition to the causes of single stream synchronisation, are performance and QoS cross-talk and partial system failure. Therefore cross-talk must be minimised and failures detected and handled accordingly.

Despite all attempts to reduce the likelihood of a loss of synchronisation, there still remains a finite possibility that such a loss will occur. Therefore, support must be provided for detecting such losses and for taking some action in response to a synchronisation loss. This action may simply consist of reporting the error, or it may be possible to take some corrective action to restore synchronisation. The nature of any corrective actions is media and application specific. These requirements impact on the communications component and require that it provide a means of monitoring synchronisation and for taking corrective action to restore synchronisation following a loss. Possible corrective action includes the provision of additional buffering, altering process and network access priorities and changing QoS requirements.

4.2.9 Real-Time Synchronisation

Real-time synchronisation refers to the process of synchronising operations and events within a system with respect to an external time scale. The term "real-time" is used to convey the fact that the external time scale is outside of the control of the system in question.

The communication delays inevitably involved in *observing* time make it impossible for multiple observers to agree, with absolute precision, on the current time. Therefore, it is impossible to maintain a single, global clock across a distributed system, whereby all system components see the same time. The best that can be achieved is to impose a bound on the skew between individual views of the global clock.

Multimedia communication systems, although having strict synchronisation requirements, cannot be classified as being *hard real-time systems.* A hard real-time system imposes a maximum delay for responding to external events, which if exceeded constitutes a possibly catastrophic system failure; fly-by-wire aircraft control systems are a good example of hard real-time systems. If an MMCS loses synchronisation, then communication quality may be degraded or lost altogether - in the vast majority of cases this will not be a disaster.[10]

MMCS's require real-time synchronisation for two purposes:

- to ensure that two or more operations occur at the same time, or more correctly, that these operations occur within the time range delimited by the allowed clock skew.
- to provide synchronisation for isochronous media, and in particular to provide a means for relating components of one isochronous stream to components of another.

Finally, for an MMCS providing communication with, and between human users, the acceptable amount of real-time synchronisation skew is determined by human reaction and perception times.

4.2.10 Distributed Processing Component Requirements

The DPC requirements are heavily influenced by the application and user interface component requirements; the DPC inherits many of its requirements from these other components. Functions provided by a DPC typically include some form of RPC system, lightweight threads and associated synchronisation primitives and naming. The DPC will provide a homogeneous interface to these functions even though they are implemented over heterogeneous software and hardware; management of heterogeneity is therefore an important MMCS function. It is the DPC interface which must be extended to meet the new requirements made by the application and user interface components.

The requirements that the DPC, its interface and implementation must satisfy are as follows:

- effectively manage the heterogeneity found in multimedia communication systems.
- provide a QoS based interface to the underlying communications component.
- the implementation of the DPC must maintain the guarantees made by the communications component;[11] i.e. those guarantees must be met on an end-to-end basis.

[10] Although, it is possible to envisage medical, air traffic control or vehicle guidance applications where maintaining communication quality is of greater importance.

[11] Or at worst distort these guarantees in a known fashion.

- provide sufficient flexibility for the control of a wide range of multimedia devices.
- support event, single and multiple media stream synchronisation.
- allow the degree of real-time processing implemented by the application to be varied, (see below).
- support scaling, both in terms of the number of hosts present in the system and in terms of increased physical separation between hosts.

Section 4.1.3 suggested that a compromise is required between rich programming and run-time environments and real-time performance. Chapter 3 showed that many existing systems separate the real-time from the non real-time components in an attempt to find such a compromise. Therefore, the DPC is required to support such partitioning of applications. This partitioning must be sufficiently flexible to allow for the inclusion of new functionality and new information media, and also to take advantage of technological advances.

4.2.10.1 Open Systems Requirement

The term "open systems" is used in a variety of ways to describe different system properties. The two most commonly used definitions are discussed before defining the usage of open systems within this dissertation.

The Open Systems Interconnection (OSI) Reference Model uses the term "open" to "emphasize that by conforming to OSI standards, a system would be able to communicate with any other system obeying the same standards anywhere in the world" [**Day83**]. That is, an open system is a system which allows heterogeneous system components to communicate with each other using a set of standards.

Lampson and Sproull [**Lampson79**] provide a somewhat different definition of an open system. In their definition an open system is one which offers a variety of facilities, each of which may be used in conjunction with, or in isolation of, one another. However, wherever a given facility is built from a set of existing facilities the existence of this compound facility should not preclude the use of its constituent parts.

These definitions turn out to conflict when practically applied. The OSI definition leads to a layered system within which each layer implementation may be substituted by another functionally equivalent implementation without affecting any of the other components. It is not allowed to bypass one layer and use an underlying layer directly; this lends itself to very inefficient implementations. Whereas, the Lampson and Sproull definition leads to an essentially hierarchically structured system, within which it is possible, and indeed encouraged, to make use of compound components as constituent components. The drawback with this approach is that modifications to a single constituent component may require changes to the possibly large number of components which make use of it. However, if the most commonly used components are standardised, as advocated by OSI, this drawback can be avoided.

The definition used within this dissertation attempts to capture the advantages of both of these definitions whilst avoiding the drawbacks. This is done by advocating a hierarchically structured system with standardised common components or facilities. The definition used is as follows:

> An open system is one which allows the interoperation and interconnection of heterogeneous systems and system components and can be incrementally extended by the addition of new system components without disturbing the existing system components.

This definition is intended to characterise systems which are extensible and flexible, in addition to adhering to standards for the interconnection of independently developed systems and system components.

The requirement for an open system is added to the list of all other DPC requirements.

4.3 Summary

This chapter has discussed the problems presented by the design and implementation of existing systems to the incorporation of multimedia communication and applications. These problems are largely a result of the fact that multimedia is a recent development and existing systems were simply not designed to support multiple information media. The high data rates and real-time performance required by voice and video are particularly problematic.

In order to understand how to design new systems capable of overcoming these problems the requirements made by the incorporation of multimedia were discussed at length. Particular attention was paid to the communication and synchronisation requirements of the new media to be incorporated. In order to overcome the lack of functional integration exhibited by existing systems seen in chapter 3, the requirements made by multimedia on *all* system components have been discussed.

The full list of requirements is long and satisfying all of them would require greater resources than those available for this dissertation. Therefore, the IMAC architecture and its prototype implementation concentrate on the following subset of these requirements:

- synchronisation of multiple, related, media streams.
- management of heterogeneity.
- support for media and application specific requirements, in particular:
  - QoS specification and negotiation.
  - error handling and recovery strategies.
  - greater application level control of the underlying communication system.
- to provide an architectural framework within which the remaining requirements may be satisfied.

The choice of these requirements is based on the observation that, as yet, little or no work has gone into addressing them, and that the absence of satisfactory solutions represents a serious obstacle to continued progress in the field of multimedia communication applications. Finally, by taking the remaining requirements into account when designing the architecture and prototype implementation it is possible to provide a framework within which they may be satisfied at some future point.

5

The IMAC Architecture

This chapter presents a detailed description of the Integrated Multimedia Applications Communication architecture (IMAC) and the principles guiding its design. Many of the ideas presented in this chapter were first described in [**Nicolaou90**].

5.1 Architectural Principles

The process of conducting the research survey presented in chapter 3, and that of researching the problems and requirements made by multimedia discussed in chapter 4, led to the formulation of the principles presented in this section.

The principles listed below and described in the following sections are the principal ones to have influenced the design of the IMAC architecture:

- Media Separation
- Modularity
- Choice
- Evolution

In recognition of the inherently distributed nature of multimedia communication systems, IMAC has been based on the ANSA architecture. Consequently, IMAC inherits many of the distribution and scaling properties of ANSA; the relationship of IMAC to ANSA is described in section 5.2.2.

Finally, section 5.1.5 discusses the relationship of Functional Integration to the architectural principles presented in this section.

5.1.1 The Principle of Media Separation

The principle of media separation states that, wherever possible, a single information medium should be stored, communicated and manipulated separately from other information media. An immediate consequence of this principle is that each medium must provide an interface for controlling its behaviour; that is, a single information medium consists of a representation, or data, component and an algorithmic component for implementing its control interface.

The principle of media separation supports the construction of open systems through the ability to add new media, and remove existing media, without unduly disturbing any other media. Also, given that some form of multiplexing must be used if multiple media are to be treated as one, then adherence to the principle of media separation minimises the use of multiplexing and thus avoids the associated disadvantages discussed in section 4.2.4.1. The media specific interfaces constitute a point at which system components interoperate and are therefore candidates for standardisation.

The granularity of the media that can be separately supported is inevitably an implementation consideration and will change as technology progresses. Some system implementations may find it impossible, largely for performance reasons, to keep certain media separate. For instance, the voice and video components of a "video-phone" conversation may be treated as a single medium to maintain "lip-synch"; that is, they may be multiplexed over a single communications channel and stored in a single file. If the merged media are in turn treated as a single compound medium with respect to the other information media present in the system then media separation is still provided but at a coarser granularity.

The interface provided by each information medium must be capable of supporting the synchronisation requirements discussed in section 4.2.8. The advantages offered by media separation with regard to the synchronisation of multiple media centre on the ability to change the set of media being synchronised and to alter the nature of the synchronisation being provided, in response to application requirements and without having to modify each individual medium.

5.1.2 The Principle of Modularity

A modular system is one whose functions can be naturally divided into coherent components that can be separately developed and maintained. Modular systems are commonly accepted as being easier to design, implement and maintain than non-modular, monolithic, systems. They also support the extension of system functionality by the addition of new modules and also offer greater flexibility through the re-use of existing modules in previously unforeseen ways and combinations. It is much easier to change the *configuration* of a modular system than a non-modular one; that is, it is possible to change the physical and logical structure of the system without affecting its constituent modules.

A criticism sometimes made of modularity is that it reduces system efficiency; however, it is the mechanism used for communication between modules which reduces efficiency and *not* modularity itself. Therefore, a modular system requires the use of efficient inter-module communication. If a given mechanism is too inefficient then it may be possible to change the configuration of the system to use a more efficient one; for instance, co-locating client and server modules on the same machine if network communication is too slow.

Modularity, whether it be in the form of layers, or a more general structure, is an essential principle for the construction of open systems. The principle of media separation discussed in the previous section is an example of the application of modularity to information media.

5.1.3 The Principle of Choice

When designing and implementing any computer system a series of decisions have to be made between a range of possible options, and in particular, between a set of different policies. The anticipated use of the system in question typically forms the basis for making these decisions. The principle of choice states that such decisions be made at the most appropriate times and not any earlier than this. For instance, the communication system should not be designed to provide a narrow set of anticipated communication requirements and thus constrain applications to these anticipated requirements, regardless of whether they are appropriate for the application or not. This implies that the system design and implementation must be sufficiently general and flexible to support as wide a range of options as possible. The amount of choice which can be provided is constrained by the other principles of media separation and modularity; that is, providing too much, or inappropriate, choice may lead to a loss of media separation or modularity.

The principle of choice is intended to ensure the construction of systems which can support the variety of diverse requirements made by multimedia applications discussed in chapters 3 and 4. The separation of policy and mechanism is closely allied to the principle of choice, and forms the basis for providing choice; a system design and implementation must endeavour to provide a set of mechanisms to which an application may apply a policy of its own choice.

As argued in section 4.2.7.1 the mechanism provided for specifying the choice being made should be *declarative*, rather than *procedural*.

5.1.4 The Principle of Evolution

As seen in section 2.3, and chapters 3 and 4, multimedia systems cover a broad range of multimedia hardware and network integration. In addition, the level of integration achieved for one medium is often different to that achieved for another; for instance, a greater level of integration is currently possible for voice than for video. The rapid pace of technological progress and the inevitable delay between some function becoming available and it being integrated into a general purpose system, suggests that systems of varied levels of integration will always exist.

It is possible to discern an evolutionary path from mixed analogue and digital systems using separate hardware and networks for different information media, all the way to fully integrated, all digital, systems using the same general purpose processors and networks for all media and applications.

If an architecture for multimedia communication systems is to be successful then it must allow for the gradual evolution from one level of integration to another. In addition it must be able to cope with the various levels of integration currently in existence and exhibited by different media.

5.1.5 Functional Integration

Functional integration, as defined in section 2.4, is concerned with the effective interoperation of system subcomponents to provide the functionality of the system as a whole. The lack of functional integration, identified as a major weakness of existing multimedia systems, shows itself as a mismatch between the functionality provided by one component and that expected of it by other components.

Functional integration is, therefore, not so much an architectural principle as a goal to be achieved by the judicious application of the principles discussed above. The principles of media separation, modularity, choice and evolution must be applied in a way which increases functional integration.

5.2 The IMAC Architecture

This section presents the IMAC architecture in detail. Not only is the architecture itself defined, but its principal features, and the design options and decisions which led to these features, are also discussed and justified. That is, this section not only describes *what* the IMAC architecture is, but *why* it is the way it is.

5.2.1 Assumptions

The principal assumption which pervades this dissertation is that a multimedia communication system (MMCS) is inherently distributed and can be viewed as an extension of existing distributed computing systems. An MMCS must, therefore, manage all of the problems that distribution entails. Similarly, this assumption requires that existing distributed systems be extended to meet the requirements made by multiple information media, and in particular, by continuous media.

A consequence of distribution is that separate information media may be implemented using physically separate processors, interconnected by one or more shared communications networks. In addition, the applications making use of these media may execute on separate processors from the media, again interconnected by shared network(s). This means that separate media streams and their controlling applications may execute in parallel and asynchronously with respect to each other.

IMAC assumes that all devices will be able to support an ANSA interface, in practice this requires that such devices be able to support an instance of the ANSA Testbench. This assumption is justified as follows:

- Etherphones (section 3.6.1) had sufficient intelligence, circa 1983, to run the Cedar run-time environment which has similar requirements to the Testbench.
- the same CPU cards used for the ISLAND Ringphones (section 3.6.2) are now in use running the ANSA Testbench over the Wanda[1] operating system. Given that the processing overhead for relatively infrequent control operations is low, then there is no reason to believe that ISLAND could not be re-implemented using the ANSA Testbench.
- the multimedia network interface under construction at Lancaster (section 3.7.5) uses an instance of the Testbench to implement device control.
- finally, both CPU speeds and memory sizes have increased significantly since the Etherphone and ISLAND phones were implemented and as a result it is fair to assume that an ANSA interface can be provided by such devices.

5.2.2 Overview and Relationship to ANSA

As a direct result of the distribution assumption IMAC was based on the ANSA architecture (see appendix A). Familiarity with the ANSA architecture is assumed in many of the subsequent sections. IMAC inherits the following features from the ANSA Computational Model:

- ANSA definition of interfaces, interface types and interface type conformance.
- ANSA invocation and parameter passing scheme.

[1] Wanda is a light-weight, thread-based, kernel in use at the University of Cambridge Computer Laboratory.

- the ANSA concept of interface trading.

ANSA is based on the client/server model of interaction. Interfaces represent the point of service provision and define the service being provided; interfaces contain definitions for a set of operations which may be invoked to use the service being provided. In order to detect programming errors as early as possible, services are typed and a client must specify the types of the services it wishes to use. Trading provides the mechanism for matching client requests for a service with an available service. A database is provided for storing available services; servers must *export* the services they provide to this database, and clients then *import* services from this database. Interface type conformance defines the algorithm used to match client requests, which are specified in terms of interface types, with the available interface types. These functions are implemented by a service called the *Trader*.

Care has been taken to preserve the minimal nature of the ANSA architecture - extensions and modifications have been kept to a minimum. The new concepts introduced by IMAC are as follows:

Streams: (section 5.2.3), the notion of a stream is introduced to represent a single, continuous, information medium, and in particular the communication of such a medium. The representation used has been chosen to explicitly support the synchronisation of multiple, related streams to each other, and also to allow applications to synchronise their execution with that of the stream.

Stream Types: (section 5.2.3), streams are unidirectional and connection oriented. Stream sources are called *plugs*, and stream sinks are called *sockets*. Plugs are connected to sockets to create end-to-end streams along which information media may flow. Stream plugs and sockets are typed and only like types may be interconnected.

Devices: (section 5.2.4), an extension of an ANSA service to include one or more stream end specifications. The combination of interface operations and stream ends is intended to represent devices such as video cameras, framestores, microphones and speakers. More complex, compound, devices can be constructed from multiple devices, provided that it is possible to synchronise the operations of the grouped devices. Devices are responsible for the generation, communication and presentation of a particular set of information media.

Device Types: (section 5.2.5), ANSA services are typed, as are streams, thus making it an easy matter for devices to be typed. A device type is the combination of its underlying ANSA service, i.e. its operation component, and its stream types. By defining a conformance relationship for devices they may be *traded* in the same manner as ANSA interfaces.

Quality of Service: (section 5.2.7), all operations in IMAC may specify a set of QoS options with which they are prepared to be invoked. At invocation time the caller may specify the QoS required for the current call. Such QoS options are expressed as constraints on the QoS which can be provided by the underlying implementation.

Orchestration: (section 5.4), refers to the management functions required to coordinate IMAC streams, devices, QoS and services (defined below).

A number of new architectural services are defined by IMAC in addition to those defined by ANSA; these are as follows:

QoS Manager: (section 5.3.1), provides a system wide database for the available QoS options and may be used to provide compile-time checking of QoS options.

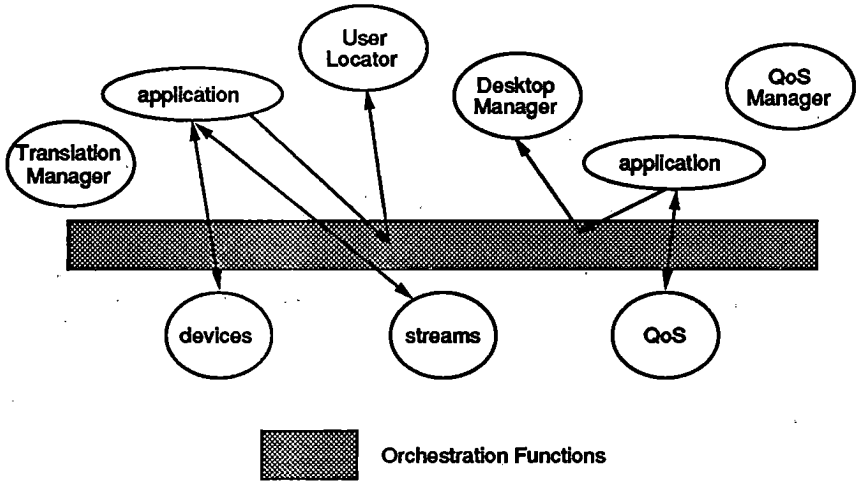

Figure 5.1: Application View of IMAC

User Locator: (section 5.3.2), is responsible for identifying the current physical location of a given user, and is intended for use by applications which require user addressing. Given a user name it will determine and return an address for that user which can be used for subsequent communication.

Desktop Manager: (section 5.3.3), the Desktop Manager provides a management interface for controlling the various devices and services provided by a single Multimedia Desktop, (see section 2.5). The ANSA Node Manager[2] (see section A.12) provides much of the functionality required. However, it lacks the ability to create and destroy services in multiple run-time environments (i.e. each Node Manager assumes a single such environment) and to manage shared resources.

Translation Manager: (section 5.3.4), provides a mechanism for resolving stream type mismatches via the insertion of stream translators.

It is possible to revisit the functional decomposition of an MMCS given in section 1.1.1, and to assign IMAC functions to MMCS components as follows:

Information Media Component (IMC): IMAC streams and devices.

Communications Component: IMAC QoS and QoS management.

Distributed Processing Component (DPC): orchestration and the features inherited from ANSA.

Application Component: services such as the User Locator, Desktop Manager and Translation Manager. Any applications built using IMAC.

User Interface Component (UIC); in order to allow for the widest possible range of user interfaces to be supported, IMAC does not provide any functions specifically for the construction of user interfaces. This is an application of the principle of choice, whereby decisions regarding the design of user interfaces are left to user interface designers. However, the other components of IMAC have been designed to satisfy the requirements which will be made by multimedia user interfaces.

[2]Designed and implemented by the author to replace a simpler design due to Hugh Tonks.

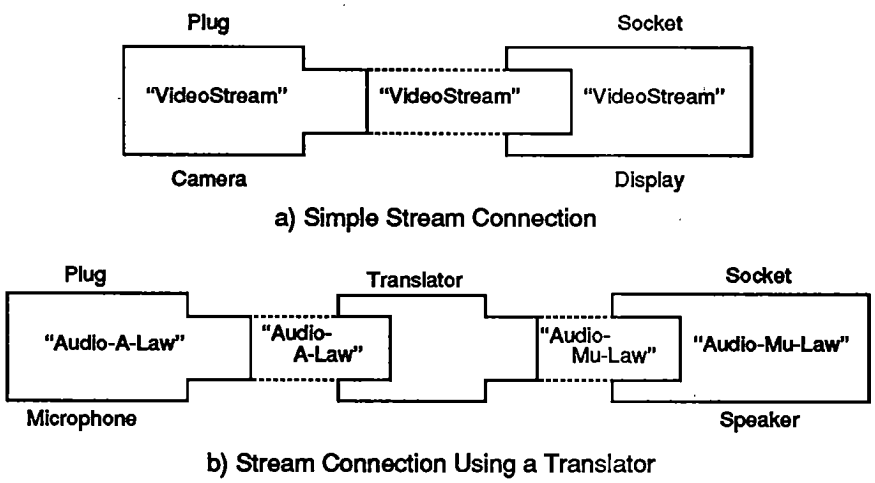

Figure 5.2: Stream Connection

IMAC applications are presented with a uniform interface, via a series of language extensions and libraries, to the functionality provided by IMAC. The coordination of the activities of separate services, and the interface to these services, is provided by orchestration functions and is a fundamental component of the interface seen by applications. This structure is illustrated in figure 5.1.

5.2.3 Streams

Streams are used to represent a single, continuous, information medium and in particular the flow of information media data from one location to another. The source is called a *plug* and the sink a *socket*. Stream plugs and sockets are typed as defined in section 5.2.4, and are jointly referred to as *stream ends*. Two plugs, two sockets, or a plug and a socket, are of the same type if, and only if, they have the same textual name. In addition plugs or sockets, may be designated as being *multiway*, in which case they may transmit to multiple sockets, or receive from multiple plugs respectively. The type of a multiway plug or socket applies to *all* of the streams it may source or sink.

A plug must be connected to a socket in order for any data to flow. Such a connection can only be made between plugs and sockets of the same type. In the case of multiway plugs and sockets, connections may be incrementally created or destroyed, again provided that the plug and socket are of the same type.

Streams provide the flexible connectivity required by multimedia applications, whilst satisfying the principle of media separation. The typing of streams ensures that programming errors are detected as soon as possible and may also allow for the automatic insertion of *translator* devices between plugs and sockets of incompatible types. A translator must have a socket of the same type as the original plug, a plug of the same type as the original socket, and must convert the data read on its socket into a form suitable for transmission over its plug. For example, to connect a plug of type `Audio-A-Law` to an `Audio-Mu-Law` socket, a translator device with an `Audio-A-Law` socket and an `Audio-Mu-Law` plug would be inserted between the original plug and socket. Translator devices are identical to any other device except that they are explicitly identified as candidates for resolving such type incompatibilities. The translation manager (see section 5.3.4) is charged with managing the insertion of translators. Figure 5.2 illustrates stream and translator interconnection.

The requirements for the synchronisation of multiple, related, information media were discussed in section 4.2.8. Three forms of synchronisation were identified: namely event, stream and real-time synchronisation. Section 5.2.3.1 defines the support for event synchronisation. IMAC supports stream synchronisation with the single concept of a *structured stream* and re-uses the mechanism used to implement event synchronisation for stream synchronisation; structured streams are described in sections 5.2.3.2 to 5.2.3.5. However, as justified in section 5.2.3.6, IMAC explicitly does *not* define a means for providing real-time synchronisation.

5.2.3.1 Event Synchronisation

As seen in section 4.2.8.1 event synchronisation refers to the requirement for an application to synchronise its execution with that of another system component. Such synchronisation is usually required in response to some exceptional condition or error. Stream synchronisation can be viewed as a particular instance of event synchronisation.

Exceptional conditions include stream termination, end-of-file, or a user terminating some communication with another user. Errors include communication and network failures, and partial failures as exemplified by a loss of communication between one application component and another, or the independent failure of a particular stream or application component with respect to other components. Exceptional conditions and errors are jointly referred to as *event synchronisation points.*

In order for a stream socket to be able to determine the difference between stream termination and a communication failure, all stream plugs must ensure that an end-of-stream indication is transmitted to the receiving socket. This indication may take the form of a token sent in the data stream, or may be provided by the underlying communication protocol.

There are three candidate methods for indicating to a controlling application that an event synchronisation point has been reached:

1. asynchronous events: data or tokens which are generated when a synchronisation point is reached and communicated, asynchronously, to any controlling application.
2. an outstanding invocation whose termination indicates the occurrence of a synchronisation point. Such invocations must be issued far enough in advance of the synchronisation point to ensure that it is not missed.
3. callbacks: when a synchronisation point is reached an operation supplied by the application is invoked, thus synchronising the source of the synchronisation point and the application.

The callback and outstanding invocation mechanisms are preferable to asynchronous events because they allow for synchronous communication between the stream and application, and therefore provide a means for synchronisation to be actually implemented. In any case, events may be easily implemented using callbacks or outstanding invocations, whilst the converse is not true.

In a distributed environment outstanding invocations are inherently more expensive to implement than callbacks because the invocation is likely to be outstanding for long periods of time, consuming communication resources all the while. The callback on the other hand is only invoked *when* the synchronisation point is reached and consequently will execute for a much shorter time. Therefore, callbacks are preferred to outstanding invocations. Note that both mechanisms can be implemented using ANSA invocations, the distinction lies entirely in how the invocation mechanism is used.

In this model an application, or another stream, provides an interface containing operations to be invoked by the stream being controlled, as and when, synchronisation points are reached. This

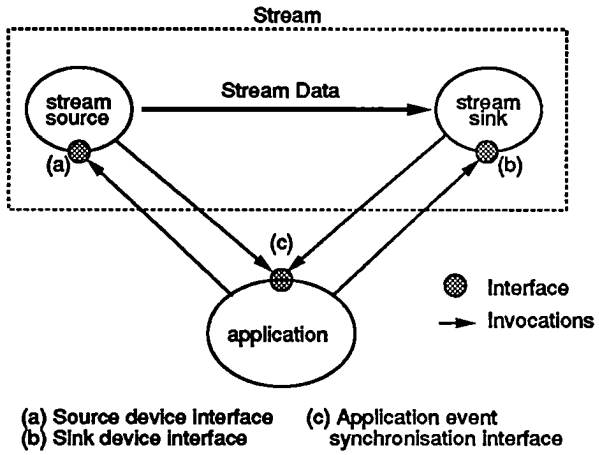

Figure 5.3: Invocation Structure for Event Synchronisation

relies on the ability to pass references to interfaces from one application or stream to another. In this way, applications play a client role when controlling streams and devices, and a server role when receiving synchronisation invocations; figure 5.3 illustrates this invocation structure.

It is possible to use asynchronous *announcements* and synchronous *interrogations* as supported by the ANSA Computational Model. Interrogations synchronise the caller and callee for the duration of the invocation, but as soon as the invocation terminates synchronisation is no longer guaranteed. Announcements may be used for operations which do not return any results and for which the caller does not wish to be blocked until the invocation is complete; the caller is only blocked for the time it takes to issue the invocation. However, no attempt is made to ensure the reliable delivery of announcements. Announcements may be directly used to represent unreliable events. The arguments passed to either form of invocation can be used to indicate the nature of the exception or error represented by the event synchronisation point. The use of ANSA operations also provides a convenient means for typing stream synchronisation points; the type of a stream synchronisation point is the type of the operation invoked when it is reached.

If more than one application is interested in the same exception then some form of multicast[3] protocol is required. Being an extension of a distributed system, and in particular of ANSA, IMAC is able to accommodate the use of a suitable protocol as it becomes available.

5.2.3.2 Stream Synchronisation

IMAC streams provide a novel means for synchronising related information media. Although streams will often be used to represent continuous media, this need not always be the case, similarly the media to be synchronised need not be restricted to continuous media. It may prove useful to represent a non-continuous medium as a stream for subsequent synchronisation to a continuous medium represented as a stream. If a medium is not represented as a stream, it may still be synchronised to stream-based media; however, any tools provided for automating stream synchronisation may not be applicable to such non-stream based media.

In addition to the features described in the previous section, IMAC streams contain a data component representing the flow of data from source to sink, which is broken up into a series of records or

[3] Also often called a *group* protocol.

frames which are visible at the stream interface. These records are called *Synchronisation Frames.*

The end of each synchronisation frame constitutes a *stream synchronisation point.* A stream synchronisation point represents a point in the execution of the stream, at which a controlling application, or another stream, may synchronise its execution to that of the original stream. Therefore, the same scheme used for indicating that event synchronisation points have been reached can also be used for stream synchronisation points; that is, an operation supplied by the controlling application is invoked when the appropriate stream synchronisation point is reached. For maximum flexibility separate interfaces should be used for event and synchronisation operations.

5.2.3.3 Logical and Physical Synchronisation Frames

The lowest level, not normally visible at the stream interface, consists of *Physical Synchronisation Frames* (PSF's). PSF's represent the lowest level data samples used within devices and by communication protocols between devices. The PSF for a video stream might be a scan line, whilst for an audio stream a $2ms$ audio sample might be used.

The synchronisation frames introduced above, are more properly called *Logical Synchronisation Frames* (LSF) and are computed from a number of PSF's. LSF's are visible at the stream interface. The number of PSF's required to construct an LSF need not be fixed, that is, it may vary from one LSF to another.

The mapping from LSF's to PSF's may be one-to-one, essentially exposing PSF's at the stream interface. Alternatively, arbitrary amounts of processing may be applied to constructing LSF's from PSF's, thus allowing very high level LSF's to be provided. For example, if voice recognition hardware is available it may be possible to provide LSF's which represent recognised words, whilst the underlying PSF's represent digitised speech samples. In most cases an intermediate amount of processing will be applied, for instance a video stream with a scan-line PSF, is likely to use a video frame for its LSF, where each LSF maps to 625 PSF's.[4]

The computation required to construct LSF's from PSF's is provided by the stream implementation, and in this way the encoding and internal media representation are isolated within each stream.

By varying the number of PSF's required to construct an LSF it is possible to vary the granularity, and therefore the degree of synchronisation provided (section 4.2.8.2) and in this way determine the point in the synchronisation spectrum that this stream occupies. Figure 5.4 illustrates some possible LSF to PSF mappings.

This two-level structure has been designed to support arbitrary degrees of synchronisation, and it allows applications to state, precisely, their synchronisation requirements in terms of LSF's and to rely on the underlying stream implementation to deal with constructing LSF's from PSF's. Ideally each stream will offer a range of LSF to PSF mappings and applications will be able to choose the most appropriate mapping, and therefore degree of synchronisation, for their current needs.

An important advantage offered by LSF's is that new media can be added to, and existing ones removed from, the set of media currently being synchronised without affecting the remaining members of the set.

[4] This assumes the use of PAL.

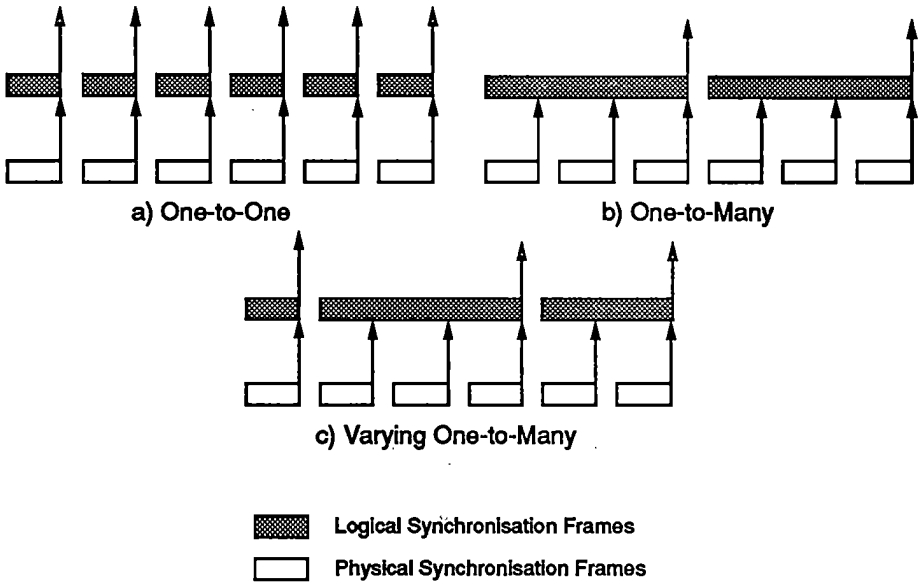

Figure 5.4: Logical to Physical Synchronisation Frame Mappings

5.2.3.4 Measuring the Degree of Synchronisation

LSF's provide a convenient metric for quantifying the degree of synchronisation; that is, the amount of acceptable skew between synchronised streams can be expressed in terms of the number of LSF's. If skew between any two streams exceeds this acceptable value then synchronisation has been lost. Such a loss of synchronisation can be detected by monitoring the synchronisation points encountered by each stream.

Depending on the nature of the LSF to PSF mapping, the number of LSF's may be expressed:

- in terms of time.
- in terms of data.
- only in terms of the LSF's themselves.

For the video stream example used above, each LSF can be expressed in terms of time or data; the time metric is given by the inverse of the frame rate, whilst the data metric is the amount of data used within each frame. If compression is used within each frame, then the data metric is no longer applicable, whilst the time metric remains valid. The audio stream example, with LSF's representing recognised words, cannot be quantified in terms of time nor in terms of data, since each LSF may take an arbitrary amount of time to compute and data to represent. For such a stream, the skew can only be expressed in terms of LSF's, in this case recognised words.

Streams whose LSF's can be directly quantified in terms of time are referred to as *isochronous*. A single stream may support one LSF which is isochronous and another which is not, thus reinforcing the distinction between the terms continuous and isochronous made in section 4.2.1. That is, continuity is viewed as a fundamental property of a medium, whereas the term isochronous is used to refer to how a communication medium is currently being used.

5.2.3.5 Identifying Logical Synchronisation Frames

The arguments passed to the synchronisation operation can be used to carry arbitrary information describing the state of the stream. Typically the information conveyed will include an indication of the LSF which has just ended, plus any other information required to relate this LSF to any other LSF in this, or in another, stream. The identification scheme chosen, and therefore, the arguments passed, must contain sufficient information to support the degree of synchronisation required, and in particular to allow LSF's from one stream to be related to LSF's from another. For isochronous streams, sufficient information must be included to allow synchronisation with respect to real-time.

Although the range of possible identification schemes is large and varied, there are a number of generally applicable guidelines as follows:

- LSF synchronisation point operations should be invoked from as near the stream sinks as possible, therefore such operations will often be associated with stream sockets.
- stream plugs must insert sufficient information in the data they send to the stream socket to allow the socket to identify each LSF.
- sockets will typically include some local information to indicate the current state of the stream and LSF.

Any values returned by a synchronisation operation may be used by the stream to influence its future execution. In this way the controlling application may control the stream via the results it returns from a synchronisation point invocation. This offers the significant advantage of reducing the number of invocations required to control the stream; if such control were not possible, then the application would be forced to make a separate call to alter the behaviour of the stream.

5.2.3.6 Real-Time Synchronisation

IMAC does not stipulate how real-time synchronisation is to be provided, because the general solution of providing a global system clock is complex, hard to implement, does not scale and in some cases may not even be required. IMAC does *not* preclude the use of a global clock; it allows for the use of any mechanism which can provide the required degree of synchronisation.

The implementation detail of crucial importance in deciding how to provide real-time synchronisation is the ratio of the delay introduced by communication to that of the acceptable synchronisation skew.

If communication delays are very much lower than the allowed skew, then no other form of synchronisation is required since communication will appear instantaneous with respect to these other synchronisation requirements. Such an implementation could start two streams at the same time simply by issuing the relevant operations directly - they would appear to start instantaneously because the communication delays are negligible. The fact that communication delays are decreasing as technology progresses, whilst many user related synchronisation requirements remain static means that this approach becomes increasingly valid.

As the communication delay reaches, or exceeds, the allowed skew, then the need for explicit synchronisation increases. In such cases a global clock, such as that provided by the Network Time Protocol [**Mills89**] or Tempo [**Gusella83**] may be the best solution.

As discussed in section 4.2.9, real-time synchronisation ensures that multiple operations occur at the same time and relates LSF's from one isochronous stream to LSF's from another; the following

sections outline a number of possible schemes for implementing such synchronisation and show that IMAC can be used to implement any of them.

Pre-Scheduling

Given the use of a global clock it is possible to ensure that operations occur at the same time, regardless of communication delays, via the use of pre-scheduling whereby operations are explicitly scheduled for some future time.

The instructions to start a voice and video stream at the same time, would take the form of "start stream at time x", where x is set far enough into the future to avoid any delays involved in issuing these instructions.

Such operations can be easily expressed as part of the operation component of an IMAC device interface, (see section 5.2.4).

Prepare and Act Operations

Another approach is to split operations into *prepare* and *act* sub-operations. The *prepare* operation instructs the device to pre-allocate all of the necessary resources and prepare for the subsequent execution of the operation in question. When the *act* operation is issued its startup time will be significantly reduced through the use of pre-allocated resources, thus reducing the delay between the issue of the operation and it having some effect. Both the MIT Audio Server (section 3.6.2.1) and VOX (section 3.9) use this approach.

Again such operations may be easily expressed for IMAC devices.

Global Clock Timestamps

A global clock can be used by plugs to timestamp all LSF's generated and by sockets to timestamp the receipt of each LSF; in this way, it is possible to relate LSF's from one isochronous stream to LSF's from another simply by comparing timestamps.

Such timestamps may be passed as arguments to stream synchronisation point operations.

Local Clock Timestamps

If no global clock is available then it is possible to use a combination of local timestamps, coupled with a knowledge of the skew between the local clocks to provide real-time synchronisation. Each plug and socket must provide an operation for determining its local time, thus enabling the caller of this operation to determine the skew between its clock and that of the callee. Plugs timestamp outgoing LSF's and sockets timestamp incoming LSF's with their local clock. The controlling application must then add the known clock skew between each stream combination to the local timestamps generated by each stream to compute a temporary approximation to a global clock which can then be used to relate the LSF's from multiple streams.

Although this scheme is inelegant compared to that offered by the use of a global clock, it may provide an adequate degree of real-time synchronisation without the need to implement a global clock. Again the required timestamps are passed as arguments to the stream synchronisation point operations.

Real-Time Synchronisation Summary

The decision to avoid stipulating a global clock was largely motivated by the principle of choice; that is, delaying policy decisions to the latest possible stage. The provision of this choice is made possible by the flexibility offered by the use of structured streams, and in particular by the fact that stream synchronisation points are represented as operations which may take an arbitrary number of arguments of arbitrary types.

5.2.3.7 Stream Summary

IMAC streams represent the unidirectional, typed, communication of a single information medium. They are structured into a sequence of so-called *Logical Synchronisation Frames*, each of which is composed from a number of lower-level *Physical Synchronisation Frames*.

Event and stream synchronisation points are communicated to any interested party using operation invocation, thus synchronising the execution of the stream and the other party.

IMAC does not stipulate how real-time synchronisation is to be provided, but provides sufficient flexibility to support a range of real-time synchronisation schemes. Chapter 6 includes a series of examples illustrating the use of IMAC streams.

5.2.4 Devices

IMAC devices are an extension of ANSA interfaces to provide a particular style of service suited to the requirements of multimedia applications. Devices consist of an operation and stream component. The stream component contains specifications for any number of stream types and stream ends; the operation component contains operations for controlling the behaviour of the device and is directly equivalent to an ANSA interface.

- a stream type consists of:
  - a textual name denoting the stream type.
  - a list of all of the interfaces containing event synchronisation point operations supported by this stream.
  - a list of all of the interfaces containing stream synchronisation point operations supported by this stream.
  - a QoS specification.
- a stream end consists of:
  - a textual name denoting the stream plug or socket.
  - the name of the stream type which this stream end sources or sinks.
  - an indication of whether it is a plug or a socket.
- an example stream type could be:

  `VideoStream : STREAM = { EventIf } { StreamIf } QOS "X 128 Y 128"`

  a stream type called `VideoStream`, supporting single event and stream synchronisation interfaces, `EventIf` and `StreamIf`, respectively and offering a QoS of 128x128 picture resolution.

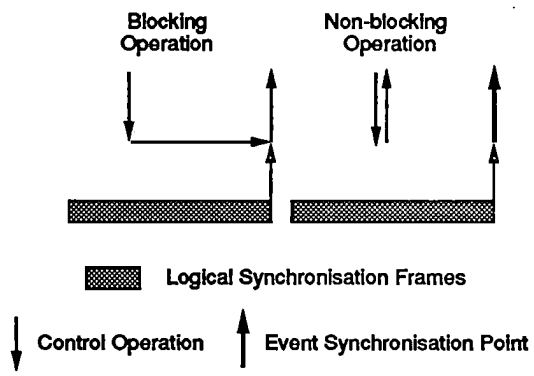

Figure 5.5: Blocking and Non-blocking Control Operations

- example stream ends:

  `VideoPlug : PLUG VideoStream`
  a plug, called `VideoPlug`, for the stream type `VideoStream`.

  `VideoSocket : SOCKET VideoStream`
  a socket, called `VideoSocket`, for the stream type `VideoStream`.

Operations in a device interface which in some way affect one, or more, streams in that interface are defined to take effect at the end of the current LSF for each stream affected. As illustrated in figure 5.5, the completion of such an operation may be indicated in one of two ways:

1. the operation is blocked until completion.
2. the operation returns immediately and an explicit event synchronisation point is defined for when the action instigated by the returned operation has taken effect.

The stream synchronisation point operations define the LSF's that the stream may support. Note that no attempt is made by IMAC to associate any behavioural semantics with the stream synchronisation point definitions; that is, there are no automated checks to determine what a particular LSF means and to ensure that the underlying implementation adheres to this meaning.

The QoS specifications take the same form as that defined in section 5.2.7 for operations. It is used to specify the range of QoS options which the stream may support, and from which a client may choose the most appropriate one for its needs. Options will typically include choosing which LSF to use from the set of possible LSF's, or specifying a particular data rate, encoding, or compression algorithm. As a simple example, an audio stream which can support either A-law or Mu-law encodings could allow the client to choose the desired encoding via QoS selection.

Devices are intended to model the underlying multimedia hardware. The decision to support multiple streams within a single device is based on the observation that multimedia hardware will often support multiple streams. The alternative of only allowing a single stream is restrictive, and in any case is a subset of supporting multiple streams. An important implementation guideline, which is illustrated by the example in section 6.4, is that devices which support multiple streams must be implemented in such a way as to allow the use of individual streams within that device in isolation of the other streams. Clearly, this is only possible if the underlying hardware allows the use of single streams in isolation of the others. By so doing, the scope for re-using such complex devices in previously unforeseen ways, and therefore system flexibility, is greatly increased.

5.2.5 Device Types

All ANSA interfaces are typed and a conformance relationship is defined to allow one interface to be substituted for another, functionally equivalent, interface. Conformance can be applied to independently developed interfaces and does not rely on any form of inheritance mechanism. Conformance is a relationship between interfaces, as opposed to a relationship between implementations as expressed by inheritance as used in Smalltalk or C++. The conformance relationship used in ANSA is defined in section A.10.1, and is extended by IMAC to include streams. For a fuller description of conformance and its implications see the ANSA Computational Model [**ANSA90a**].

A device type is specified by a (stream component type, interface type) pair. A *stream component type* is a set of *stream end signatures*, and a *stream end signature* consists of:

- a stream end name.
- a stream type name.
- a stream direction: a plug or a socket.

A device X conforms to a device Y if:

1. the stream component type of X conforms to the stream component type of Y.
2. the interface type of X conforms to the interface type of Y.

The rules for interface type conformance are given in section A.10.1.

A stream component type X conforms to a stream component type Y if:

3. for every signature in Y there is a signature in X which defines a stream end of the same name *and* stream type.
4. for every signature in Y which defines a plug the signature in X with the same stream name defines a plug.
5. for every signature in Y which defines a socket the signature in X with the same stream name defines a socket.

For instance, an interface containing stream ends `VideoSource` and `AudioSource`, of stream types `VideoStream` and `AudioStream` respectively would conform to an interface containing only the `VideoSource` stream end. Stream types represent information media and stream ends must be of the *same* stream type, that is, the same medium, if they are to be used in place of one another.

The trading service for an IMAC system must be extended to use device conformance for matching device requests to available devices.

5.2.6 ANSA Quality of Service

ANSA already provides two mechanisms which can be used to provide limited QoS support:

- trader properties and constraint expressions, (section A.3).

- interface and operation attributes, (sections A.1.1.1 and A.5).

Trader properties and constraints may be used to provide device wide QoS; that is, to specify a QoS which applies to a device as a whole. For example, a camera device could export three instances of the same interface to the trader, each with a different set of properties as follows:

Export 1: XResolution 128 YResolution 128

Export 2: XResolution 512 YResolution 512

Export 3: XResolution 1024 YResolution 1024

An import with a constraint of "XResolution > 512 and YResolution > 512" would obtain the third export. The client would then invoke operations on this interface to control a 1024x1024 video image. This is a very restrictive form of QoS since it applies to an entire interface and does not provide a mechanism for the service provider to determine the QoS actually requested. In this example the service provider is free to provide any resolution greater than 512x512. The fact that it does not know the QoS actually requested constrains it to provide the maximum 1024x1024 resolution.

In addition the current implementation of the trader does not allow the importer to determine the properties of the offer it has imported. The import constraint used above could be satisfied by either a 512x512 or 1024x1024 resolution image, but the client has no way of determining which one has been selected.

Attributes may apply to an entire interface, or to a particular operation within an interface, and provide a declarative means for controlling the various distribution transparencies provided by ANSA. The intended use for attributes is to specify the provision of transparencies such as atomicity, replication and concurrency control. Attributes will be implemented as transformations applied to the language extensions provided by ANSA for managing distribution.

Neither of these mechanisms can be used to directly control the underlying communication system, and in general cannot influence dynamic system behaviour.

5.2.7 IMAC Quality of Service

IMAC provides a new mechanism for the specification of communication oriented QoS on a per-operation basis, with the added stipulation that applications be able to determine the QoS that is being used for a given operation.

Interface operations may specify a set of separate QoS options with which they are prepared to be invoked. Operation invocations may then specify a QoS request which must be supported by this set. Although QoS may appear in the specification of an interface, it does *not* contribute to the type of the interface and consequently plays no part in conformance.

QoS options are expressed as constraints on the underlying communication system. The system is viewed as offering a set of communication properties which can be used in a variety of ways by the application. The communications system is viewed as the QoS provider or server, and the application components using communication resources as clients. This model follows naturally from the client/server paradigm in which the provider of a service or resource is represented as a server, and the user of that service or resource as a client. The novel aspect is that clients specify the services and resources they require as constraints on the set of available services and resources. In this way, clients express their requirements in a declarative, as opposed to procedural, manner.

The process of matching a particular QoS constraint to the available set of QoS offers is called *QoS negotiation* and involves the following operands:

QoS offers: the underlying system makes a series of QoS offers representing the QoS it can support.

QoS specification: each operation in an interface may specify a set of constraints which are used to select the set of QoS offers which can be used for that given operation.

QoS request: the invoker of an operation is able to specify a single QoS constraint for use with that operation. This constraint must be satisfied by the offers selected by the QoS specification.

Given that IMAC is intended for use in an open distributed environment, the QoS mechanism must be able to manage a variety of communication architectures and protocol suites. As stated in section 4.2.7.1 the QoS specification available at the application level must be declarative and expressed in application level terms and concepts, therefore some means of mapping from application level QoS to communications level QoS is required. These requirements are met by the following constructs:

QoS Domains: used to delimit the scope of QoS offers, specifications and requests. A domain represents a particular implementation and configuration of a particular protocol architecture.

QoS Layers: each QoS domain is *layered*, with each layer representing a point at which QoS is provided.

QoS Mapping: some form of mapping function is required to map QoS constraints at layer $n+1$ to constraints suitable for layer n.

Each domain may have an arbitrary number of layers depending on the nature of the communication architectures and protocols it represents. In this way end-to-end QoS may be provided by representing user interface, application and other system components, in addition to the communications system, which provide some form of QoS, as QoS layers.

An important point is that there need not be a one-to-one correspondence between QoS and protocol layers, and that other system components may be represented as QoS layers.

The term QoS negotiation is extended to include negotiation at *all* layers, and the term per-layer QoS negotiation is introduced to refer to negotiation at a single layer. Per-layer QoS negotiation takes the form a single QoS constraint which is presented as a request to the layer in question. This request, if successful, has two results:

1. a list of QoS offers made at this layer, any of which can be used to provide the QoS specified by the constraint.
2. a list of QoS constraints, one for each offer, to be passed on to the next layer. These constraints are produced by the application of a QoS mapping function. A mapped constraint may be null, in which case it will be matched by *all* QoS offers.

The terms *static* and *dynamic* QoS are used to refer to the two results of per layer QoS negotiation. These names are chosen to reflect the use to which these results will be put. The first result is the *dynamic* QoS since it will be subsequently presented to the underlying system as a request for resources, the results of which will vary depending on the current resource utilisation of the

system. The process of selecting a single offer, from a set of candidate offers, is called dynamic QoS negotiation. The second result is used in subsequent QoS negotiation whose results should only change when the system configuration is changed, that is, when new QoS offers are added or old ones removed.

In order to ensure the consistent use of QoS domains and constraints between interface providers and users some form of system wide database is required for storing QoS information. This service, called the QoS Manager, is described in section 5.3.1.

5.2.7.1 QoS Algorithms

Given that the underlying system must make a series of QoS offers for each layer in each domain it supports, it is possible to outline the algorithms used for negotiating QoS across all layers.

The algorithm for QoS specifications is outlined in figure 5.6[5]. This algorithm is recursive and identifies all possible combinations of QoS offers which can be used to provide the stated QoS specifications. For simplicity, the resulting offers are concatenated into a single string. However as shown in section 5.2.7.2, the offers resulting from QoS negotiation at each layer can be combined to form a set of protocol stacks, any of which can potentially provide the specified QoS. In this way, QoS negotiation is carried out at each layer, and the QoS constraints presented to each layer are mapped from one layer to the next by the per-layer QoS negotiation operation.

The algorithm for QoS requests is similar to that used for specifications, except that it includes an additional check to ensure that only QoS offers which are supported by *both* client and server are used. This assumes that the set of offers produced by the QoS specification algorithm are available to the client. The algorithm for QoS requests is outlined in figure 5.7[5].

IMAC does not specify the representation to be used for QoS offers, specifications or requests, neither does it specify the per-layer QoS negotiation algorithm. This is another application of the principle of choice, and allows the implementation to choose the most appropriate representation and algorithm for its requirements.

5.2.7.2 QoS Protocol Stacks

The QoS algorithms can be viewed as either producing, or operating on, a set of protocol stacks, any of which may be used to provide the requested QoS. The QoS specification algorithm produces such a set, and the QoS request algorithm subsequently selects the most appropriate member of this set.

Such protocol stacks consist of a sequence of QoS offers, where each offer is taken from the results of per-layer QoS negotiation at each layer. The set of protocol stacks is obtained by combining each offer at layer $n+1$ with each offer at layer n. This is illustrated in figure 5.8 where each possible path from the top layer to the bottom layer represents a protocol stack. The total number of protocol stacks is given by the number of possible paths from the root of the tree to its leaves.

This view highlights the utility of QoS for specifying a multiplicity of protocol stacks and for selecting a single stack which best matches the applications communications requirements. As discussed in section 4.2.4.5 this is required if full advantage is to be taken of minimal multiplexing.

[5]The domain is specified for the entire interface.

```
PROCEDURE negotiate_constraint( layer, constraint )
  RETURNS ( result, offers )
BEGIN

  new_offers := negotiate QoS at the layer for constraint

  IF negotiation unsuccessful THEN
    result := FALSE
    offers := NULL
    RETURN
  END

  IF layer == last layer THEN
    result := TRUE
    offers := new_offers
    RETURN
  END

  next_layer = next layer in the current domain

  offers := NULL
  sucess := FALSE
  FOR offer IN new_offers DO
    next_constraint := mapped constraint for offer

    ( r_result, r_offers ) :=
      negotiate_constraint( next_layer, next_constraint )

    IF r_result == TRUE THEN
      offers := offers + r_offers -- string concatenation
      success := TRUE
    END

  DONE

  result := success
  RETURN

END

FOR constraint IN each constraint in the QoS specification DO

  layer := first layer in the current domain

  ( success, offers ) := negotiate_constraint( layer, constraint )

DONE
```

Figure 5.6: Algorithm for Negotiating QoS Specifications

```
current_constraint := QoS request, success := TRUE

FOR layer IN each layer in the current domain DO

  negotiate QoS at layer for current_constraint

  IF negotiation unsuccessful THEN
    success := FALSE
    BREAK -- leave this loop
  END

  FOR offer IN offers resulting from negotiation DO
    IF offer cannot be used to satisfy QoS specification THEN
      discard offer
    END
  DONE

  next_offer := result of dynamic QoS negotiation on remaining offers
  current_constraint := mapped constraint for next_offer

DONE
```

Figure 5.7: Algorithm for Negotiating QoS Requests

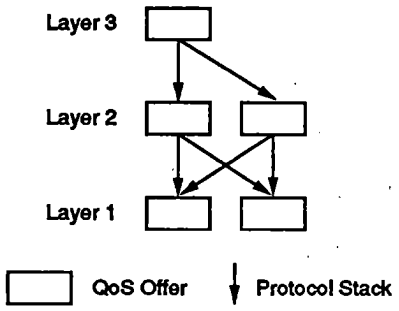

Figure 5.8: QoS Protocol Stacks

5.2.7.3 End-to-End QoS

Dynamic QoS requests can be made at any time preceding the invocation. If QoS is to be provided on an end-to-end basis then sufficient resources must be available at *both* the client and server ends of the invocation. In other words the dynamic QoS requests must be satisfied by both client and server.

There is a tradeoff to be made between when and for how long resources are allocated and the degree of certainty that sufficient resources will be available when they are required. At one extreme it is possible to pre-allocate the maximum resources ever likely to be required; this guarantees that the QoS required will be available, but may be prohibitively expensive to implement. On the other hand it is possible to only allocate resources when they are required; although this makes efficient use of resources it makes providing QoS guarantees considerably more difficult. Providing guarantees does not necessarily require the pre-allocation of resources. For instance, some form of priority based scheme may be used to ensure that guaranteed resources can be made available

when they are required, but may be available for wider use at other times. In such a scheme only the available priorities are pre-allocated and not the resources themselves. IMAC attempts to offer a compromise between these two extremes as discussed below.

In order to make efficient use of the available resources it is necessary to defer making dynamic QoS requests until the latest possible time. In particular, if the server is to efficiently allocate resources it must know the QoS actually requested by the client. This implies that some communication takes place between the client and server *prior* to the invocation of the QoS controlled operation. IMAC does *not*:

- stipulate precisely when or what form the communication of QoS results from the client to the server should take place - just that it should occur before the operation is invoked.
- stipulate that QoS be re-negotiated on a per operation invocation basis - it is left to the implementation to decide when re-negotiation takes place.

This is an application of the principle of choice and allows the system implementor to choose the optimal combination of QoS re-negotiation and client/server communication of QoS for the implementation environment in question. In this way, it is possible to make an implementation specific trade-off between efficient resource utilisation and QoS guarantees which is likely to be more efficient than one made at an earlier time. The exact trade-offs made should be invisible to the application writer; that is, application code written in an environment where one trade-off has been made should also work in a different environment where a different trade-off has been made. Therefore, the only observable difference should be one of performance.

5.2.7.4 Availability of QoS Results

Both the client and server applications must have access to the results of the preceding QoS negotiation since the QoS actually provided is likely to affect the subsequent behaviour of said applications. The QoS information available falls into the following categories:

1. server operations being informed of the QoS request specified by the client.
2. client and server being informed of the QoS offers used, at the *uppermost*, QoS layer to satisfy the QoS request.
3. client and server being informed of the QoS offers used at *all* layers.

IMAC stipulates that 1 and 2 must be provided whilst 3 is optional. This allows applications to vary their behaviour and in particular the degree of interactivity they provide in response to the QoS currently in use; see section 4.2.5.

Allowing access to the lower layers of QoS negotiation may lead to non-portable applications and is therefore discouraged. However, there may be applications which have an absolute requirement for such information.

5.2.7.5 Multi-Channel Synchronisation

An inevitable consequence of providing per-operation QoS is that invocations using different QoS and hence communication channels, will no longer be synchronised with respect to each other. The

term *multi-channel synchronisation* is used to refer to the synchronisation of operation invocations issued over separate communication channels.

In many cases, such operations are inherently independent of one another and hence have no need for multi-channel synchronisation. However, in other cases such synchronisation will be required.

Any implementation of multi-channel synchronisation is likely to rely on the insertion of sequence numbers, at the client, which can be used to re-order invocations at the server. Such sequence numbers may be generated, and invocations ordered, at any of the following levels:

- within the remote invocation protocol, with appropriate sequence numbers being inserted in the protocol header and re-ordering implemented by the destination protocol implementation.
- as part of the stub code generated for marshalling and unmarshalling invocation arguments and for invocation dispatching at the server.
- as part of a more general mechanism such as a group execution protocol or atomic transaction manager.
- left to the application to include explicit sequence numbers and implement re-ordering itself.

Each of these possible solutions has its own advantages and disadvantages, and each is best suited to a different implementation environment. The principle of choice requires that IMAC does not make the choice between these options, and it is consequently left to the implementation to choose the most appropriate for its needs.

5.2.7.6 QoS Summary

IMAC provides comprehensive support for QoS on an end-to-end, per-operation basis. QoS is specified at the application level as a constraint on the services provided by the underlying system. QoS constraints may be mapped from one layer to the next, thus shielding higher levels from lower-level QoS details.

The algorithms for QoS negotiation were outlined and the results of these algorithms are subsequently made available to the client and server components of the operation in question.

Although IMAC QoS is primarily targeted at the communication system, its structure is sufficiently general to allow other system components to be represented as a QoS layer and thus participate in QoS negotiation.

5.3 IMAC Services

This section defines the functions of the various IMAC services mentioned in section 5.2.2 in greater detail.

5.3.1 QoS Manager

The QoS manager provides a system wide database for storing QoS data and may also be used to provide compile-time checking of QoS specifications and requests. Its functions can be split into two broad categories:

Database Management: creation of QoS domains, layers within domains and naming of domains and layers. Domains and layers are named using textual strings.

Domain Operations: per domain operations including:

- registering QoS offers at a given layer.
- specifying QoS mappings at a given layer.
- per-layer QoS negotiation.

The exact format for QoS offers, specifications, requests and mappings is *not* defined by IMAC, it is left to the implementation to choose the most appropriate representations for its needs. Similarly the exact nature of per-layer QoS negotiation will depend on the representations chosen and is also implementation dependent.

The QoS negotiation algorithms outlined in section 5.2.7.1 may be implemented within the QoS manager or in some other system component which uses the QoS manager purely as a database. The provision of per-layer QoS negotiation allows for compile-time checks to be implemented by programme development tools which access the QoS manager database and QoS negotiation facilities.

5.3.2 User Locator

The User Locator is responsible for identifying the current physical location of a given user, and given a user name it will determine and return an address for that user which can be used for subsequent communication. The request takes the form of "find an instance of the specified service at the user's current location". The address returned takes the form of an interface reference. In order to associate a location with a service it is necessary to impose the convention that all services and IMAC devices which can be accessed by the user locator, include a property in their export to the trader which specifies their location. For instance a property of the form "Location Basement" could be used. The algorithm implemented by the User Locator is as follows:

1. determine the physical location of the user.
2. import the specified service with the constraint that the location property is the user's current location.

IMAC does not specify how a user should be located, just that the User Locator is informed of the user's physical location whenever it changes. This allows for a variety of location schemes to be implemented.

5.3.3 Desktop Manager

The Desktop Manager provides a management interface for controlling the various devices and services provided by a single Multimedia Desktop, (see section 2.5). It extends the ANSA Node Manager (section A.12) to support the creation and destruction of devices in multiple run-time environments, and to provide management of shared resources.

The first extension can be implemented by extending the Node Manager's service description to include an optional specification of the service to use for creating and destroying the service in question. This allows for each service to specify the means used to create and destroy it.

The Node Manager already allows the number of activations for a given service to be bounded and thus provide for simple resource management. The Desktop Manager extends this scheme to allow multiple services to be collected into an *activation group* and to limit the number of activations allowed for the entire group. Shared resources can then be represented by a group and all users of that resource included in the group, thus allowing access to the resource to be conveniently controlled.

As a simple example consider a workstation equipped with a single speaker and which supports `AudioWindow` and `Telephone` IMAC devices which use this speaker. By creating a new activation group, which allows at most one activation, and which contains the `AudioWindow` and `Telephone` devices, it is possible to ensure that only one device is active, and hence using the speaker, at any given time.

In order to allow for more sophisticated resource management policies to be built on top of this mechanism the Desktop Manager must allow for the external control of the number of activations, and for access to the current status of the activation group. In this way, a device may temporarily relinquish control of a shared resource by explicitly decrementing the activation count for the group in question, without terminating its execution.

5.3.4 Translation Manager

The Translation Manager provides a database for translator devices which can be searched for devices with a plug and socket of specified types. All devices capable of translating from one stream type to another must be registered with the Translation Manager in addition to any other databases such as the Trader or Desktop Manager. The Translation Manager is consulted whenever an attempt to connect a plug to a socket fails due to a stream type mismatch.

5.4 Orchestration

Orchestration is the name given to the management functions required to coordinate IMAC streams, devices, QoS and services. Orchestration provides a uniform interface to the other components of IMAC and provides commonly used functions as part of this interface.

There is no single architectural component or service charged with implementing orchestration. Orchestration functions will typically be provided by a combination of language extensions, application libraries and services such as the translation and QoS managers.

In order to implement orchestration functions it is necessary for *all* device interfaces to provide a common set of management operations. In other words, all device interfaces conform to a common interface, containing the required management operations. The operations provided, their arguments, results and terminations are not defined by IMAC - it is left to the implementation to specify this management interface.

However, IMAC does provide some general requirements for the system implementor, and in particular the following functions must somehow be incorporated into the orchestration interface.

- type checked connection management of stream plugs and sockets.
- explicit identification of translator devices, and subsequent interaction with the translation manager to register their existence.

- trapping of stream connection failures due to type mismatch and subsequent interrogation of the translation manager to find a suitable translation device.
- specification of client QoS requests and server QoS specifications.
- specification of QoS offers provided by particular client and server implementations, that is, providing a means for stating the current communication system configuration of client and servers.
- creation and passing of interface references for event and stream synchronisation operations.
- provision of an interface to the user locator.

Providing orchestration functions as language extensions, rather than as libraries, offers a number of important advantages:

- provides a simple, coherent, programming model.
- allows for compile-time checking.
- decouples the application from lower level implementation details and interfaces.

The first two advantages simplify application development. The third allows for the independent evolution of the application and the underlying system, for instance, it is possible to change lower level system interfaces without disturbing the application.

The functions provided by orchestration, and the way they are provided, are intended to reduce the complexity and thus ease the programming burden of coordinating the activity of multiple system components to provide a coherent view of the underlying system components.

5.5 Summary

The IMAC architecture and the principles which influenced its design have been presented in detail. In addition, the motivation and justification for its design have been presented, as have the features deliberately omitted from IMAC. IMAC is an extension of the ANSA architecture and its relationship to, and features inherited from ANSA have also been discussed.

IMAC streams provide the basis for the synchronisation of multiple, related, information media, whilst IMAC devices are instrumental in managing heterogeneity. Comprehensive support is provided for end-to-end, per-operation QoS. A suite of services are defined for locating users, managing devices, QoS and stream translation.

All IMAC components are coordinated by the so-called orchestration functions which provide a uniform interface to all of the other IMAC functions and components.

6

IMAC Examples

This chapter presents some examples illustrating how IMAC may be used to implement a variety of synchronisation schemes and to manage heterogeneity. A final example contains some sample orchestration functions and demonstrates how such functions simplify interacting with multiple IMAC services.

The pseudocode used in these examples is not part of IMAC; it is used purely to illustrate and simplify the explanation of the examples presented. The only things of note are the **INVOKE**, **WAIT_FOR** and **RETURN INVOCATION** statements. **INVOKE** issues a remote invocation, **WAIT_FOR** suspends the calling thread until one, or more, specified invocations are received, and **RETURN** allows a **WAIT_FOR** invocation to return. The IMAC language is ANSA PREPC with the extensions defined in chapter 7.

6.1 Event Synchronisation Example

This example provides a solution to the event synchronisation problem outlined in section 4.2.8.1.

A video stream of type **VideoStream** is implemented by a **Camera** device containing a plug, and a **VideoWindow** device containing a socket. The interfaces for these devices are outlined in figure 6.1.

The **VideoWindow Create** operation will not return until a video window is actually displayed on the screen, this device also invokes the following event synchronisation operations:

VideoStarted: an interrogation invoked when the video stream is first transmitted or received.

VideoTerminated: an interrogation invoked when the last sample is transmitted or received.

The controlling application creates a video stream, and allows it to run until the user requests its termination; this application is presented in pseudocode form in figure 6.1, and diagrammatically

```
Camera: DEVICE =
BEGIN

  videosource: PLUG VideoStream;

  OPERATION Play; -- start video output
  OPERATION Stop; -- stop video output

END

VideoWindow: DEVICE =
BEGIN

  videosink: SOCKET VideoStream;

  OPERATION Create; -- create a video window
  OPERATION Destroy; -- destroy video window

END

Application: PROGRAM =
BEGIN

  INVOKE VideoWindow.Create
  INVOKE Camera.Play

  WAIT_FOR VideoStarted INVOCATION
  -- issue prompt for user input

  RETURN VideoStarted INVOCATION

  WAIT_FOR user input
  -- user input received

  INVOKE Camera.Stop

  WAIT_FOR VideoTerminated INVOCATION
  RETURN VideoTerminated INVOCATION

  INVOKE VideoWindow.Destroy

END
```

Figure 6.1: Event Synchronisation Example Pseudocode

in figure 6.2. The `VideoStarted` event synchronisation point allows the application to perform any initialisation which is required once the video stream is active but before it is displayed; in this example the application issues a prompt for the subsequent termination of the video stream.

6.2 Monitoring Synchronisation

The previous example can be easily extended to implement simple monitoring of the video stream's synchronisation and also to handle communication errors gracefully.

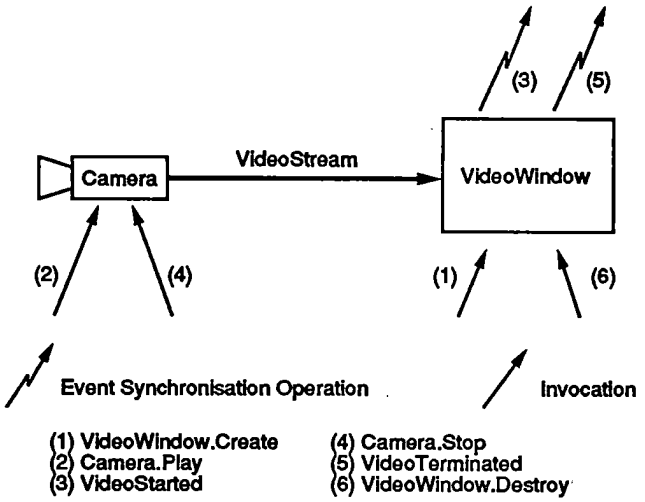

Figure 6.2: Event Synchronisation Example Diagram

The video stream is extended to support LSF's representing a single video frame, the end of each LSF represents a stream synchronisation point which results in the invocation of a `FrameReceived` interrogation. `FrameReceived` takes two arguments:

`st_LSF_seq`: the sequence number of the stream's current LSF.

`st_LSF_time`: the timestamp of the stream's current LSF.

`FrameReceived` has three results:

`sync_error`: a boolean, which if true, indicates that a synchronisation error has been detected.

`app_LSF_seq` the sequence number of the application's current LSF.

`app_LSF_time` the timestamp of the application's current LSF.

The stream implementation examines these results and if a synchronisation error has been detected it must take some action to restore it. By examining the `app_LSF_seq` and `app_LSF_time` results it can determine if it is running ahead or lagging behind the application. If behind, it can attempt to catch up by dropping any buffered data and moving to the most recently received data. If ahead, it may slow itself down by ignoring recently received data. In both cases it may be possible to use flow control, or even to vary the QoS being used, in order to vary rate at which LSF's are transmitted and received.

The issue of flow control is complicated by the potential use of multicast. For instance, if a stream sink finds itself running ahead of its controlling application, then it may ask the source to slow down its rate of transmission. However, if the source is multicasting to multiple sinks then the source must decide whether to slow its transmission to this single sink, or to *all* sinks. This problem is compounded if some of the sinks are themselves multicasting. The obvious solution of applying flow control on a per-sink basis, that is, transmitting at different rates to each multicast sink, may be very expensive to implement if it precludes the use of hardware multicasting provided by the underlying network.

```
-- create video window and start video stream
LOOP
  WAIT_FOR FrameReceived OR CommunicationError INVOCATION

  IF CommunicationError THEN
    BREAK -- leave loop
  END

  -- FrameReceived invocation.
  sync_error := FALSE

  IF first invocation THEN
    prev_st_LSF_seq := app_LSF_seq := st_LSF_seq
    prev_st_LSF_time := app_LSF_time := st_LSF_time
    prev_local_time := current_time()
    RETURN FrameReceived INVOCATION
    CONTINUE -- iterate
  END

  n_frames := prev_st_LSF_seq - st_LSF_seq
  expected_frames := (current_time() - prev_local_time) / frame_time

  IF n_frames > (expected_frames+2) OR
     n_frames < (expected_frames-2) THEN
    -- Synchronisation has been lost
    sync_error := TRUE
    app_LSF_seq_no = prev_st_LSF_no + expected_frames
    app_LSF_time = prev_st_LSF_time + (expected_frames * frame_time)
    RETURN FrameReceived INVOCATION
  ELSE
    -- Synchronisation is maintained
    RETURN FrameReceived INVOCATION
  END

  prev_st_LSF_seq := st_LSF_seq
  prev_st_LSF_time := st_LSF_time
  prev_local_time := current_time()

END
-- Terminate the video stream
```

Figure 6.3: Monitoring Stream Synchronisation Example

The event synchronisation operation, `CommunicationError`, is invoked by either the plug or socket device if a communication error is detected. A stream synchronisation skew of two video frames is allowed. Figure 6.3 extends figure 6.1, to monitor the streams synchronisations and to handle communication errors.

This example assumes that the time taken to transmit and receive the **FrameReceived** operations is significantly less than the allowed skew. It also does not monitor the communication latency between the stream plug and socket.

A frame rate of 50 frames per second requires $20ms$ per frame, thus imposing an upper bound of $40ms$ (two frames worth of skew) for the **FrameReceived** operation. Given that current RPC implementations provide circa $10ms$ performance and that announcements, although unreliable, will be even faster since they need not wait for a reply, the first assumption is valid. Section 9.1

includes more detailed RPC performance results for the ANSA Testbench and other contemporary RPC implementations.

Note, that because both announcements and interrogations may not reach the application within the tolerated communication latency as a result of communication errors, any practical implementation must impose a timeout at each point waiting for such operations. In most cases the timeout can be set to the same value as the acceptable synchronisation skew and if it expires a loss of synchronisation will almost certainly have occurred.

6.3 Mutually Synchronised Streams

This example illustrates how real-time synchronisation between two streams might be implemented. The two streams are synchronised with respect to each other, and with respect to real-time. The controlling application is only required to start and stop both streams at the same time. For simplicity, both streams are of the same `VideoStream` type used in the previous example. One of the streams could be replaced by a different stream type without altering the synchronisation algorithm used in any fundamental way. A global clock is assumed and the `Camera` operations are extended to support pre-scheduling with the addition of a "time to start" argument. The new `Camera` interface and application are outlined in figure 6.4. A time of one second is assumed to be long enough to outlive the duration of operation invocations.

Two LSF's (i.e. video frames) worth of skew is allowed between peer streams and between each stream and real-time. Synchronisation is implemented within the `VideoWindow` devices. Incoming LSF's are added to a first in, first out, (FIFO) buffer to smooth out communication jitter. Synchronisation is implemented at the point where LSF's are removed from the FIFO by only displaying LSF's which satisfy *both* of the following synchronisation conditions:

1. the LSF lies within the range defined by the allowed skew with respect to real time; that is, within ± 1 frame time.
2. the LSF lies within the range defined by the allowed skew between peer streams; that is, within ± 2 LSF's.

The real-time skew is specified as half of that allowed between peer streams to ensure that even if the peer streams are at opposite extremes of the real-time skew (i.e. one is 1 frame ahead of real-time, whilst the other is 1 frame behind) then inter-stream synchronisation is still maintained.

The first check, for real-time synchronisation, is easily implemented by comparing the timestamp of the LSF removed from the FIFO with the current value for the global clock. The check for the second condition can be implemented by comparing the sequence number of the LSF just removed from the FIFO, with that of the most recent LSF reported by the peer stream via an invocation of `FrameReceived`.

If real-time synchronisation has been lost (condition 1) then the stream must decide if it is running ahead, or behind, of real-time and take appropriate action as outlined in the previous example. If synchronisation has been lost with respect to the peer stream, the local stream must decide if it, or the peer stream, is in error. This can be determined by checking its real-time synchronisation, that is, if the local stream meets the first condition but fails the second, then the peer stream is in error. If it fails both conditions then it is in error, but cannot decide on the status of the peer stream.

The local stream can tell the peer stream that it (i.e. the peer stream) is out of synchronisation via the results of the peer stream's next `FrameReceived` invocation. Similarly the local stream must

```
Camera: DEVICE =
BEGIN

  PLUG VideoStream;

  OPERATION Play [ time ]; -- start video output
  OPERATION Stop [ time ]; -- stop video output

END

Application: PROGRAM =
BEGIN

  INVOKE VideoWindow_1.Create
  INVOKE VideoWindow_2.Create

  time_x = current_time + 1 second

  INVOKE Camera_1.Play [ time_x ]
  INVOKE Camera_2.Play [ time_x ]

  -- wait for user input, etc

  time_y = current_time + 1 second

  INVOKE Camera_1.Stop [ time_y ]
  INVOKE Camera_2.Stop [ time_y ]

  INVOKE VideoWindow_1.Destroy
  INVOKE VideoWindow_2.Destroy

END
```

Figure 6.4: Pre-Scheduled Operations

```
Thread1 : THREAD =
  LOOP
    WAIT_FOR incoming LSF OR stream termination

    IF stream termination THEN
      INVOKE StreamTerminated
      BREAK -- leave loop
    END

    IF first LSF THEN
      INVOKE StreamStarted
    END

    add LSF to FIFO buffer
END
```

Figure 6.5: Mutually Synchronised Streams: LSF Reception Thread

```
Thread2: THREAD =
  LOOP
    WAIT_FOR next frame_time

    IF FIFO buffer empty THEN
      current_LSF := prev_LSF
    ELSE
      current_LSF := LSF at head of FIFO buffer
      prev_LSF := current_LSF
    END

    IF synchronisation check 1 fails THEN
      IF ahead of real-time THEN
        apply flow control
        ITERATE -- ignore this LSF
      ELSE -- behind real-time
        flush multiple buffers from FIFO
        display most recently received LSF
        ITERATE
      END
    ELSE
      IF synchronisation check 2 fails THEN
        tell Thread 3 that a synchronisation error has been detected
      END
    END

    display current_LSF

    st_LSF_seq := current_LSF.seq_no
    st_LSF_time := current_LSF.time
    INVOKE FrameReceived

    IF sync_error THEN
      -- peer claims that we are out of synchronisation
      IF synchronisation check 2 using app_LSF_seq_no
         and app_LSF_time fails THEN
        attempt to resynchronise
      END
    END
  END
END
```

Figure 6.6: Mutually Synchronised Streams: Synchronising Thread

examine the results of any **FrameReceived** invocations it makes to determine if the peer stream has detected that it (i.e. the local stream) has lost synchronisation. The local stream must then decide what action to take based on the previous result of the real-time synchronisation check and by repeating the second check using the newly received arguments in the **FrameReceived** invocation. If, for instance, it has lost synchronisation with respect to its peer stream it may attempt to restore such synchronisation by increasing the amount of skew it experiences with respect to real-time, (so long as it remains within the allowed bounds) and thus catch up, or wait for its peer. In this way both streams monitor each other's synchronisation,[1] in addition to their own.

This synchronisation algorithm is outlined in figures 6.5, 6.6 and 6.7 and makes use of three threads

[1] Each stream acts as an application monitoring the synchronisation of the other.

```
Thread3: THREAD =
  LOOP
    WAIT_FOR FrameReceived INVOCATION

    app_LSF_seq := current_LST.seq
    app_LSF_time := current_LST.time
    IF Thread 2 has detected a synchronisation error THEN
      sync_error := TRUE
    ELSE
      sync_error := FALSE
    END
    RETURN FrameReceived INVOCATION
  END
END
```

Figure 6.7: Mutually Synchronised Streams: Invocation Reception Thread

as follows:

1. to receive LSF's from the stream plug and add them to the FIFO.
2. to remove frames from the FIFO, to perform the synchronisation checks and take any necessary action, to display frames and to issue **FrameReceived** invocations on the peer stream.
3. to wait for **FrameReceived** invocations from the peer stream.

If multiple synchronisation losses are detected in quick succession and attempts to resynchronise fail then either, or both, streams can invoke the **CommunicationError** event synchronisation operation to inform the application of an irretrievable loss of synchronisation. Similarly if **FrameReceived** invocations repeatedly fail then **CommunicationError** should again be invoked.

Implementing such mutually synchronised streams is undoubtedly complex but provides increased robustness in the face of synchronisation losses and relieves the application from the burden of monitoring synchronisation. However, this complexity is confined to the stream implementation and does not affect the application. In addition, the fact that the synchronisation interface is clearly defined in terms of stream synchronisation operations makes it possible to build a set of library streams implementing often used synchronisation algorithms. A particularly useful example would be the provision of a real-time clock stream which would periodically issue and accept a variety of synchronisation point operations and be used by streams requiring real-time synchronisation but which do not have access to a real-time clock. That is, real-time synchronisation could be achieved by synchronising streams to another, clock, stream which is known to be synchronised with respect to real-time.

Note that this example is only intended to give a general view of how LSF's can be used to implement synchronisation, and is not in any way intended to be definitive.

6.4 Managing Heterogeneity

The previous examples have concentrated on the control and synchronisation of IMAC streams, this example illustrates how IMAC devices can be used to manage heterogeneity.

```
Microphone: DEVICE =
BEGIN

  audio: PLUG AudioStream;

  OPERATION Start; -- start voice input
  OPERATION Stop;  -- stop voice input

END

CompactDiscPlayer: DEVICE =
BEGIN

  audio: SOCKET AudioStream;

  OPERATION Start; -- start playing the current track
  OPERATION Stop; -- stop playing the current track
  OPERATION Next; -- skip to next track
  OPERATION Prev; -- skip to previous track
  OPERATION Program [ tracks ]; -- program a sequence of tracks

END

VideoPhone: DEVICE =
BEGIN

  audio: PLUG AudioStream;
  video: PLUG VideoStream;

  speaker: SOCKET AudioStream;
  screen: SOCKET VideoStream;

  OPERATION Start; -- start communication
  OPERATION Stop; -- stop communication
  OPERATION Suspend; -- suspend communication
  OPERATION Resume; -- resume communication

END
```

Figure 6.8: Real-Time Voice Devices

Figure 6.8 contains definitions for a variety of devices capable of sourcing a real-time voice stream. Each of these interfaces is likely to be supported by a different variety of workstation or network server consisting of different multimedia hardware. An application charged with obtaining voice input from *any* of these would, at first sight, be required to manage *all* of the different interfaces specified. However, the use of device conformance allows the application to deal only in terms of the simplest device which provides the functionality it requires, in this case the **Microphone** device. The **CompactDiscPlayer** and **VideoPhone** devices both conform to the **Microphone** device and may therefore be used in its place. Although conformance allows such functionally equivalent devices to be substituted for each other it does not guarantee that the underlying implementation may actually be used in the manner expected by the simpler device. For instance, the **VideoPhone** implementation may not be capable of supporting voice only communication. This leads to the simple implementation (see section 5.2.4) guideline that devices supporting multiple streams must allow the use of a subset of the streams supported and not insist that *all* the streams be used.

The fact that the same interface may be implemented across a number of different hardware

```
VideoErrors: INTERFACE =
BEGIN

  OPERATION CommunicationError

END

VideoEvents: INTERFACE =
BEGIN

  OPERATION VideoStarted
  OPERATION VideoTerminated

END.

Application: PROGRAM =

  -- user_1 and user_2 are specified on the command line
  sourceDesktop := LOCATE( user_1, DesktopManager )
  destDesktop : LOCATE( user_2, DesktopManager )

  camera := sourceDesktop.Create( Camera )
  videowindow := destDesktop.Create( VideoWindow )


  stream := CONNECT camera.videosource TO videowindow.videosink

  errors := CREATE.INTERFACE( VideoErrors )
  events := CREATE.INTERFACE( VideoEvents )

  SETEVENTS errors FOR camera
  SETEVENTS errors FOR videowindow
  SETEVENTS events FOR videowindow

  -- control the video stream as in the previous example

  DESTROY stream

  destDesktop.Destroy( videowindow )
  sourceDesktop.Destroy( camera )

END
```

Figure 6.9: Orchestration Example

platforms is completely transparent to the clients of the interfaces.

6.5 Orchestration

This section illustrates some likely orchestration functions. The event synchronisation example (see section 6.1) is revisited and the orchestration operations required to locate users, create and destroy devices, create streams and set synchronisation point operations are shown. These operations are shown in pseudocode form in figure 6.9. The application takes command line arguments identifying the users wishing to communicate. The LOCATE orchestration function interfaces to the User

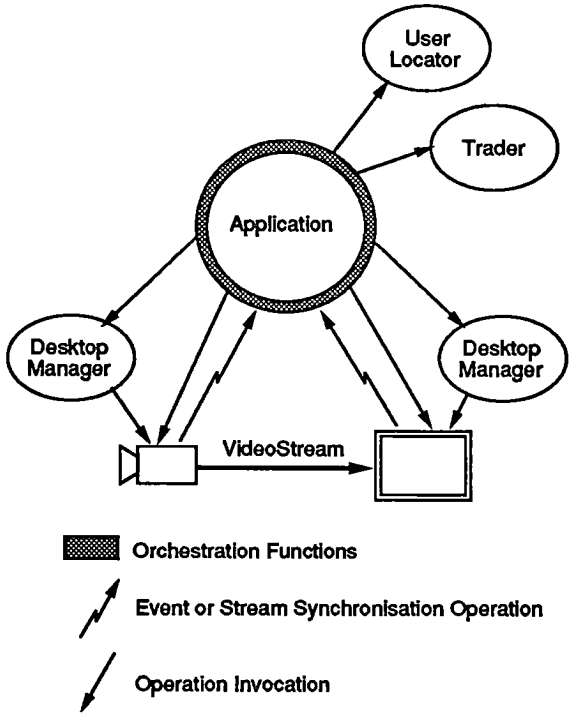

Figure 6.10: Orchestration Interaction Diagram

Locator and Trader and finds an instance of the Desktop Manager running on the workstation nearest each of the users. If the users are not found, this operation will report their absence. The Desktop Manager is then used to create an instance of the `Camera` and `VideoWindow` devices at the respective locations. These operations will fail if insufficient resources are available to create the devices. For example, if the source workstation only supports one camera and that camera is already in use then the request to create a new `Camera` device will be refused. Having created the devices their stream plug and sockets may be connected to create an end-to-end stream along which video may flow. Then the interfaces containing required event synchronisation operations (i.e. `VideoErrors` and `VideoEvents`) are created and passed to the two devices. Note, that only the `VideoWindow` is asked to invoke the `VideoStarted` and `VideoTerminated` event synchronisation operations, whilst both devices are asked to invoke `CommunicationError`. This is because this particular example only requires synchronisation with the `VideoWindow` device, other applications may require synchronisation with the `Camera` as well.

Each of the orchestration functions themselves (`LOCATE`, `SETEVENTS` etc) may be complex but this complexity is hidden from the application. In this way orchestration simplifies the process of interacting with multiple services. Figure 6.10 illustrates the interactions involved in this example.

6.6 Summary

The examples given in this chapter have outlined how IMAC can be used to control and synchronise information media streams, to manage heterogeneity and to provide orchestration.

Although IMAC does not provide any algorithms for implementing synchronisation it does provide a flexible framework within which to implement such algorithms. The provision of a well defined

interface for stream synchronisation allows a library of commonly used streams and their associated synchronisation algorithms to be compiled and made available for re-use.

7

A Prototype Implementation

This chapter describes a prototype implementation of the IMAC architecture carried out in order to establish the feasibility, and identify the strengths and weaknesses of the architecture. A complete implementation, including demonstrable multimedia applications, would require greater resources, both time and material, than those available for the purposes of this dissertation. Therefore, the prototype implementation concentrates on the original and novel aspects of IMAC at the expense of other system components which although important, have and are being researched elsewhere. The description of the prototype implementation concentrates on its design and structure as opposed to lower level implementation details. This allows the work presented to be more easily compared with other related architectures and system designs.

Just as the Testbench is an example implementation of ANSA, the prototype implementation presented here is an *example* implementation of IMAC, and is by no means definitive.

The practical work for this dissertation was initially based on version 2.5 of the Testbench. The results of this initial work, presented in section 7.2.1, proved sufficiently general to be incorporated into version 3.0 of the Testbench.[1] Version 3.0 was then used as the new basis for the prototype implementation presented here, the resulting prototype implementation is often referred to as IMAC 3.0. Some of the design and implementation of version 3.0 was the work of the author, and where such work is of direct relevance to IMAC and to this dissertation, it is described as being part of IMAC 3.0.

[1] The exact syntax and detailed changes made were agreed with Joe Sventek before being incorporated into version 3.0.

7.1 Overview

All implementation and experimental work has been carried out over UNIX, primarily because of the powerful programme development environment it provides. However, the resulting prototype implementation can be easily ported to the other platforms supported by the Testbench; these include MS-DOS, VMS and Wanda. No use has been made of UNIX specific features other than those already used by the Testbench.

The Testbench is briefly described in section A.2 and in the Testbench Implementation Manual [**ANSA90b**]. The Testbench can be viewed as having four principal components, each of which has been extended or modified to create the prototype implementation presented here. These components are:

Capsule: the run-time system for a Testbench application, including support for threads and communications. These functions are accessed via a procedural interface, called the *capsule library.* A single instance of this run-time system is called a *capsule.*

IDL: the Interface Definition Language used to define ANSA interfaces, and an associated compiler, STUBC, which generates stub code from the IDL definitions. STUBC also generates a *signature* file containing a description of the interface compiled.

PREPC: a preprocessor which scans C programs for embedded statements (referred to as PREPC statements) which augment the original program to bind to interfaces and invoke remote operations. These statements are translated into calls on either the capsule library or the stub procedures generated by STUBC.

Services: the various services required by an ANSA distributed system. In Testbench version 3.0 the following service's are provided:

- **Trader:** (section A.3), provides a directory and management facility for distributed application components.
- **Factory Service:** (section A.11), provides a means for creating and destroying new application instances.
- **Node Manager:** (section A.12), provides an interface for managing services on a single node.

The features provided by IMAC 3.0 can be grouped as follows:

ANSA Deficiencies: both the Testbench implementation and the Engineering Model[2] are found to be deficient in a number of respects when used to support IMAC and multimedia in general. Wherever possible these deficiencies have been remedied.

QoS: the implementation of IMAC QoS within the Testbench communication system, the representation of QoS offers and constraints, and QoS negotiation algorithms. The extension of IDL and PREPC to provide a programming interface for QoS.

IDL Streams and Devices: extensions to IDL to support application programming of IMAC streams and devices.

PREPC Streams and Devices: extensions to PREPC to support application programming of stream, device and connection management.

[2]The abstract design for the Testbench.

IMAC Services: the design and implementation of the various services required by IMAC.

Orchestration Functions: functions provided for co-ordinating IMAC streams, devices and services.

7.2 ANSA Deficiencies

The ANSA deficiencies fall into the following areas:

- features lacking from version 2.5 of the Testbench and subsequently added to version 3.0 by the author.
- features lacking from 3.0 which have been added as part of IMAC 3.0.
- conformance is only partially implemented.
- Engineering Model deficiencies.

7.2.1 Testbench Version 2.5 Deficiencies

Version 2.5 of the Testbench lacked facilities for managing references to ANSA interfaces and for application level timers.

7.2.1.1 Interface References

Although IDL supported an `InterfaceRef` data type representing a reference to an ANSA interface, it did not provide a means for explicitly associating instances of this data type with the interfaces to which they refer and hence could not type check their use. `InterfaceRef`'s could, however, be passed as arguments and returned as results of operation invocations.

Another important omission was the lack of a mechanism for dynamically creating interface references which were to be subsequently passed *directly* to other applications; that is, were not to be traded. A possible kludge was to create an `InterfaceRef` by exporting an interface of the appropriate type to the Trader. This was not only inelegant, but led to offers being registered with the trader which were not intended to be generally available.

These omissions represent a major obstacle to implementing an IMAC prototype because of IMAC's heavy reliance on the ability to freely generate and distribute interface references to other applications, streams and devices. Frequent use of interface references in this manner is also likely to lead to an increase in programming errors associated with their incorrect usage; type checking would greatly reduce the occurrence of such errors.

The following sections outline the extensions made to IDL and PREPC to provide the required functionality.[3]

IDL Extensions

The following new statements were added to IDL and are illustrated in figure 7.1:

[3] The syntax shown here is the final form agreed with Joe Sventek for integration into version 3.0 of the Testbench.

```
-- IDL Interfaces

SampleTypes: INTERFACE
BEGIN
  SampleInt: TYPE = INTEGER;
END.

Sample: INTERFACE
NEEDS SampleTypes;
BEGIN
  SampleIfRef: INTERFACEREF OFTYPE Sample;
  Op1: OPERATION [ i: SampleInt ] RETURNS [ j: SampleInt ];
END.

-- PREPC statements, start with an !

! DECLARE { ir } : SampleIfRef SERVER

! { ir } :: Sample$Create( args )

! {} :: Sample$Destroy( ir )
```

Figure 7.1: IDL and PREPC Extensions

`NEEDS InterfaceName`
: a declaration directing STUBC to read the interface specified by `InterfaceName` and add it to the context of the current interface.

`InterfaceRefType: INTERFACEREF OFTYPE InterfaceName`
: a type definition stating that `InterfaceRefType` is of type `InterfaceRef` *and* is used to refer to an instance of the interface `InterfaceName`. Such definitions are passed to PREPC via the signature file, thus allowing PREPC to type check the usage of any `InterfaceRef` variables of this type.

PREPC Extensions

PREPC insists that all `InterfaceRef` variables be declared before they are used. The `DECLARE` statement, shown below, states that the variable will be used to contain an interface reference of the type `InterfaceRefType`, and that it will be used as either client or a server. `InterfaceRefType` must have been defined in an IDL specification.

```
DECLARE { variable } : InterfaceRefType CLIENT
DECLARE { variable } : InterfaceRefType SERVER
```

PREPC is now able to type check the usage of this variable and to detect any programming errors relating to its misuse.

Two further statements, illustrated in figure 7.1, were added to support the creation and destruction of interface references.

`{ variable } :: InterfaceName$Create( optional arguments )`
: Creates a new interface reference to the interface specified by `InterfaceName` and assigns it

to `variable`. The result `variable`, must have been previously declared to be a reference to `InterfaceName`.

{} :: `InterfaceName$Destroy( variable )`
Destroys a previously created interface reference; this is type checked in an identical manner to `Create`.

These extensions also allow user defined state to be associated with each interface reference created. The optional arguments to the `Create` statement, if specified, are passed to a user supplied function which may then allocate interface specific state. This state is then stored in a database keyed by the returned interface reference. Operations are provided by the capsule library for retrieving such state given the interface reference. Similarly a user supplied routine will be invoked to destroy such state when a `Destroy` statement is executed. The user supplied routines, for an interface `IfName`, must be named `IfName_Create` and `IfName_Destroy`.

7.2.1.2 Timer Management

Testbench 2.5 lacked any facilities for application level timers, that is, there was no way for an application thread to suspend itself for a specified period of time, nor to specify an operation to be executed when a timeout period has elapsed. This deficiency makes it difficult to construct applications which implement any form of real-time synchronisation.

A generalised timer facility was implemented to provide both thread suspension and execution of a specified operation on timeout expiry. The Testbench communication system was modified to make use of this generalised timer facility instead of the ad-hoc mechanisms originally used.

7.2.2 Testbench Version 3.0 Deficiencies

The 3.0 implementation of the Trader has two major deficiencies:

1. if an import request can be satisfied by multiple service offers in the Trader's database, the Trader will randomly select *one* of these offers and return that as the result of the import. The client is unable to exercise any choice over the service offer selected.
2. there is no mechanism for the client to obtain the properties specified for the service offer it has obtained. This prevents the client determining which, from the range of possible offers, has been selected.

The first omission makes it impossible to apply any other selection policy than the random one imposed by the Trader. The second deficiency (see section 5.2.6) deprives the client of the opportunity to modify its behaviour to suit the service offer actually obtained. These problems have been overcome as follows:

- the existing Trader **Lookup** operation, responsible for selecting a particular offer, has been enhanced to return the property list specified for the returned offer. Random selection is still applied in the case of multiple offers matching the import request.
- a new `Query` operation is provided which returns *all* of the offers matching an import request. This operation allows an arbitrary selection policy to be applied by its invoker.

The other major restriction associated with trading, identified in section 5.2.6, is the inability for the service provider to determine the constraint expression used to select its offer. This restriction has not been tackled since it would involve extensive changes to components other than the trader. A simple implementation would be for the Trader to invoke an operation on the server to inform it of the constraint used. However, such a solution would not scale well and is therefore rejected. A better scheme would be for the client to include its constraint expression as part of its first invocation on the server.

7.2.3 Implementation of Conformance

Type conformance is only partially implemented in version 3.0, and in particular there is no implementation of the conformance relationships defined in section 5.2.5. Instead, IDL provides a simple mechanism for forcing one interface type to conform to another, but it is left to the programmer to decide if *independently* developed interfaces conform.

`IS COMPATIBLE WITH InterfaceName;`
: states that the interface containing this statement conforms to `InterfaceName` and effectively *copies* `InterfaceName` into the current interface, thus ensuring that the conformance relationship is maintained.

When new interface types are registered with the Trader it is possible to state which other interface types they conform to. In this way, the Trader is able to build a directed acyclic graph of conformance relationships, without requiring any knowledge of the conformance relationship itself.

This scheme is also used in IMAC 3.0; the only extension being that `IS COMPATIBLE WITH` also copies stream related information.

7.2.4 Engineering Model Deficiencies

The ANSA Engineering Model makes two assumptions which become invalid in a multimedia environment:

- bursts of simple interactions will be more common than sustained bulk transfer, therefore latency is the key factor affecting performance.
- communications resources are assumed to be expensive and in order to support scaling such resources must be shared, and therefore multiplexed, wherever possible.

The first assumption has led to the tight coupling of the CPU scheduler and the communications system to ensure that CPU time is quickly scheduled to process communication requests. In addition, memory copying is kept to a minimum and the path from an ANSA interface to the network interface is kept as short as possible. These are all useful features for a multimedia system. However, the final consequence of this assumption is counter-productive, namely that there is no need to explicitly allocate and guarantee resources for the duration of the invocation. Indeed, doing so would be extremely inefficient under the envisaged communication requirements. Unfortunately, multimedia communication requires the guaranteed provision of resources over prolonged periods, and not only for single invocations, but for a succession of invocations.

The second assumption has led to a layered communication system with each layer providing a number of communications channels to be multiplexed between a greater number of higher level

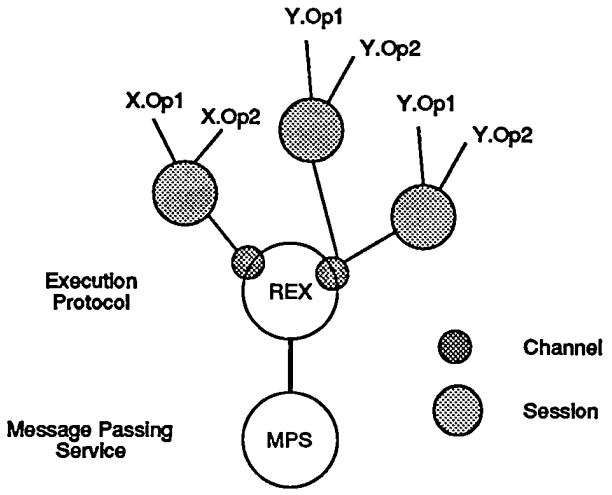

Figure 7.2: Multiplexing in The Testbench

channels. Such a system inevitably suffers from the disadvantages associated with multiplexing discussed in sections 4.2.4.1 and 4.2.4.2.

The following sections describe the design of the ANSA communication system, and the modifications made to overcome its deficiencies in greater detail.

7.2.5 ANSA Communication System

The Testbench relies on the underlying operating system to provide transport protocols, and also assumes that such protocols are accessed via a Berkeley UNIX socket style interface. The currently supported protocols are: UNIX named pipes, UDP, TCP and MSNL [**McAuley89**]. The Testbench implements three protocol layers:

Message Passing Services (MPS): provide an interface to the transport protocols provided by the underlying operating system; there is an MPS for each supported protocol.

Execution Protocols: implement the invocation of ANSA operations. Currently two protocols are defined: the Remote Execution Protocol (REX) for point to point invocations, and the Group Execution Protocol (GEX) for multiway, or group, invocations. A REX implementation is supplied with version 3.0, whilst GEX has only been partially implemented.

Sessions: used to store the end-to-end state required for a remote invocation and to synchronise the execution of the scheduler and the communications system.

Efficient resource utilisation (particularly of memory) is achieved by multiplexing the channels provided by each of these between those of the next layer.

Each MPS provides a *single* channel to each execution protocol. MPS's provide a stateless interface, and rely on the execution protocol to provide complete addressing information for each message transmitted, and on the underlying operating system for each message received.

Execution protocols provide *channels* for issuing operations to a specified remote interface and for receiving invocations on a specified interface. These are called *plug* and *socket* channels respectively.

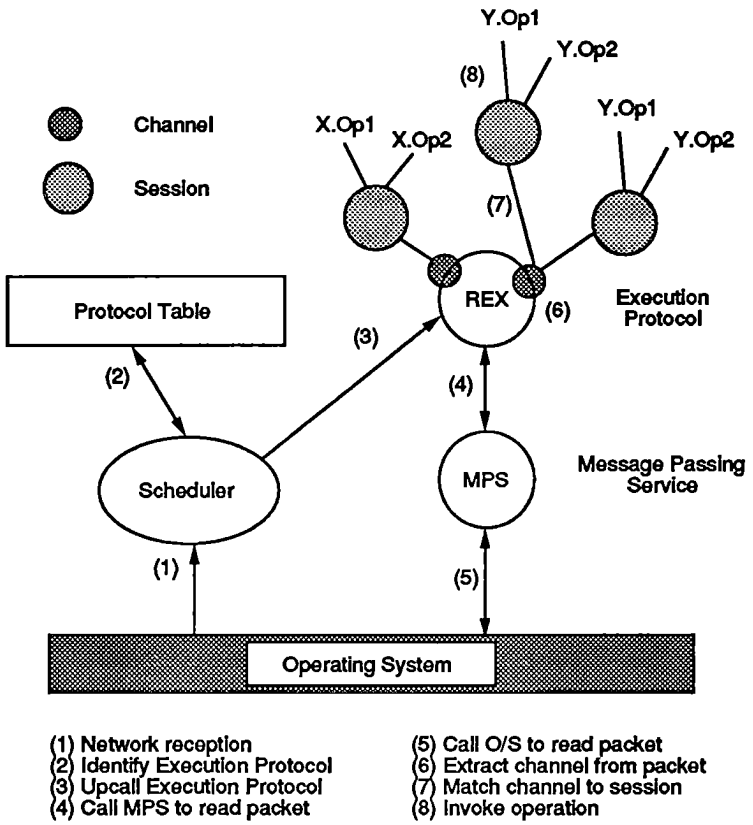

Figure 7.3: Network Reception in The Testbench

There is a one-to-one correspondence between channels and interfaces. The terms *plug* and *socket* are not intended to imply uni-directional communication, but rather that sockets may be used to wait for un-solicited receptions, whilst plugs cannot. Servers transmit invocation replies over sockets, and clients receive replies over plugs.

Sessions are created dynamically for each client/server interaction pair; therefore, a different session is required for each client invoking operations in a single interface. Channels are multiplexed between all of the *sessions* supported by the interface in question. Therefore, a server providing three interfaces, each being used by two clients, requires three sockets and six sessions. Sessions are shared across all of the operations in an interface, that is, all operations invoked by a particular client on the same interface will use the same session. Figure 7.2 illustrates this structure for a server supporting two interfaces, X, Y, each providing two operations. Interface X is being used by one client, whilst Y is used by two clients.

A new session is created on receipt of the first invocation from a previously *unseen*[4] client, and an identifier for the newly created session is passed back to the client in the reply to the invocation. This identifier is then included in subsequent invocations from that client.

Sessions are decayed, and if no invocations appear on a session for a pre-determined period of time, the session is garbage collected. The exact duration of this period depends on the precise semantics of the execution protocol and on the maximum time for which the lower level transport protocols are likely to retransmit the same packet; that is, a session must be kept for at least as long as it is

[4] In this context *unseen* means a client which does not include a specific session identifier in its invocation.

possible to receive a retransmitted packet from the last invocation. This scheme avoids the need for an explicit message exchange to create and destroy session state and thus reduces latency.

The process of creating sessions is called *binding*, and the time at which sessions are created (i.e. when binding occurs) is referred to as the *bind-time*. A client can force the server to create a new session by issuing a new invocation which does *not* include a specific session identifier.

At the lowest level, the scheduler waits for new network receptions and uses a data structure called the *protocol table* to associate network receptions on a particular Berkeley socket (not to be confused with channel sockets) with an execution protocol. On being called by the scheduler the execution protocol invokes the appropriate MPS to actually obtain the newly received packet and subsequently decodes the address fields in that packet to determine the source and destination end points. It may then match the destination end point to a channel socket, which may in turn be matched to an existing session, if one was specified in the packet, or cause a new session to be created otherwise. Figure 7.3 illustrates this process.

This design suffers from the following disadvantages:

- excessive asynchronous multiplexing, which inevitably introduces jitter.
- the stateless MPS interface does not allow for the association of QoS with a higher level session or channel. Although the QoS request could be included in every transmission request this would be prohibitively expensive, and in any case does not solve the problem of specifying QoS for network receptions or for multiple invocations.
- it is assumed that all operations in an interface use the same session, thus imposing the constraint that any QoS provided be on an interface wide basis.

The prototype implementation attempts to overcome each of these problems as follows:

- a new MPS interface has been designed explicitly to support QoS and to allow multiplexing to be reduced. This interface is based on a new communication abstraction called a *Local Channel Resource*, and is described in section 7.2.5.1.
- the restriction that all operations in an interface use the same session is not inherent in the design of the communications system, but is an assumption made by the stub code to invoke the underlying communication system. STUBC has been modified to use separate sessions for operations which specify QoS constraints.
- sessions provide a convenient vehicle for storing the results of QoS negotiation and also allow QoS to be associated with a succession of invocations. Whenever a session is decayed, any associated resources are freed, and whenever a new session is created, QoS must be re-negotiated. Section 7.3 describes the implementation of QoS in more detail.

7.2.5.1 Local Channel Resources

An important design goal for the new MPS interface was that the provision of QoS should not adversely affect the performance and scaling properties of communication *not* requiring QoS. Local Channel Resources (LCR's) were designed to provide a compromise between the scaling properties provided by multiplexing on the one hand, and the desire to minimise multiplexing and provide QoS on the other.

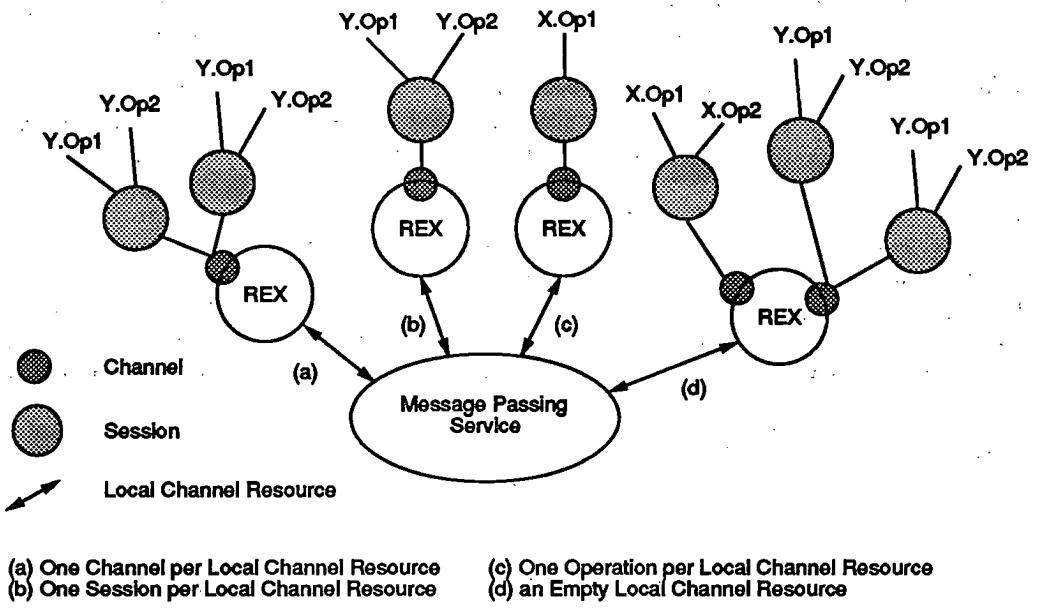

Figure 7.4: Multiplexing Using Local Channel Resources

Local Channel Resource Design

Local channel resources represent the *local* communication *resources* required to guarantee a given QoS; they do *not* represent communication end-points and do *not* contain any addressing information. Therefore, it is impossible to transmit data to, or receive data from, an LCR. It is left to higher level protocols to make use of the resources provided by LCR's to implement data transmission and reception. Similarly LCR's do *not* provide end-to-end QoS, again, it is left to higher level protocols to provide end-to-end QoS by creating appropriate LCR's at the source, destination and any intermediate capsules.

LCR's may support any degree of multiplexing ranging from the general case of multiplexing at all levels, to providing no multiplexing whatsoever. A single LCR may be multiplexed between multiple higher level plugs or sockets, alternatively a single LCR may be reserved for a single plug or socket. Each such channel may, or may not, be multiplexed between multiple sessions. In this way it is possible to vary the degree of multiplexing to suit higher level QoS requirements, and allows multiplexing to be viewed as a particular QoS attribute. A capsule may receive requests to create new, non-multiplexed, LCR's over an existing multiplexed LCR. This forms the basis for implementing end-to-end QoS as described in section 7.3.3.

It is also possible for an LCR to provide no particular QoS, and hence no resource guarantees; such LCR's are referred to as being *empty*. Empty LCR's will typically be multiplexed between multiple channels and provide identical functionality to that provided by the existing, stateless, MPS interface. In IMAC 3.0 each execution protocol creates a single empty LCR which it uses for *all* invocations which do not require a specific QoS.[5] In this way, there is no scaling penalty to be paid for such operations. The smaller the degree to which LCR's are multiplexed, the greater the cost incurred, that is, the greater the number of LCR's required. Each active LCR is likely to require the allocation of a transport level protocol port and associated resources in the underlying operating system, in addition to the resources used within the capsule.

[5] Requests to create new, non-multiplexed, LCR's will be received using this LCR.

Figure 7.4 shows the various multiplexing combinations which may be provided at the server using LCR's.

Whenever a new LCR is created, precautions must be taken to ensure that the provision of the requested resources do not overcommit the communication system and therefore potentially compromise the guarantees made for existing LCR's. Empty LCR's, which have no QoS requirement of their own, must defer resources to other, non-empty, LCR's.

Finally, LCR's can be used for both data transmission and reception; that is, they are bi-directional. However, there is no restriction that the same QoS be provided for both directions, it is left to the QoS specification to state the resources required for each direction. Allowing duplex communication allows for the efficient use of underlying transport protocols which also provide bi-directional communication. The alternative, of using two uni-directional LCR's, would force the use of two underlying communication channels where one would suffice.

Local Channel Resource Interface

The Local Channel Resource Interface (LCRI) provided as part of IMAC 3.0 provides a uniform interface to the underlying transport protocols and allows full advantage to be taken of both connection-oriented and connection-less protocols. The principal features of the LCRI are as follows:

- the LCRI supports two types of LCR: namely, *client* and *server* LCR's. Both may be used to send and receive data, but only server LCR's may be used to accept new network connections.
- the creator of a server LCR must specify an upcall to be invoked when a new connection is accepted. The arguments to this procedure include an indication of the newly created LCR, it is then possible to specify another procedure to be upcalled when data is received on the new LCR.
- the operation for creating client LCR's includes an implicit connection request, which need only be implemented by connection-oriented MPS's.
- all transmission and reception operations include full addressing information, thus allowing the use of a single LCR to be used to communicate with multiple sites. This follows directly from the fact that LCR's do not represent communication end-points.
- a cost function is defined to return an indication of the cost of providing a given QoS.

The ability to use server LCR's to accept connections maps directly to the functionality provided by connection-oriented protocols. It can also be simulated by a connection-less protocol by arranging for its MPS to invoke the specified upcall when the LCR is created, as opposed to when a new connection is accepted. The only problem is that of deciding how to obtain an LCR for the supposedly newly accepted connection; this is solved by re-using the original server LCR, that is, the server LCR is specified as the new LCR.

The fact that transmission and reception operations include full addressing information allows a single LCR, regardless of the underlying protocol, to be multiplexed between multiple higher level communication channels. In this way, it is left to the higher level protocols to decide how to multiplex the resources provided by an LCR.

One restriction is applied to the use of server LCR's in order to allow for the implementation of end-to-end QoS by higher level protocols. That is, a non-empty LCR is constrained to only accept a connection request from a specified end-point or end-points. Section 7.3.3 describes the

implementation of end-to-end QoS in detail. On the other hand, an empty LCR may listen for connections from *any* remote end point.

Both client and server LCR's are created to satisfy a particular QoS offer and dynamic QoS negotiation is assumed to have taken place *before* this interface is invoked, (see section 7.3.2). The negotiation algorithm used will base its decision as to which QoS offer to use on the results of the cost function provided by the LCRI.

Another important feature of the LCRI is that when an LCR is created, the creator may associate some data with that LCR, which will be subsequently made available as an argument to both the connection accepted and data received upcalls. This data may then be used to speed up the mapping from LCR to higher level protocol state and thus improve performance. In addition, if this mapping can be implemented in constant time, then jitter may also be reduced.

QoS provision offers significant advantages even for connection-less protocols. For instance, it is now possible to create multiple LCR's over a connection-less or a connection-oriented MPS and use each of these to communicate with a different capsule, using a different transport channel and QoS for each. Not only does this allow for QoS to be provided, but it also reduces the amount of multiplexing required. It is also possible for each MPS to gather per-LCR performance statistics, which can be used by higher level protocols to optimise their performance.

Local Channel Resource Implementation

The existing MPS interface has been replaced by one based on Local Channel Resources, and the UNIX named pipe, UDP and TCP MPS modules have been re-implemented to use this new interface. The differences between the new and previous implementations are as follows:

- the scheduler now calls the appropriate MPS module, rather than the execution protocol, to process network receptions.
- each LCR uses a separate Berkeley socket and network address. This allows an array lookup to be used to map Berkeley sockets to LCR's and hence provide constant time demultiplexing of incoming data. In addition, non-empty LCR's are given priority over empty ones, that is, if network receptions are pending for both an empty and a non-empty LCR, then the reception for the non-empty LCR will be serviced first.
- REX associates a channel identifier with each non-empty LCR it uses. This is then used to implement a constant time lookup for matching data received over LCR's to channel sockets. The previous implementation used a complex search algorithm to match incoming data to channels, and as a result the new implementation should provide better performance over non-empty LCR's.
- QoS may be used to control the policy applied when the finite number of available Berkeley sockets is exhausted. LCR's may be designated as *permanent*, in which case their sockets are never re-used, whilst non-permanent LCR's have their sockets re-cycled on a least recently used basis.

The prototype implementation also imposes a number of restrictions:

- it does not support the multiplexing of multiple sessions over a single channel for non-empty LCR's; that is, there is a one-to-one correspondence between channels and sessions when they are used to provide a particular QoS. This simplifies the implementation, without suffering any great loss of flexibility.

- as a result of the use of UNIX, the only resources managed are Berkeley sockets and the cost of providing a given QoS is measured as the number of sockets required to realise it, (i.e. 0 if an existing socket can be re-used or 1 if a new socket is required).

Local Channel Resource Summary

The design of the ANSA communications system has been reviewed and its shortcomings identified. A new MPS interface has been designed and implemented to overcome these shortcomings. It provides a new communication abstraction, called a *Local Channel Resource*, which provides a means of associating a particular QoS with the communications resources required to realise and guarantee that QoS. The new interface combines the features of connection-less and connection-oriented transport protocols to provide a QoS based interface with flexible multiplexing options. This combination of connection-oriented and connection-less protocol interfaces is made possible by the fact that LCR's do not represent communication end-points, but rather communication resources at end-points. By so doing it is possible to provide QoS without adversely affecting the performance and scaling properties of communication not requiring QoS.

The prototype implementation only offers a limited range of QoS, but reduces multiplexing, and also jitter, through the use of constant time lookups for incoming receptions.

7.3 QoS

This section describes the QoS representation and negotiation algorithms used, the implementation of QoS within the higher levels of the communication system, the provision of end-to-end QoS and the programming interface provided to the underlying QoS implementation.

The prototype implementation supports two QoS layers: one layer for execution protocols and one for MPS modules. However, there are no restrictions placed on the number of QoS domains provided, and hence there may be multiple instances of these layers.

The algorithms and data structures discussed below do not make any assumptions as to the number of layers used and are designed to cope with an arbitrary number of layers.

7.3.1 QoS Representation and Negotiation

There are two distinct representations used for QoS within the Testbench. The first is used to specify QoS offers and constraints, and for matching QoS constraints to offers; this representation is described in section 7.3.1.1. It is used to implement per-layer QoS negotiation as defined in section 5.2.7.

The second is concerned with representing the QoS required by a given interface in a suitable form for communication to *potential* clients of that interface. The word *potential* is used to indicate that such clients will be able to invoke operations on the interface in question, if, and only if, they can provide the QoS required by the interface and its operations. ANSA interface references are the logical place to include such QoS information, and their extension to include such information is described in section 7.3.1.2.

```
-- 2 QoS macros

"VideoProtocol" "x,y,z"  "(Rate >= x and Delay <= y and Encoding == 'z')"
"AudioProtocol" "x,y"    "(Rate >= x and Delay <= y)"

-- 3 QoS offers and associated suppliers

"Name Video VideoProtocol '1000,1,pal'" "video_1"
"Name Video Type Cheap VideoProtocol '100,10,pal'" "video_2"
"Name Audio AudioProtocol '10,15'" "audio_1"

-- QoS constraints and results

"Name == Video"  -- returns two offers

  "Name Video VideoProtocol '1000,1,pal'"              - original offer
  "(Rate >= 1000 and Delay <= 1 and Encoding == 'pal')" - expanded offer
  "video_1"                                            - supplier

  "Name Video VideoProtocol Type Cheap '100,10,pal'"   - original offer
  "(Rate >= 100 and Delay <= 10 and Encoding == 'pal')" - expanded offer
  "video_2"                                            - supplier

"Name == Audio"  -- returns one offer

  "Name Audio AudioProtocol '10,15'" - original offer
  "(Rate >= 10 and Delay <= 15)"      - expanded offer
  "audio_1"                          - supplier
```

Figure 7.5: Per-Layer QoS Negotiation Example

7.3.1.1 Per-Layer QoS Negotiation

The experimental nature of this implementation coupled with the wide range of uses to which QoS could be put by applications, requires as flexible a representation as possible. The representation used by the Trader for specifying interface properties and constraints was chosen as a suitable starting point for QoS representation. It was chosen primarily because of its ability to support arithmetic comparison and boolean connectives to link sub-expressions. Appendix B includes a definition of the Trader constraint language. The implementation of this language has been separated out from the rest of the Trader and made available for wider use.

QoS offers are specified as a list of (name, value) pairs, called properties. Associated with each QoS offer is a textual string identifying its supplier. Property values may be textual strings, sets of textual strings or numbers.

Constraint expressions are used to search a set of QoS offers for matching offers. The result of a constraint expression is the set of all offers matching that constraint. Constraint expressions may include tests for equality and inequality of string values, set membership and numeric comparison. Boolean connectives may be used to link sub-expressions, whilst minimum and maximum operators are provided for numeric expressions.

In order to support the mapping of QoS constraints from one layer to another (see section 5.2.6) property lists and constraints have been extended to provide a simple macro facility, illustrated in figure 7.5 and defined below:

Macro Definition: macro definitions have three parts:

1. the macro's name.
2. an optional list of comma separated formal parameters.
3. the macro body, specified as a textual string, containing instances of the formal parameters which will be replaced by the actual parameters when the macro is expanded.

Macro Properties: macros can be specified as properties, with the macro's name specified as the property name and its actual parameters specified as a string value for that property. Individual parameters are separated by commas; for instance, two parameters `a` and `b` would appear as `'a,b'`.

Constraints: macros are treated as string properties when evaluating constraint expressions.

Macro Expansion: macros are expanded when they are specified as a property in a QoS offer. Expansion consists of replacing all instances of the formal parameters in the macro body with the corresponding actual parameter specified in the property value. All macros appearing in a QoS offer are expanded and the results of each expansion concatenated to form a single expanded string. This string is made available along with its corresponding, un-expanded, offer as part of the result of evaluating a constraint expression. Non-macro properties are *not* copied into the expanded string.

The result of a constraint expression is a list of matched offers, each of which includes three items:

QoS offer: specified as property list.

Expanded QoS: the result of expanding any macros specified in the QoS offer.

QoS supplier: a textual string identifying the supplier of this offer.

Given that the intended use for macros is to map QoS constraints at one layer to constraints at the next, the body of the macros used will take the form of a constraint expression.[6] Figure 7.5 gives a complete example of per-layer QoS negotiation.

Due to the size of the code and data required to implement the constraint language it is implemented by the QoS Manager and *not within* each capsule. This scheme incurs the overhead of a remote invocation for every constraint expression evaluated. However, because QoS offers change slowly over time (only when the system is re-configured) it is possible to cache the results of previously evaluated, or well known, constraint expressions. The QoS Cache is described in section 7.3.1.4.

To implement per-layer QoS negotiation all that remains to be provided is for QoS offers and macros to be associated with a single QoS layer and for constraint expressions to be evaluated within the scope of that layer. The QoS Manager and QoS Cache are responsible for implementing QoS layers.

7.3.1.2 QoS Interface References

The data type used to represent ANSA interface references is called an `InterfaceRef`. In version 3.0 it consisted of a unique identifier for the interface referred to, and a sequence of addresses (called an `AddressHint`) that the interface may be invoked on. Each such address (called an `AddressRecord`) contains a channel identifier and MPS address. Although this supports the use of a separate channel over each MPS, version 3.0 always used the same channel over all MPS's.

[6] This is not enforced by the macro facility itself, but is a convention adopted for its use.

This structure has been extended in IMAC 3.0 to include a description of all of the QoS offers which may be used to provide the QoS specified for the interface. As shown in section 7.3.4, QoS constraints may be specified for all or individual operations within interfaces. These QoS specifications are then negotiated, using the algorithm defined in section 7.3.1.3, to determine all the QoS offers (or protocol stacks) which may be used to provide the required QoS. It is these offers which are encoded in the new `InterfaceRef` data type, and communicated to clients of the interface in question, via an instance of this data type. The `AddressHint` field is now a sequence of `ExtendedAddressRecords`, each of which contains the following sub-structures:

`AddressRecord`: identical to that used in the original `InterfaceRef`.

`ProtocolStack`: a list of protocol layers, each of which consists of:

- `LayerName`: the name for this layer.
- `ProtocolOfferList`: a list of QoS offers which may be used at this layer, where each protocol offer contains the following:
  - `Offer`: the QoS offer itself.
  - `Supplier`: the supplier of this offer.

`OperationList`: the list of operations to which this QoS applies.

On receipt of such an `InterfaceRef` the client must match the QoS offers it wishes to use at each layer against those available at the server interface. If no match exists then communication is not possible, if a match does exist then communication may be possible, but is subject to the negotiation of dynamic and end-to-end QoS as described in subsequent sections.

A set of minimal QoS offers are defined, for which no particular QoS resources are required, but which identify execution and MPS protocols. In this way, it is possible to encode the *configuration* of the server's communication system in an `InterfaceRef`. These offers consist of a single property, `Name`, whose value is the name of an execution or MPS protocol, (i.e. one of `REX`, `IPC`[7], `UDP`, `TCP`, or `MSNL`). The client is now able to avoid attempting to use an unsupported protocol and even to choose the most efficient protocol for its requirements. Such an `AddressHint` is referred to as the *default* `AddressHint` and is used for operations which do not include a particular QoS specification. For instance, a capsule supporting REX, and three MPS modules, requires a default `AddressHint` consisting of three `ExtendedAddressRecords`, one for each possible combination of REX and MPS protocol.

7.3.1.3 Static QoS Negotiation

This section describes how *static* QoS negotiation involving multiple layers is implemented. The algorithms outlined in section 5.2.7.1 are followed, but only the static results of per-layer QoS negotiation are considered. This allows the complete set of protocol stacks which can potentially be used to provide a given QoS to be determined, but does not attempt to realise any of these stacks. Sections 7.3.2 and 7.3.3 describe dynamic and end-to-end QoS negotiation and show how one of these stacks is selected and realised.

At the server, the algorithm for QoS *specifications* (given in figure 5.6) is executed, and the result for each constraint is represented as an `AddressHint`, containing as many `ExtendedAddressRecord` structures as there are combinations of matching offers at each supported protocol layer. All operations without an associated QoS specification are grouped together and have a *default*

[7] UNIX named pipes.

`AddressHint` constructed for them. The resulting series of `AddressHints` are then merged into a single `AddressHint` for inclusion in an `InterfaceRef`. The server's algorithm is executed whenever a new interface reference is created.

The client implements the algorithm given in figure 5.7 for QoS *requests*, with one important optimisation. Given that the client knows all the QoS offers that can be used to satisfy the server's QoS specification (contained in the server's `InterfaceRef`) in addition to knowing all of the locally supported QoS offers, it can determine which, if any, of the server's offers it could possibly use. In this way, it can decide whether communication is possible with the server in advance of executing the QoS request algorithm. This avoids the need for per-layer QoS negotiation and possible interaction with the remote QoS Manager. Therefore, no great cost is incurred in situations where communication is impossible. In the case of *default* `AddressHint`'s only this algorithm need be executed, and not the QoS request algorithm. Therefore, operations which do not require QoS do not pay a performance penalty for its use by other operations.

Dynamic and end-to-end QoS negotiation are also performed from within the client's algorithm, but are described separately, in sections 7.3.2 and 7.3.3, in order to simplify their explanation. The client's algorithm is executed whenever the client re-binds to the server.

7.3.1.4 QoS Cache and Configuration

The QoS Cache provides a per-capsule cache of QoS offers and constraints known to match those offers, and is used to reduce the number of invocations made to the QoS Manager. The QoS Cache is structured as a set of domains, each consisting of a linked list of named QoS layers. The name of a domain is the name of its uppermost QoS layer.[8] The QoS Cache is implemented in two parts:

Simple Cache: the low-level management and database portion which supports the creation of new layers and the registration of offers with each layer. For each offer, there is a list of constraints which are *known* to be satisfied by this offer. A constraint request for a given layer, is implemented by searching the constraint lists for each offer in the specified layer, for constraints which are *identical* to that specified in the request. It is also possible to cache negative results, that is, to specify a constraint that is known to have failed to match an offer. This scheme means that the simple cache has no knowledge whatsoever of the constraint language, and uses only string comparison to identify matches.

Transparent Cache: provides transparent access to the underlying simple cache, or the QoS Manager in the case of a cache miss. Having invoked the QoS Manager to execute the constraint expression it is responsible for loading the simple cache with constraints known to match the resulting QoS offers. If the expression failed, then the specified constraint is cached as having failed to match *all* offers in the specified layer. In this way, all future requests for the same constraint can be satisfied from the simple cache.

The current implementation of the QoS Cache does not provide any means for invalidating existing cache entries; that is, it assumes that the QoS configuration does *not* change during the lifetime of a capsule. This is justified by the observation that the configuration of capsule's communication system can only be changed at compile or link time and hence cannot be changed whilst the capsule is actually running.

The configuration of a capsule's communication system has two components and is represented by the QoS offers it supports:

[8] The same structure is used for the QoS Manager.

Default Configuration: defined at compile time, and used to construct the default `AddressHint` for this capsule. This information is stored in the well known domain *ex.ansa*[9], consisting of the layers *ex.ansa* and *mps.ansa*. These offers are pre-loaded into the simple QoS cache, and since their format is known (i.e. a single property stating the name of execution or MPS protocol) it is also possible to register constraints which are known to match these offers.

Full Configuration: the complete list of QoS offers supported by this capsule, in all other domains, and read from a text file during capsule initialisation. These offers are pre-loaded into the simple QoS cache, but no matching constraints are known at this time, so any attempts to match against these offers will result in an invocation on the QoS Manager.

In order for this scheme to work, a consistent set of QoS domains, layers and offers, must be maintained between the QoS cache in individual capsules and the QoS Manager. In the prototype implementation consistency is maintained manually and an automated scheme would be required for a more widely used implementation. For instance, a set of system administration tools could be provided which interrogate the QoS Manager and examine the configuration files used by capsules for inconsistencies. A better scheme would be to automatically generate new configuration files whenever the QoS Manager's database is updated. Ensuring the consistency of the default configuration is easier than for the full configuration since it will change less frequently.

7.3.2 Dynamic QoS Negotiation

Dynamic QoS negotiation is concerned with selecting the most appropriate QoS offer, from a list of candidate QoS offers, based on current resource utilisation. It is therefore responsible for managing any resources which have a bearing on QoS provision. Within IMAC 3.0, such resource management is implemented using *cost functions*, that is, each QoS supplier provides a function for determining the cost of providing a given QoS offer. In this way, it is possible to keep track of the cost of the resources allocated so far and use this information to implement a range of resource management policies, such as simple load-balancing across the various QoS suppliers. In addition, if the total cost of all the available resources is known, the incremental cost of providing a QoS offer can be determined and used to implement system-wide resource management. In this way it is possible to determine the relative cost of providing a given QoS with each supplier.

Dynamic QoS negotiation is invoked from within the QoS request algorithm (sections 7.3.1.3 and 5.7) at each QoS layer encountered.

The prototype implementation defines a standard interface for dynamic QoS negotiation with the following operations:

QoSReserve: selects a single QoS offer from a list of QoS offers, and tentatively allocates the resources required to guarantee it. A data structure, called a `QoSHandle` is returned, which identifies the resources allocated, and is used as an argument to all the other operations in this interface.

QoSCancel: cancels a previous reservation and makes any reserved resources available for re-use.

QoSConfirm: confirms a reservation and allocates the associated resources. A time-out period may be specified for the maximum time that the allocated resources may remain unused. If this period is exceeded then they may be freed and made available for re-use. In this way, resources remain allocated until a `QoSDiscard` operation is executed or the time-out expires. A successful call to `QoSReserve` does not necessarily mean that `QoSConfirm` will also succeed.

[9] The name of a domain is the name of its uppermost layer.

QoSDiscard: discards a confirmed reservation and frees the associated resources.

The use of separate operations for resource reservation and confirmation allows for a number of resource optimisations to be made. For instance, resources may be overcommitted during the reservation phase, on the assumption that some reservations will be subsequently cancelled. In addition, given the delay introduced by the remote invocation of `Bind`, it is possible to receive subsequent QoS requests, whose requirements may be taken into account before confirming previously reserved resources. In the extreme, later reservations may preempt earlier ones. This is made possible by delaying irreversible resource management decisions to the latest possible instant. Thus, even if `QoSReserve` succeeds, `QoSConfirm` may fail, but if `QoSConfirm` succeeds then the resources are guaranteed for as long as they are required.

Two instances of this interface are provided, one each for the execution and MPS protocol layers. The execution protocol instance manages the allocation of channel plugs and sockets, whilst the MPS instance manages LCR's. The current implementation is constrained by the use of UNIX as the underlying operating system and implements only rudimentary resource management. The execution protocols do not provide a cost function, since the only resources they consume are memory, whose cost is negligible over UNIX.[10] The cost functions provided by the MPS reflect the number of Berkeley sockets required to realise the QoS offer, and will be either 0 if an existing socket can be re-used or 1 if a new socket is required. In this way it is possible to ensure a fair distribution of the available sockets across MPS's.

However, the use of a uniform interface for dynamic QoS negotiation allows more elaborate policies to be easily implemented in the future. In addition, it is possible to change the underlying implementation of the communication system without affecting the QoS negotiation algorithms and their implementation.

7.3.3 End-to-End QoS

This section describes how the previously described algorithms for negotiating QoS and the interface for dynamic QoS negotiation are used to implement end-to-end QoS.

The client side dynamic QoS negotiation algorithm selects a single QoS offer at each supported protocol layer and tentatively allocates the resources required to provide that QoS.

In order to implement end-to-end QoS, a means must be found for communicating the QoS offers actually selected by the client to the server with which it wishes to communicate. Such communication can be implemented *in-band*, that is, the selected QoS offers are contained within the invocations issued to the server. Alternatively, an *out-of-band* approach may be taken in which the QoS offers are communicated separately from the invocations to which they refer and are used to create a separate communications channel over which subsequent invocations may be issued.

The in-band scheme avoids the need for a separate QoS communication stage and since the required QoS is encoded in every invocation, the server does not need to maintain any QoS related state. However, the protocol overhead, both in terms of data communicated and processing time at the server, is increased, because the QoS is contained in every invocation and processed on a per-invocation basis. In addition, such a scheme hinders the pre-allocation of resources and results either in all resources being allocated on demand as invocations are received or with certain resources being cached in anticipation of their subsequent use. Both of these factors increase latency, and if resources are allocated solely on demand,[11] or on cache miss, then jitter is also likely to be

[10] Providing the system is not thrashing.

[11] The demands for resources will inevitably be asynchronous.

```
Capsule: INTERFACE =
BEGIN

  BindStatus: TYPE = { C_F, C_S };
  BindReason: TYPE = CARDINAL;
  BindRecord: TYPE = RECORD [ destAddress:AddressRecord ];

  BindResult: TYPE = CHOICE BindStatus OF {
    C_F =>BindReason, C_S =>BindRecord
  };

  Bind: OPERATION [
    destIfId:InterfaceId;
    exProtocol:CARDINAL; exOffer:STRING;
    mpsProtocol:CARDINAL; mpsOffer: STRING;
    qosRequest: STRING
  ] RETURNS [ BindResult ];

  UnBind: OPERATION [ binding: BindRecord ] RETURNS [];

END
```

Figure 7.6: Bind and UnBind Operations

increased.

Within IMAC 3.0 the out-of-band approach has been adopted for the following reasons:

1. it allows for the easy pre-allocation of resources.
2. it minimises per-invocation communication and processing overhead.
3. it is possible to use an existing execution protocol, recursively, to communicate with the server. The only restriction is that the recursive call cannot make use of QoS.

The ability to re-use an existing execution protocol is particularly powerful since it provides well defined communication semantics and allows the full flexibility available to the application programmer to be used from within the communication system. Consequently it is possible to describe the operations provided using IDL and use the stub code generated by STUBC.

The disadvantage is that a separate invocation is required, thus increasing the total latency of operation invocation. However, this can be minimised by only issuing such an invocation when QoS is re-negotiated, and hence only incurring the associated overhead for the first of a sequence of invocations. Given that multimedia communication is likely to be long lived then such re-binding will be required relatively infrequently and therefore its cost can be amortised over a large number of invocations.

Two operations are provided by every server capsule, called **Bind** and **UnBind**; figure 7.6 contains the IDL definitions for these operations. The end-to-end QoS algorithm is illustrated in figure 7.7, and summarised below:

1. in order to invoke the **Bind** operation a channel identifier and MPS address must somehow be obtained for it. In addition, an interface identifier is required for the interface to be subsequently invoked. Both of these items can be found in the **AddressHint** component of the original **InterfaceRef**.

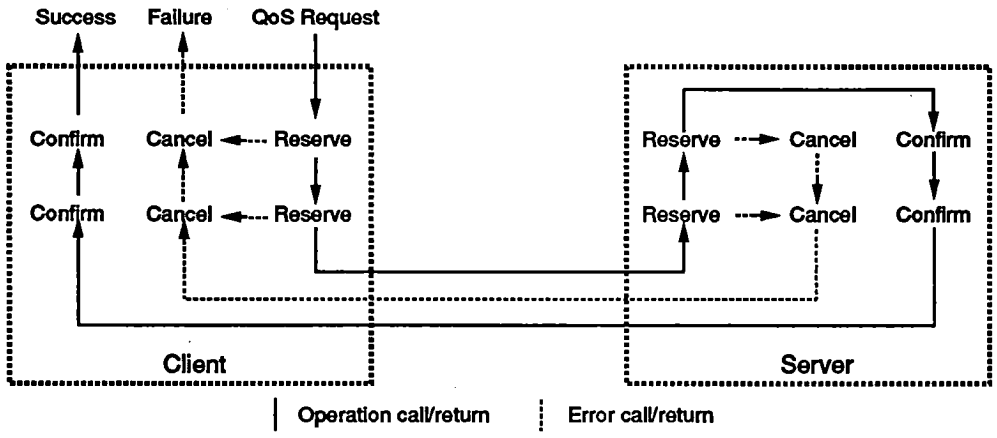

Figure 7.7: End-to-End Resource Allocation

2. at the client, `QoSReserve` is invoked[12] first for the execution protocol and then for the MPS modules, to select a single QoS offer and reserve any required resources.

3. the QoS offers for each layer, the original QoS constraint and the interface identifier are passed as arguments to the `Bind` operation. The QoS constraint is included so that it may be made available to the application implementing the operation to be subsequently invoked.

4. the implementation of `Bind` invokes `QoSReserve` to allocate the necessary resources at the server. If successful, `QoSConfirm` is used to confirm these resources, otherwise `QoSCancel` is used to cancel them. The MPS operation is invoked first and then the execution protocol, thus resources are allocated bottom-up at the server.

5. `Bind` either returns an error condition or a completed `BindRecord` if successful.

6. the client examines the result of its `Bind` invocation, and if it failed, the previously reserved resources are cancelled, otherwise they are confirmed. If a confirmation fails then the client must call `UnBind` to free the confirmed resources at the server. The information in the `BindRecord` is copied into the client's channel and session data structures, and from there into the header of subsequent invocations.

7. the client issues invocations on the server.

8. to un-bind, the client calls the server's `UnBind` operation, with the `BindRecord` that was returned by the original call to `Bind`. If the client fails to call `UnBind` then the server will free any allocated resources when the timeout specified when the resources were confirmed expires.

In version 3.0 every invocation of `Bind` or `UnBind` would result in the creation of a new session in the server, thus *doubling* the number of sessions required. In order to avoid this overhead, and thus improve scaling, IMAC 3.0 provides a new form of channel, called a *promiscuous* channel. Promiscuous channels support a *single* session, which is multiplexed between *all* clients of its associated interface. In this way, only a single session is required for `Bind` and `UnBind` regardless of the number of clients that invoke them. The use of a large number of sessions represents a scaling problem for the current Testbench because of the slow rate at which it decays idle sessions; it takes 30 minutes to discard an unused session. An alternative solution would be to increase the rate at

[12] From within the QoS request negotiation algorithm.

```
SameQoS: [ "Name == 'Video'" | "Name == 'CheapVideo'" ] INTERFACE =
BEGIN
  Same1: OPERATION [] RETURNS [];
  Same2: OPERATION [] RETURNS [];
END

DifferentQoS: INTERFACE =
BEGIN
  Diff1: [ "Name == 'Video'" ] OPERATION [] RETURNS [];
  Diff2: [ "Name == 'Audio'" ] OPERATION [] RETURNS [];
END
```

Figure 7.8: IDL QoS Specification

which sessions are decayed; unfortunately, this would require that servers be able to communicate with their clients to establish the validity of the session being requested.

7.3.4 QoS Programming Interface

This section describes the interface presented to the application programmer for stating QoS specifications and requests. IDL has been extended to support QoS specification for either individual operations, or for entire interfaces. If the same QoS is required for a number of separate operations then they must be placed in a separate IDL interface and the required QoS specified for that interface.

QoS specifications take the form of a series of constraint expressions, (see section 7.3.1.1); multiple constraint expressions are separated by a vertical bar (|) symbol. Such QoS specifications may appear either in the definition of a single operation, or at the head of an interface definition. Figure 7.8 contains two interfaces, the first of which defines two operations, both of which will use the same *encompassing* QoS, and hence the same communication channel. The QoS specification for these operations allows the use of either a `Video` or `CheapVideo` QoS. The second interface contains operations with individual QoS specifications; `Diff1` supports the use of a `Video` QoS, whilst `Diff2` requires an `Audio` QoS. STUBC generates the code required to interface with the underlying QoS specification algorithm implementation, and thus shields the application programmer from the implementation details of this algorithm.

If QoS specifications exist for both the interface and individual operations, then QoS for the individual operations overrides that for the interface. Operations which do not provide a QoS specification are referred to as requiring an *empty* QoS. Such operations will be issued and received using an empty LCR and will, therefore, be multiplexed.

The QoS domain and layer in which QoS specifications are evaluated, is specified as an argument to STUBC, thus allowing for the system to be re-configured without the need to change interface definitions.

Invocations of IDL operations are written as PREPC statements and preprocessed to generate C code which invokes the stub code generated by STUBC. The PREPC invocation statement has been extended to include a QoS request. This takes the form of a single constraint expression, specified as either a C string constant (i.e. enclosed in double quotes) or as a C string variable. The fact that the QoS request may not be known until run-time precludes compile-time checking; in this instance, the flexibility of being able to dynamically select the QoS to use at run-time outweighs the benefits of compile-time checking. In particular, this allows clients to dynamically

```
! DECLARE { same } : SameQoS CLIENT
! DECLARE { diff } : DifferentQoS CLIENT

! { same } <- traderRef$Import( "SameQoS", "/", "" )

! {} <- same$Op1() [ "Name == 'CheapVideo'" ]
! {} <- same$Op1() [ "Name == 'CheapVideo'" ]

! same$Discard

! {} <- same$Op1() [ "Name == 'Video'" ]
! {} <- same$Op2() [ "Name == 'Video'" ]

! same$Discard

! { diff } <- traderRef$Import( "DifferentQoS", "/", "" )

! {} <- diff$Op1() [ "Name == 'Video'" ]
! {} <- diff$Op2() [ "Name == 'Audio'" ]

! diff$DiscardOp( "Op2" )

! {} <- diff$Op1() [ "Name == 'Video'" ]
! {} <- diff$Op2() [ "Name == 'Audio'" ]

! diff$Discard
```

Figure 7.9: PREPC QoS Requests

vary their behaviour in response to exceptional conditions and error.

Dynamic and end-to-end QoS negotiation take place the *first* time an operation is called, and the resulting binding is re-used by all subsequent invocations. For interface wide QoS, the first invocation of any operation in that interface will result in this negotiation taking place, for operation specific QoS it will take place the first time that particular operation is called. The **Discard** statement can be used to discard the binding for *all* the operations in an interface, whilst the **DiscardOp** statement only discards the binding for the specified operation. In both cases an invocation to **UnBind** will be issued to free all QoS resources maintained at the server.

Figure 7.9 contains example invocations of the operations defined in figure 7.8.

An attempt to invoke **Diff1** or **Diff2** with a QoS other than that specified in its IDL definition will fail. Similarly, attempts to invoke **Same1** or **Same2**, with any QoS other than **Video** or **CheapVideo** will also fail.

The Testbench imposes the convention that the first argument of all server operations, is reserved for passing system information to that operation. The QoS request and offers currently in use are passed to the invoked operation as part of this first argument. PREPC provides the **OfferInUse** statement to allow clients to determine the QoS offer currently in use for a given operation. In both cases, only the QoS offers in use at the uppermost QoS layer are available. The syntax for this statement is as follows:

```
{ offer } <- interfaceRef$OfferInUse( operationName )
```

7.3.5 QoS Summary

This section has surveyed the QoS facilities provided by the prototype implementation of IMAC. The Trader constraint language has been adopted as the basis for representing QoS offers and constraints, and is used to implement per-layer QoS negotiation. A macro facility has been added to allow QoS offers to be mapped from one QoS layer to another and thus shield applications from the implementation details of their QoS specifications and requests. The constraint language is implemented within the QoS Manager, as opposed to within every capsule, and a cache is used to reduce the number of interactions required with the QoS Manager.

A uniform interface is provided for managing resources, and is used for both execution protocols and MPS modules. The provision of separate *reserve* and *confirm* resource operations allows for efficient end-to-end resource management. This interface is used for both dynamic QoS negotiation within the client and to implement end-to-end resource management for both client and server.

The use of UNIX as the underlying operating system has severely restricted the scope of the resource management provided by IMAC 3.0. Future ports to more flexible operating systems will enable the implementation of a wider range of resource management policies within the framework currently provided.

A simple interface is provided for specifying QoS specifications in interfaces, and QoS requests for operation invocations. QoS may be applied to individual operations within interfaces or to entire interfaces. QoS requests may be varied at run time, thus allowing clients to respond to exceptional conditions and errors.

Figure 7.10 illustrates the complete series of steps involved in QoS negotiation, from when an `InterfaceRef` is first created to when `UnBind` is invoked.

7.4 IDL Streams and Devices

The IDL implementation of IMAC streams provides separate statements for defining stream types and instances of those types, called stream ends. A stream type consists of a name, a list of event synchronisation interfaces, a list of stream synchronisation interfaces and a QoS specification. A stream end is a named instance of a named stream type whose direction is qualified as being either a plug, a multi-plug, a socket or a multi-socket. Multi-plugs and multi-sockets support multicast and multidrop communication. The IDL syntax, followed by some examples, is given below:

```
StreamType: STREAMTYPE =
  { list of event synchronisation interfaces }
  { list of stream synchronisation interfaces }
  [ QoS specification ];

StreamEnd: STREAMEND StreamType
  [ PLUG | SOCKET | MULTIPLUG | MULTISOCKET ];

Examples:

AudioStream: STREAMTYPE = { AudioEvents } { AudioLSFs }
  [ "Encoding ALaw" | "Encoding MuLaw" ];

MicroPhone: STREAMEND AudioStream PLUG;
Speaker: STREAMEND AudioStream SOCKET;
```

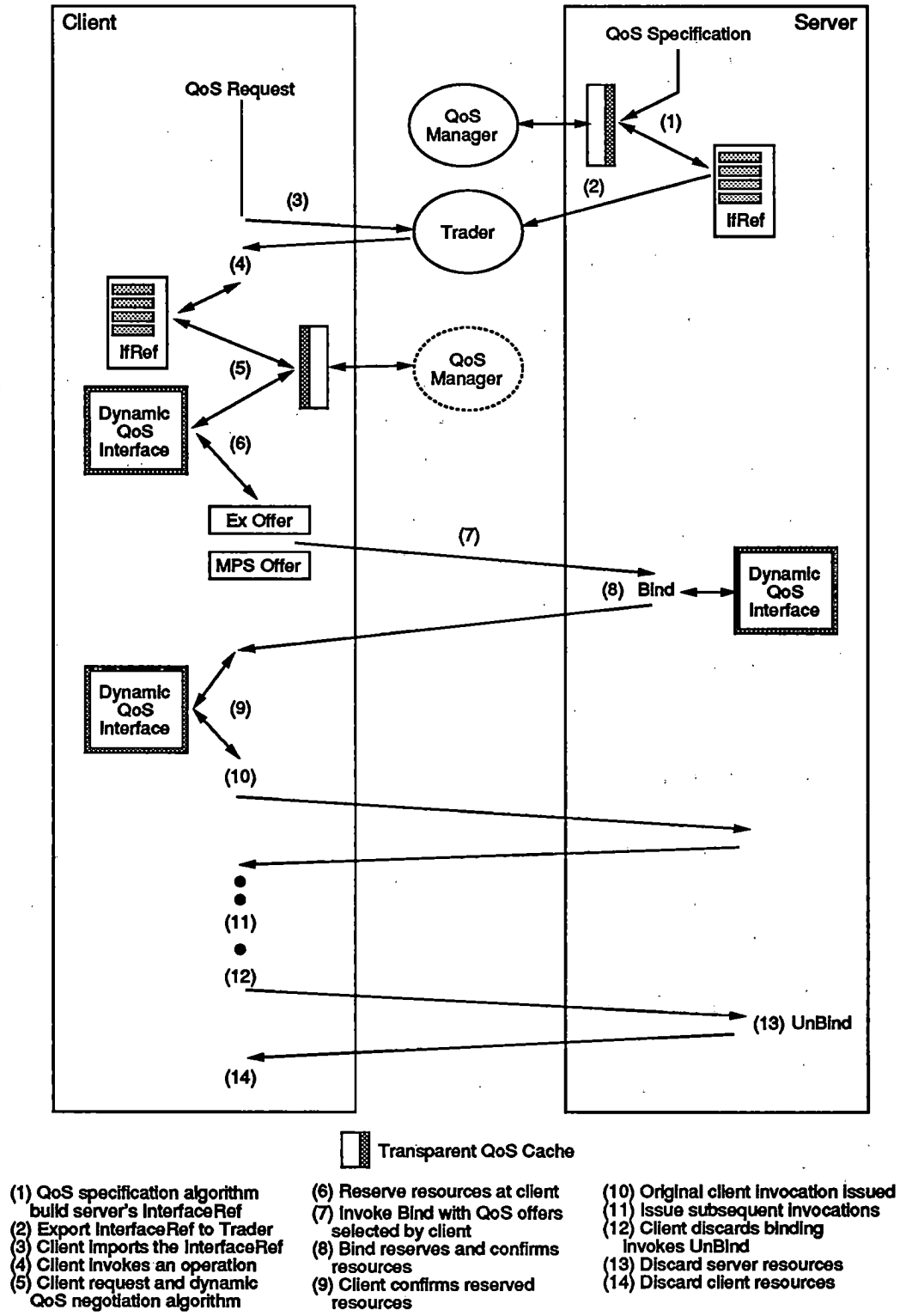

Figure 7.10: Stages In End-to-End QoS Negotiation

In order to create a path over which data may flow, two, or more, stream ends must be connected to form an end-to-end stream; often just called a stream. In order for such a connection to be made the following conditions must be met:

1. the source and sink streams must all be of the same stream type; in IMAC 3.0 this means that the stream types must have the *same name.*
2. the directions and connectivity of the stream ends must be *compatible.* Only plugs and multi-plugs may be used as sources, and sockets and multi-sockets as sinks, and the number of sources must *match* the number of sinks. For instance a single plug can only be connected to a single socket, but two plugs cannot be connected to a single socket.

These two conditions are not to be confused with device conformance, which applies to interfaces and not to stream ends; there is no corresponding notion of stream type conformance.

IMAC devices are distinguished from ordinary interfaces by the use of the keyword `DEVICE` in place of `INTERFACE`. The only effect of this keyword is to force the device to conform to the orchestration interface, `IMACOrchestrate`. This interface contains a set of management operations which must be implemented by all devices and is described in section 7.10.

In the current implementation there is no support for automatically synchronising operation invocations with LSF boundaries; it is left to the device implementation to provide such synchronisation itself. Section 9.4.2 presents a design for such support.

STUBC generates the following additional code specifically for streams and devices:

- stream type and stream end definitions are written to the signature file and thus made available to subsequent invocations of STUBC and PREPC.
- a UNIX Bourne shell script for updating the QoS Manager's database with the QoS information for the newly defined stream types. This script creates a single QoS domain for each stream type defined, and registers all of the QoS offers supported by each stream type with this domain. The name of the QoS layer is that of the stream type. A stream type contains QoS offers, as opposed to constraints, becauses it is viewed as service provider and not a service user. Once this script has been run, per-layer QoS negotiation can be used to match the QoS constraints specified at stream creation time to these offers.
- a C file containing a statically initialised set of data structures representing the stream types and stream ends defined in this interface. Library procedures are provided for accessing these data structures. In this way, all of the stream related information available at compile time is also available at run-time. These data structures are referred to as the *stream and device configuration.*

Chapter 8 presents a complete example illustrating the use of IDL streams and devices.

7.5 PREPC Streams and Devices

PREPC divides the process of creating an end-to-end IDL stream into three stages:

Stream Declaration: provided to support type checking and to allow the use of multiple instances of the same stream ends within a single application. The `STREAM` statement is provided for this purpose and specifies a named *template* for the stream to be subsequently created, which identifies the source and sink devices and stream ends. The syntax for this statement is as follows:

```
STREAM StreamName { list of plugs } -> { list of sockets }
```

The stream name is used to refer to this declaration from other PREPC statements. Stream plugs and sockets are identified by the name of the device interface and the name of the stream end within that device, separated by a full stop. For instance a stream plug `VideoSource` in an interface `Camera` would appear as `Camera.VideoSource`.

Obtaining Device Interfaces: in order to realise a stream, it is first necessary to obtain, or create, instances of the source and sink device interfaces which implement the required stream ends. An interface reference must be obtained for each device involved, and may be obtained by any means available to the application, including the Trader, User Locator or Desktop Manager.

Stream Realisation: once interface references have been obtained for all of the devices implicated by the stream declaration, it is possible to invoke the management operations provided by these devices to realise the stream in question. PREPC provides the `REALISE` statement which takes these interface references as arguments and invokes the required management operations on behalf of the application. A *stream reference* is returned which may subsequently be used to perform further management operations on the realised stream. A QoS request may be specified when a stream is realised in the same way as for operation invocations. The `REALISE` statement has the following format:

```
REALISE { stream reference } <- StreamName
  { interface references for devices containing plugs } ->
  { interface references for devices containing sockets }
  [ QoS request ]
```

In order to be able to type check the use of stream references the `DECLARE` statement has been extended to support their declaration as follows:

```
DECLARE { stream reference } : StreamType STREAMREF
```

`StreamType` must be a stream type defined in an IDL interface.

A stream declaration will fail if either of the conditions given in the previous section for connecting stream ends are not met. However, if it is only the *stream types* of the plug and socket which do not match then it may be possible to use a translator to reconcile this mismatch. The facility provided by the current implementation for managing stream translation is described in section 7.10.2.1.

The number and type of the interface reference arguments to a `REALISE` statement are checked to ensure consistency with the original declaration; the interface references must refer to the devices given in the original stream declaration. Once a stream has been realised, its behaviour may be controlled by directly invoking the operations in its device interfaces. PREPC also provides the following statements for setting and clearing event and stream synchronisation interfaces:

```
{} <- device$SetEventSync( StreamEnd, SyncInterface, InterfaceRef )
{} <- device$ClearEventSync( StreamEnd, SyncInterface )
{} <- device$SetStreamSync( StreamEnd, SyncInterface, InterfaceRef )
{} <- device$ClearStreamSync( StreamEnd, SyncInterface )
```

In all cases, **device** is an interface reference for the device generating the event or stream synchronisation points. The **StreamEnd** argument is required to distinguish between the possibly multiple stream end points supported by a single device, and must be set to the name of the stream end point in the device interface. The **SyncInterface** is the name of the event or stream synchronisation interface that is to be set or cleared. **InterfaceRef** is the corresponding interface reference which will receive synchronisation operation invocations. In this way, the controlling application may conveniently specify the synchronisation interfaces that it wishes to be invoked on. The stream implementation makes use of the stream configuration generated by STUBC to check the correctness of these arguments at run-time.

PREPC provides a **DestroyStream** pseudo-operation which may be invoked on a stream reference, to shut down an existing stream. The **OfferInUse** statement used by clients to determine the QoS in use for a specified operation may also be invoked on a stream reference to determine the QoS in use for a specified stream end. It can only be used after a successful **REALISE** statement and has the following syntax:

```
{ offer } <- streamRef$OfferInUse( Device.StreamEnd )
```

The current implementation does not provide any means of automating the implementation of the event and stream synchronisation interfaces; section 9.4.2 presents a design for automating this process.

Section 7.10 describes the management operations implemented by all devices and the additional support provided to aid in the implementation of devices themselves.

Chapter 8 presents a complete example illustrating the use of PREPC streams and devices.

7.6 QoS Manager

The prototype implementation of the QoS Manager (section 5.3.1) provides a system wide data base for QoS information, and an implementation of the constraint language and macro facility used for per-layer QoS negotiation.

QoS domains are represented as a linked list of QoS layers, with the name of the head layer in this linked list being used as the name of the domain. This scheme avoids the need for a separate field to identify QoS domains both within the QoS Cache and within interface references. QoS offers and macro definitions are posted to a single layer, and QoS constraints evaluated within the scope of a single layer. The QoS Manager can be invoked from *within* the Testbench communications system and therefore resides at a well known network address.

7.7 User Locator

The current implementation of the User Locator makes use of the Olivetti Active Badge system[13] to determine the physical location of users. Every user is equipped with an *active* badge, which transmits an infra-red signal, every fifteen seconds or so, uniquely identifying each badge. A network of receiver stations is used to detect these signals and the user's locations is taken as that of the receiver which most recently detected a transmission from that user's badge. The interface

[13]Donated by Olivetti Research Limited to the Computer Laboratory.

to this system is provided by two ANSA interfaces:[14] one providing location information and the other a database of user names, badge identifiers and station locations.

The User Locator uses a private instance of the database interface in order to associate its own information for describing station locations; currently the name of the nearest workstation is used. In this way, it is possible to identify the workstation that the user is currently closest to. The workstation name is then used as part of a Trader constraint expression to locate an instance of the service required at the workstation in question. The User Locator is thus able to provide an extended version of the Trader's Lookup operation. This operation takes an extra argument identifying the required user and only returns an instance of the service requested running on the workstation that the specified user is currently nearest to. If this service is unavailable at the workstation in question then an error is reported.

The level of indirection provided by the User Locator shields the application programmer from the details of the Active Badge system, and allows this system to be updated or even replaced, without affecting any application code. The User Locator is also able to make use of the IMAC 3.0 extensions to the Trader to implement its own policies for selecting between multiple offers. Alternatively it could provide some form of access control or customisation, whereby the user being located may specify particular times or locations at which he or she may not be contacted. In either case, these policies are implemented behind the standard User Locator interface and are transparent to the application programmer.

7.8 Desktop Manager

The Node Manager supplied with Testbench version 3.0, has been extended to provide some of the additional functions required by the Desktop Manager. The Node Manager, described in detail in A.12, was designed and implemented by the author, with the explicit goal that it could be used as the basis for a subsequent Desktop Manager implementation. As a result, only the following changes were required:

- the Node Manager `Run` operation, which is used to create new service instances, has been extended to return the `InterfaceRef` and `ActivationId` of the newly created service. Returning the `InterfaceRef` avoids the need for a subsequent interaction with the trader. The `ActivationId` can be subsequently used to destroy the instance created.
- activation groups, as defined in section 5.3.3, have been implemented, and are described below.

No support is currently provided for the creation and destruction of services in multiple run-time environments; primarily because UNIX has been the only run-time system used for the prototype implementation. Providing such a facility is straight forward; the only change required being an extension of the service description to include a *factory* service which is to be used to create and destroy instances of the service in question. A separate factory service must then be provided for each run-time system used.

The Node Manager allows the maximum number of instances supported for an individual service to be specified, and once this limit has been reached all further requests to create a new instance are refused until an existing instance has been destroyed. The Desktop Manager extends this scheme to support the management of resources used by multiple services. Such shared resources are represented as *activation groups*, to which any service using that resource must belong. Resource

[14] Jointly implemented by Joe Dixon and Andy Harter.

usage is measured in terms of the number of service instances, or activations, that a resource can support. Each group has an upper limit on the number of activations that it can support, and the Desktop Manager will refuse to create any further activations once this limit has been reached. A given service may belong to multiple activation groups, and the maximum number of instances that may be created for that service is given by the minimum of:

- the activation limit for the service itself.
- the activation limit for each of the activation groups that the service belongs to.

In addition, activation groups allow the current number of activations to be externally controlled. In this way, a service may temporarily relinquish, or attempt to acquire, the use of a shared resource by respectively decrementing, or incrementing, the activation number for the activation group in question.

7.9 Translation Manager

The need for a separate service to implement the Translation Manager has been avoided by making use of the Trader extensions described in section 7.2.2. This is achieved by adopting the convention that all translator devices include the following common set of properties when exporting their interfaces to the Trader:

`"StreamSocket StreamType"`: states the type of the supported stream socket.

`"StreamPlug StreamType"`: states the type of the supported stream plug.

In addition, all devices which can be used as translators must conform to the interface type `Translator`. In the current implementation `Translator` is a null interface, and is used solely as a means of grouping translator devices. In this way, *all* translator exports can be obtained using a Trader `Search` operation, for the type `Translator`. Once the list of all translators has been obtained their property lists may be searched to identify exports which provide a socket and plug of the required type.

7.10 Orchestration

The orchestration functions provided by IMAC 3.0 fall into the following categories:

Stream Management: primarily concerned with stream connection management, and with setting and clearing synchronisation interfaces.

Service Interfacing: provides an interface to the IMAC services.

Translator Management: functions for managing the use of Translator devices.

Stream and Device Implementation: provides support for implementing streams and devices.

7.10.1 Stream Management

As mentioned in section 7.10.3, all devices conform to a common interface, called **IMACOrchestrate**. This interface has been designed to support the use of a wide range of underlying transport protocols, and presents a general interface to connection management.

The approach taken is to provide separate operations for creating an address to be used for subsequent communication, for listening for connections on a supplied address and for connecting to a specified address. Addresses are represented as `InterfaceRefs` and can therefore support the use of the Testbench communication system, in addition to any other underlying transport protocol. If the Testbench communication system is *not* used then only the MPS portion of the **AddressRecord** need be used.

The operation to create an address takes a QoS request as an argument, and then performs any per-layer QoS negotiation required; section 7.10.3 describes the support provided for implementing such negotiation. The connect and listen operations take a *sequence* of addresses to connect to, or listen from, and can thus be used for multicast and multidrop communication. In order to create a stream connection the **REALISE** statement generates code to implement the following algorithm:

1. invoke the create address operation for each stream source.
2. invoke the create address operation for each stream sink.
3. invoke the listen operation with the sequence of addresses resulting from steps 1 and 2.
4. invoke the connect operation with the sequence of addresses from steps 1 and 2.

The results of all of these invocations are stored in the stream reference returned as the result of the **REALISE** statement.

Given that *any* of these invocations may fail, some means of determining which one failed is required. This is achieved by generating code to check the results of each invocation and to call a user supplied operation if an error is detected. A different procedure is required for each connection management operation, thus implicitly identifying the source of the error.

An operation is provided to destroy an address, and is invoked for all stream sources and sinks by the **DestroyStream** operation.

The implementations of these management operations must interface with the underlying transport protocol actually used to communicate the stream data. The degree of processing required will depend on how closely the underlying protocol matches the connection management interface.

Finally, operations are provided for setting and clearing event and stream synchronisation interfaces and are invoked by the corresponding PREPC statements.

7.10.2 Service Interfaces

There are only two statements provided for interfacing to IMAC services: one for the User Locator and one for the Translation Manager.

The **UImport** statement is an extension of the Trader **Import** operation which takes an additional argument identifying a user and then invokes the User Locator as opposed to the Trader, to import the required interface for the specified user. The following example illustrates the use of this statement to locate an instance of the **Echo** service for the user **Cosmos.Nicolaou**.

```
! DECLARE translator : Translator CLIENT

! STREAM "FirstHop" { Source.X } -> { Translator.X }
! STREAM "SecondHop" { Translator.Y } -> { Sink.Y }

! DECLARE { source } : Source CLIENT
! DECLARE { sink } : Sink CLIENT
! DECLARE { firstHop } : FirstHop STREAMREF
! DECLARE { secondHop } : SecondHop STREAMREF

! TRANSLATE { translator } <- { Source.X } -> { Sink.Y }

! REALISE { firstHop } <- "FirstHop" { source } -> { translator }

! REALISE { secondHop } <- "SecondHop" { translator } -> { sink }
```

Figure 7.11: Stream Translation Example

```
! { userLocator } <- traderRef$Import( "UserLocator", "/" "" )
! { ir } <- userLocator$UImport( "Cosmos.Nicolaou", "Echo", "/" \
  "Name == 'Echo'" )
```

The **Translate** statement is provided for obtaining a suitable translator device given a pair of stream end points to translate between; it has the following form:

```
! TRANSLATE { translator } <- { source end point } -> { sink end point }
```

The result is an **InterfaceRef** for a suitable translator, or if no such translation could be found an error. The stream end points are identified in the same way as in a stream declaration (i.e. **Device.StreamEnd**). This statement generates an invocation of the Trader **Search** operation and scans the result for a suitable translator as outlined in section 5.3.4. This statement can only be used for one-to-one streams and not for multicast or multidrop streams.

7.10.2.1 Translation Management

The prototype implementation does not provide any support for *automatically* inserting the translator devices in the event of a stream type mismatch. Therefore, the application writer must explicitly manage the use of translator devices; the required process is best illustrated by example. In order to connect a stream plug of type X, to a socket of type Y, a translator that converts from X to Y is required. Two streams must be declared, one from the device sourcing X to the translator, and one from the translator to the device sinking Y. The **TRANSLATE** statement, described in section 7.10.2, can then be used to obtain a suitable translator; the **REALISE** statement can be used to realise *both* streams. Figure 7.11 illustrates this example.

Unfortunately, the stream declarations will fail since the **Translator** device type does not implement any stream ends. Therefore, the type checking of the stream ends must be deferred until a suitable translator has been found. Currently, the implementation of **STREAM** detects the use of the interface **Translator** and defers type checking the appropriate stream-end. The **TRANSLATE** statement makes a note of the type of the translator device found and makes this information available to a subsequent **REALISE** statement, which is then able to implement the required checking.

This approach of providing the minimal amount of support required for the application writer

to explicitly manage translation has a number of advantages: it is simple to implement, does not preclude the provision of automatic translation management in the future, and may even be used as the basis for implementing such automatic management. In any case, even if automatic translation management were to be provided then the ability to explicitly control translation would have to be retained for the few applications whose requirements could not be met by the automatic scheme.

7.10.3 Stream and Device Implementation

PREPC provides the following statements which are intended to be of use for implementing the management operations required by IMAC devices.

`{} :: StreamType$InitStreamQoS()`
: reads the stream configuration for the stream type, `StreamType`, and initialises the QoS cache in readiness for per-layer QoS negotiation.

`{ r, offers } :: StreamEnd$Match( qosRequest )`
: performs per-layer QoS negotiation and returns the set of QoS offers which can satisfy the request. It is then left to the device to determine which of these offers to use.

`{ r, streamEnd } :: Device$FindStream( streamEndName )`
: searches the configuration for the specified device interface for the specified stream end. The result contains the data structure representing the stream end. This can be used to validate requests to set and clear synchronisation interfaces.

`{ r, stream } :: Device$FindStreamType( streamTypeName )`
: identical to `FindStream`, except for stream types as opposed to stream ends.

The result `r` contains a status code describing the result of the operation.

7.10.4 Orchestration Summary

When deciding what orchestration functions to provide there is a compromise to be made between providing just enough functionality to be of genuine use to the application writer on the one hand, and providing too much functionality on the other. Providing too much is likely to lead to a complex interface which is hard to programme, hard to maintain and liable to frequent change. Therefore, IMAC 3.0 provides a minimum number of widely used, and flexible orchestration functions; rarely used or complex functions are not provided. In particular, the decision not to implement the automatic insertion of translator devices was justified on these grounds; that is, it may only be rarely used and its implementation is likely to be complex.

7.11 Summary

The prototype described in this chapter represents an almost complete implementation of the IMAC architecture. It has identified and remedied a number of important deficiencies present in the ANSA Testbench on which it was based; in particular the communication system has been re-implemented to fully support QoS. A new communication abstraction, called a Local Channel Resource, has been designed and implemented. Local channel resources provide a compromise between the performance advantages offered by the use QoS and the scaling properties provided

by multiplexing. In particular, communication not requiring QoS does not pay a scaling or performance penalty for its implementation. The performance of QoS based communication has been improved by the ability to use separate communication channels and the reduction in jitter and latency achieved through the use of constant-time look-up for network receptions.

A simple programming interface is provided for the use of QoS which hides the implementation details from the programmer. In addition, QoS has also been used to manage the configuration of the communication system and represents a powerful tool for implementing network management.

Extensive support is provided for IMAC streams and devices which has been fully integrated into the two programming languages supported by the prototype implementation. In this way, streams and devices have been integrated into the distributed computing environment provided by the Testbench.

The full set of IMAC services have been implemented and, where appropriate, orchestration functions have been provided for their access. Orchestration functions are provided not only for use by controlling applications but also for use by stream and device implementations.

There are three principal areas that have not been implemented, namely: multi-channel synchronisation, the synchronisation of device operations to LSF boundaries and automatically generated implementations for event and stream synchronisation interfaces. Although not implemented, designs have been prepared for each of these features and are presented in sections 9.4.1, 9.4.3 and 9.4.2.

8

A Complete Example

This chapter presents a complete example illustrating the use of the prototype IMAC implementation and demonstrates how the various components of this implementation can be used to build a real application. The example application manages and monitors the event and stream synchronisation of a uni-directional video stream. It makes use of the User Locator and Desktop Manager to create such a stream between named users. The description is kept as general as possible to give an overall understanding of how other applications could be constructed.

The example described has been completely implemented except for the transport of video data; uninitialised memory buffers are used in place of video data and are transmitted at the rate allowed by the underlying operating system, workstation and network.

8.1 Application Structure

The example application has two components: a control component which manages and monitors the video stream, and the implementation of the stream itself. The stream is composed of two devices, one for the stream source and one for the stream sink. The logical separation of these components does not imply a corresponding physical separation; for instance, all components could be co-located on the same workstation, within the same address space, or separated by a wide area network. The boundaries between these components are represented as IMAC devices, streams and synchronisation interfaces.

Figure 8.1 illustrates this structure and identifies the interfaces used; figure 8.2 shows the IDL definitions for these interfaces. The implementation of the management operations provided by *all* devices is discussed in section 8.3; these operations are invoked by the orchestration statements contained in the control component.

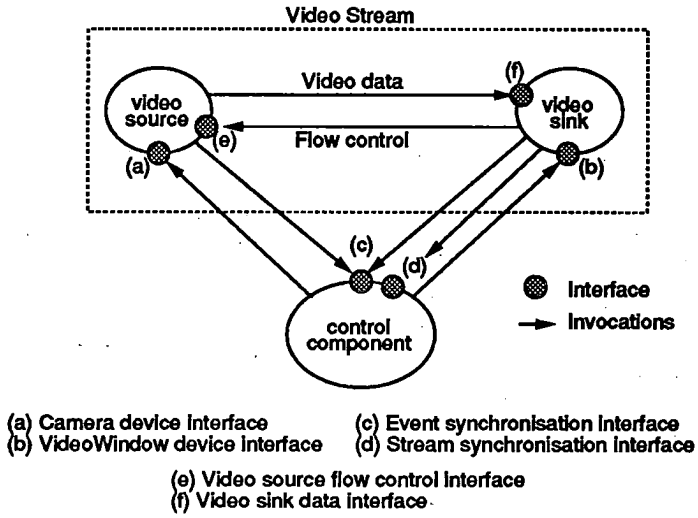

Figure 8.1: Example Application Structure

These interfaces are created and communicated as follows:

- the `Camera` and `VideoWindow` device interfaces are created when their corresponding devices are created. If the Desktop Manager is used to create these devices then the `InterfaceRefs` for the newly created interfaces will be returned directly to the control component.
- the video source and video sink interfaces are created and exchanged when the video stream is realised. The `REALISE` statement generates the invocations on the management operations required to achieve this.
- the control component creates instances of the event and stream synchronisation interfaces it wishes to receive invocations on, and registers them with the devices using the corresponding PREPC statements.

Figure 8.3 illustrates the sequence of PREPC statements required to create this video stream and associated interfaces. The brevity of this example demonstrates that application complexity has been effectively reduced by the provision of a powerful, language based interface.

8.2 Control Component Synchronisation

The control component has two sub-components:

- a central synchronisation loop responsible for implementing the required event and stream synchronisation algorithm.
- the implementations of the event and stream synchronisation operations themselves.

Some means is required for communicating the fact that synchronisation operations have been invoked, and therefore stream synchronisation points reached, to the synchronisation loop. Existing

```
VideoEventSync : INTERFACE =
BEGIN
  VideoStarted: OPERATION [] RETURNS [];
  VideoStopped: OPERATION [] RETURNS [];

  ExceptionCode: TYPE = { communicationError };
  ErrorDetected: OPERATION [ code: ExceptionCode ] RETURNS [];
END

VideoStreamSync : INTERFACE =
BEGIN
  TimeStamp: TYPE = CARDINAL;
  SequenceNumber: TYPE = CARDINAL;

  FrameReceived: OPERATION [
    st_LSF_seq: SequenceNumber; st_LSF_time: TimeStamp
  ] RETURNS [
    sync_error: BOOLEAN;
    app_LSF_seq: SequenceNumber; app_LSF_time: TimeStamp
  ];
END

VideoStream : INTERFACE =
NEEDS VideoEventSync;
NEEDS VideoStreamSync;
BEGIN
  VideoStream: STREAMTYPE = { VideoEventSync } { VideoStreamSync } [
    "X 128 Y 128" | "X 256 Y 256" | "X 512 Y 512" ];
END.

Camera : DEVICE =
NEEDS VideoStream;
BEGIN
    VideoSource: STREAMEND VideoStream PLUG;

    Play: OPERATION [] RETURNS [];
    Stop: OPERATION [] RETURNS [];
END.

VideoWindow : DEVICE =
NEEDS VideoStream;
BEGIN
    VideoSink: STREAMEND VideoStream SOCKET;

    Create: OPERATION [ X, Y: CARDINAL ] RETURNS [];
    Destroy: OPERATION [] RETURNS [];
END.
```

Figure 8.2: Stream and Device Interfaces for a Video Stream

```
! DECLARE { cam } : Camera CLIENT
! DECLARE { vwin } : VideoWindow CLIENT
! DECLARE { eventSync } : VideoEventSync SERVER
! DECLARE { streamSync } : VideoStreamSync SERVER

! STREAM "VPipe" { Camera.VideoSource } -> { VideoWindow.VideoSink }

! DECLARE { vpipe } : VPipe STREAMREF

-- locate the desktops for the users
! { userLoc } <- traderRef$Import( "UserLocator", "/" "" )
! { from_dtop } <- userLoc$UImport( from_user, "DesktopManager", "/", "" )
! { to_dtop } <- userLoc$UImport( to_user, "DesktopManager", "/", "" )

-- create the devices
! { cam, cam_id } <- from_dtop$Run( "Camera" )
! { vwin, vwin_id } <- from_dtop$Run( "VideoWindow" )

-- realise the video stream
! REALISE { vpipe } <- "VPipe" { cam } -> { vwin } \
    [ "X == 128 and Y == 128" ] Signal *

-- create and set event and stream synchronisation interfaces
! { eventSync } :: VideoEventSync$Create()
! { streamSync } :: VideoStreamSync$Create()

! {} <- cam$SetEventSyncPoint( "VideoSink", "VideoEventSync", \
    eventSync )
! {} <- vwin$SetEventSyncPoint( "VideoSink", "VideoEventSync", \
    eventSync )
! {} <- vwin$SetStreamSyncPoint( "VideoSink", "VideoStreamSync", \
    streamSync )

--
-- control and monitor the synchronisation
-- of the video stream
--

-- clear event and stream synchronisation interfaces
! {} <- vwin$ClearEventSyncPoint( "VideoSink", "VideoEventSync" )
! {} <- vwin$ClearStreamSyncPoint( "VideoSink", "VideoStreamSync"  )

-- destroy the video stream
! {} <- vpipe$DestroyStream{}

-- destroy the devices
! {} <- from_dtop$Kill( "Camera", cam_id )
! {} <- from_dtop$Kill( "Camera", vwin_id )
```

Figure 8.3: Application for Controlling a Video Stream

synchronisation primitives such as semaphores or eventcounts and sequencers [**Reed79**] can be used. The only complexity introduced is the requirement to wait for the occurrence of *multiple* synchronisation points simultaneously. Section 9.4.2 describes a design for the automatic generation of code and synchronisation primitives required to implement this. The examples in sections 6.1 and 6.2 have already illustrated how the control loop could be implemented.

8.3 Device Implementation

The device implementation must perform the following functions:

- implement the device interface, including management operations.
- synchronise the invocations of device operations to stream LSF's.
- invoke the appropriate stream synchronisation operations when synchronisation points are reached.
- implement the transport of stream data.

The PREPC statements described in section 7.10.3 simplify the implementation of the management operations; in particular the `Stream$Match` statement provides easy access to per-layer QoS negotiation. However, synchronisation of operation invocation to LSF's must be implemented directly, and the scheme used will be highly dependent on the transport protocol used to communicate stream data.

Data transport may, or may not, be implemented using the IMAC communications system; in either case the control component is unaware of the mechanism used. If IMAC communication is used then it is likely that use will be made of the QoS facilities it provides. The QoS used for data transport should not be confused with that used for the stream, as specified in the stream interface and when the stream is realised. A stream QoS request specified in a `REALISE` statement is passed to the management operations implemented by the device. These may then use the `Stream$Match( qosRequest )` statement to match the request against the offers specified in the stream's IDL definition. The device implementation may then map the returned QoS offers into a new, and different, QoS request for use when transporting the stream's data. The mapping used is entirely implementation dependent. For instance a stream offering a QoS of `"X 128 Y 128"` might be mapped to QoS request `"VideoProtocol 128,128,FullRate"`.

8.4 Summary

This chapter has outlined a complete example, and used it to illustrate how the various components of the prototype IMAC implementation can be used to build such an application. The description has covered all of the major components of the application and has been kept as general as possible to impart an understanding of how other applications could be constructed.

9

Evaluation and Extensions

This chapter presents an evaluation of IMAC and its prototype implementation, designs for unimplemented portions of IMAC and suggestions for future research. The evaluation has the following components:

- a quantitative evaluation of the performance of the prototype implementation.
- a qualitative evaluation of how well the requirements made by multimedia systems have been addressed by the IMAC architecture and its implementation, and a discussion of the benefits realised as a result of satisfying these requirements.
- a set of requirements for future systems to enable them to better support IMAC implementations.

9.1 Performance Evaluation

In order to determine the cost of implementing IMAC the performance of its prototype implementations is compared against that of the original version of the Testbench on which it is based.

All experiments were run between lightly loaded Hewlett Packard Series 9000/375[1] workstations running HP/UX[2] connected by the Computer Laboratory Ethernet. These measurements were taken either in the evening or over the weekend when the network load is lightest. The results presented were obtained by averaging the results of multiple experiments, where each experiment

[1] Rated at approximately 11 VAX MIPS.

[2] Running the full complement of daemons.

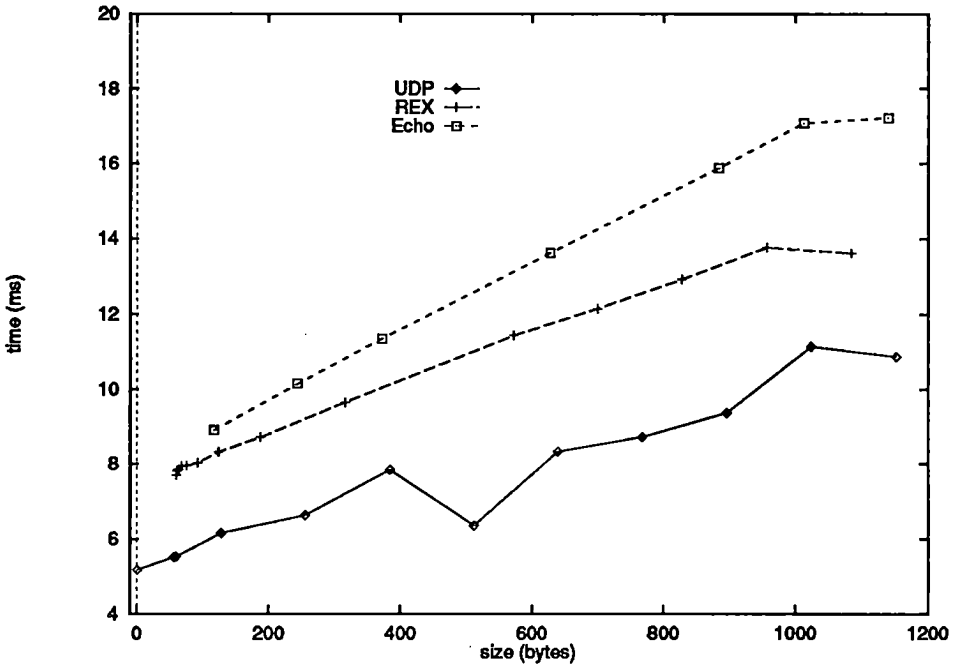

Figure 9.1: Version 3.0 RPC Performance

consisted of multiple runs of 1000 or 10,000 invocations. In addition, where the results of comparative experiments are presented, care was taken to ensure that these experiments were carried out under as near identical conditions as possible.

9.1.1 Version 3.0 RPC Performance

The time taken to execute a single RPC has three major components:

system time: the time spent in the underlying operating and communication system for transmitting and receiving network packets.

protocol time: the time spent in the RPC protocol itself.

stub time: the time spent in stub code to invoke and dispatch remote operations, and to marshall and unmarshall arguments and results.

Figure 9.1 shows the performance of version 3.0. All measurements shown are for an *echo* operation which sends *and* receives n bytes of data. Plots are given for raw UDP performance,[3] for the raw RPC protocol, REX, and for an application level function, called *Echo*, which returns a single string argument as its result. The size plotted along the x axis is the *total* size of the network buffers transmitted and received, and includes protocol and stub headers. The UDP timings represent the *system time*, the difference between the REX and UDP timings the *protocol time* and the difference between REX and Echo, the *stub time*.

[3]The improvement in UDP performance seen for 512 and 1024 byte and larger packets is almost certainly due to a larger number of network buffers of that size being available.

System	Time (ms)
Firefly RPC [Schroeder89]	2.66
Amoeba [van Renesse88]	1.40
Testbench	7.51

Table 9.1: Performance of Contemporary RPC Systems

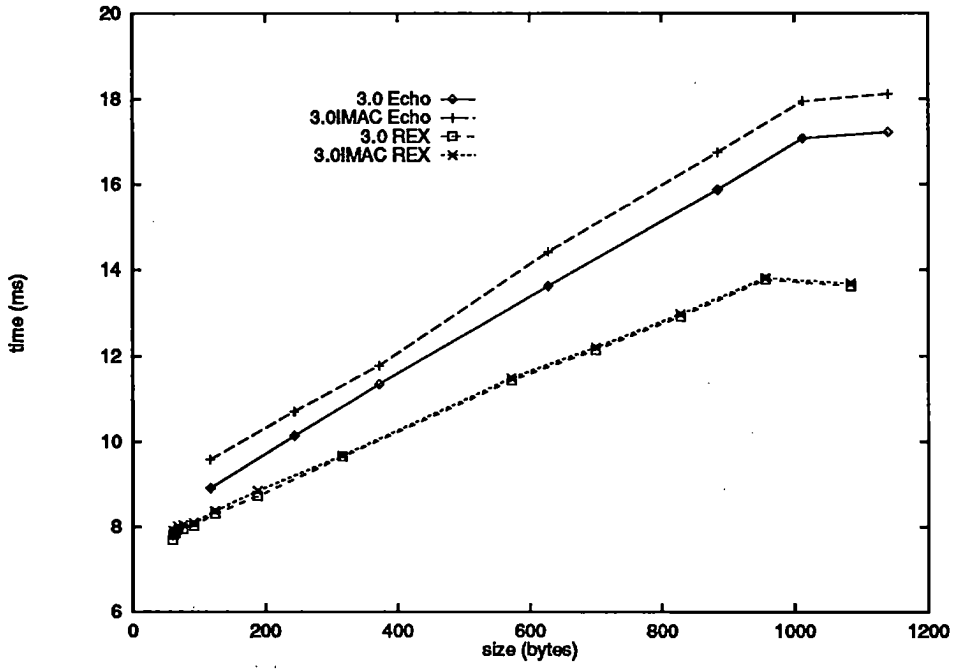

Figure 9.2: Version 3.0 and IMAC 3.0 Performance Comparison

This graph shows that the protocol time is greater than the stub time for small packet sizes but is gradually overtaken by the stub time as data size increases. However, the protocol time remains constant with respect to buffer size and system time. The sum of the protocol and stub time is just greater than the system time, indicating that the system accounts for just under half of the total latency. The marshalling and unmarshalling of the simple strings used by the **Echo** operation is dominated by memory copying and represents the minimal marshalling overhead. More complex arguments and results are likely to require even greater processing time.

Table 9.1 shows that the performance of the Testbench RPC system is substantially worse than that of other contemporary systems. However, it is the only one of these systems to run over UNIX as a user process, to have been ported to a wide variety of operating systems and not to have had any concerted effort to improve its performance. Even so, it still achieves sub $10ms$ performance for invocations whose argument and replies are less than 64 bytes in size.

9.1.1.1 IMAC 3.0 Performance

Figure 9.2 compares the performance of IMAC 3.0 to that of Testbench version 3.0. The timings for the raw REX protocol are very similar, but IMAC 3.0 appears to be worse, by a constant amount, for the **Echo** operation.

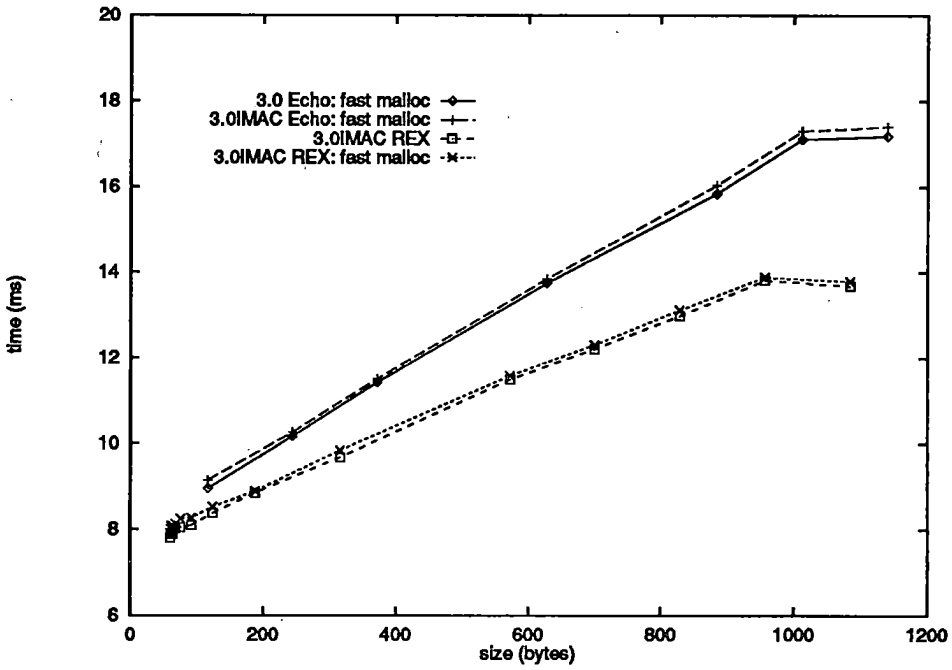

Figure 9.3: RPC Performance Using Fast Malloc

In order to investigate the source of this additional delay the client and server test programs were profiled using the UNIX `gprof` program. The results showed that the new implementation was spending a far greater amount time in the HP/UX `malloc` library procedure. Therefore, the experiments were repeated using a faster implementation of `malloc`, also provided by HP/UX; these results are presented in figure 9.3. The use of the faster `malloc` improved the performance of the `Echo` operation, but appears to have slowed the raw REX timings. Further investigation revealed that the source of the original additional delay was an obscure bug in the Testbench 3.0 buffer management code[4] which resulted in a failure to use pre-allocated buffers for each invocation and generated a call to `malloc` for *every* invocation.

Figure 9.4 shows the results of a repeated set of experiments with the buffer management bug fixed. The differences between 3.0 and IMAC 3.0 are now in the order of tens of microseconds and are almost certainly due to the different programming style used for IMAC 3.0. The use of global variables has been reduced, a greater number of arguments are passed on the stack and several of the global variables used in 3.0 have been transformed into structures and hence incur an extra copying overhead when assigned. In addition, IMAC 3.0 has been coded in a *defensive* fashion and includes a large number of run-time assertions and correctness checks. The performance advantages gained through the use of Local Channel Resources (LCR's) have been cancelled out by these other losses. Performance could be improved by streamlining the new code and removing many of the correctness checks.

Table 9.2 gives the actual timings for 3.0 and IMAC 3.0, and shows that IMAC 3.0 is between $0.05ms$ and $0.08ms$ slower for all but the 512 byte `Echo` operation, for which it is slower by $0.27ms$. It is not clear why IMAC 3.0 should perform slightly worse for medium sized packets, however, the most likely cause is due to a change in the ratio of time spent waiting for network operations to complete and the time spent marshalling and unmarshalling data. The difference is sufficiently

[4] This bug will be fixed in future releases of the Testbench.

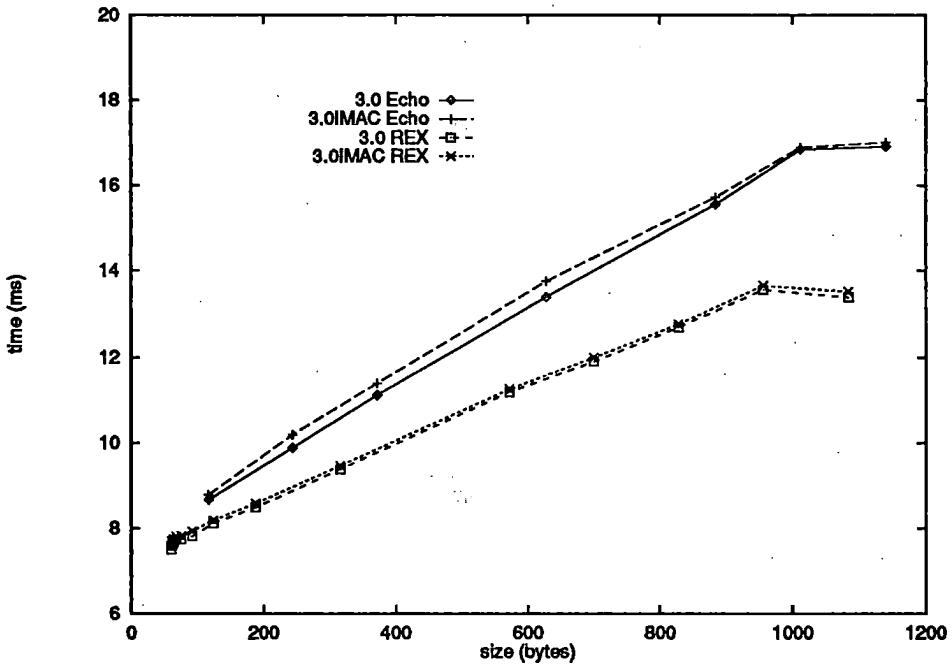

Figure 9.4: RPC Performance With Improved Buffer Management

Version	REX (ms)	Null RPC (ms)	Echo (ms)		
			1 byte	512 bytes	1024 bytes
3.0	7.51	8.48	8.67	13.40	16.90
IMAC 3.0	7.58	8.53	8.75	13.67	16.97

Table 9.2: Version 3.0 and IMAC 3.0 RPC Timings

small to not be a cause for major concern.

9.1.2 Extended Interface References

The other major difference between 3.0 and IMAC 3.0 is the extended format for interface references. Table 9.3 gives timings for a simple operation to return an `InterfaceRef` as a result and for the Trader `Lookup` operation. The `InterfaceRef` timings give an indication of the marshalling overhead for the new interface references, whilst the `Lookup` timings include the additional processing overhead incurred within the Trader. Timings are shown for a minimal sized, default interface reference[5] and for a larger one containing a substantial amount of QoS information; the interface reference used is that created for the `QoSTest` interface shown in figure 9.5. The times for the default interface reference represent the best that can be achieved using IMAC 3.0. The overhead for the default interface reference for both experiments is in the order of a few milliseconds, but this increases to approximately $15ms$ for the larger interface reference. This increased overhead is entirely due to the large size of the interface reference used, approximately 1200 bytes, which required the use of fragmentation to transmit it using several transport protocol packets. The size

[5] i.e. one that contains a default `AddressHint`.

System	Interface Reference		Trader Lookup	
	Default	With QoS	Default	With QoS
3.0 (ms)	9.28	N/A	12.14	N/A
IMAC 3.0 (ms)	11.43	25.80	15.61	27.00
Overhead	2.15	16.52	3.47	14.86

Table 9.3: IMAC 3.0 Interface Reference Overhead

```
QoSTest: INTERFACE =
BEGIN
  Op1: [ "Name == 'Stream'" | "Name == 'LocalStream'" ]
    OPERATION [ Num : CARDINAL; Str : STRING ]
    RETURNS [ CARDINAL; STRING ];

  Op2: [ "Name == 'DataGram'" ]
    OPERATION [ ] RETURNS [ STRING; CARDINAL ];

  Op3: [ "Name == 'Stream'" ]
    OPERATION [ Num : CARDINAL ] RETURNS [ ];

  Op4: OPERATION [ Num : CARDINAL ] RETURNS [ ];

  QEcho: [ "Name == 'Stream'" | "Name == 'LocalStream'" |
           "Name == 'DataGram'" ]
    OPERATION [ Src : STRING ] RETURNS [ STRING ];
END.
```

Figure 9.5: Interface Used for QoS Timings

of the interface reference can be reduced by either reducing the number of operations including QoS specifications, or by using an interface wide QoS specification. This increased overhead is not considered a serious problem because of the fact that interface references are communicated relatively infrequently. If a large number of interface references are as verbose as that for the `QoSTest` interface then the Trader may become a bottleneck; its best case performance has been reduced to 65 `Lookups` per second for the default interface reference and to 37 for the `QoSTest` interface reference.

Figure 9.6 shows the time taken to create an interface reference for the `QoSTest` interface shown in figure 9.5. The graph shows the time taken to create an increasing number of interface references; as this number is increased the overhead of communicating with the QoS Manager is amortised over the greater number of creations. Two plots are shown, one using the standard `malloc` and the other the fast `malloc`. The time to create a single instance is approximately $400ms$ using the standard `malloc` and $250ms$ using the fast `malloc`. The standard `malloc` fragments memory very quickly and leads to severe performance degradation as the number of interface references created increases; the fast `malloc` is much more stable and tends to a lower limit of approximately $22ms$ for interface reference creation. The time to create interface references bottoms out after creating approximately 60 interface references. The fast `malloc` was used for all subsequent experiments involving QoS.

As seen from figure 9.6 the QoS Cache is effective in reducing the number of invocations to the QoS Manager. For the `QoSTest` interface, a total of 10 QoS Manager invocations were required for any number of interface references. These can be accounted for as follows:

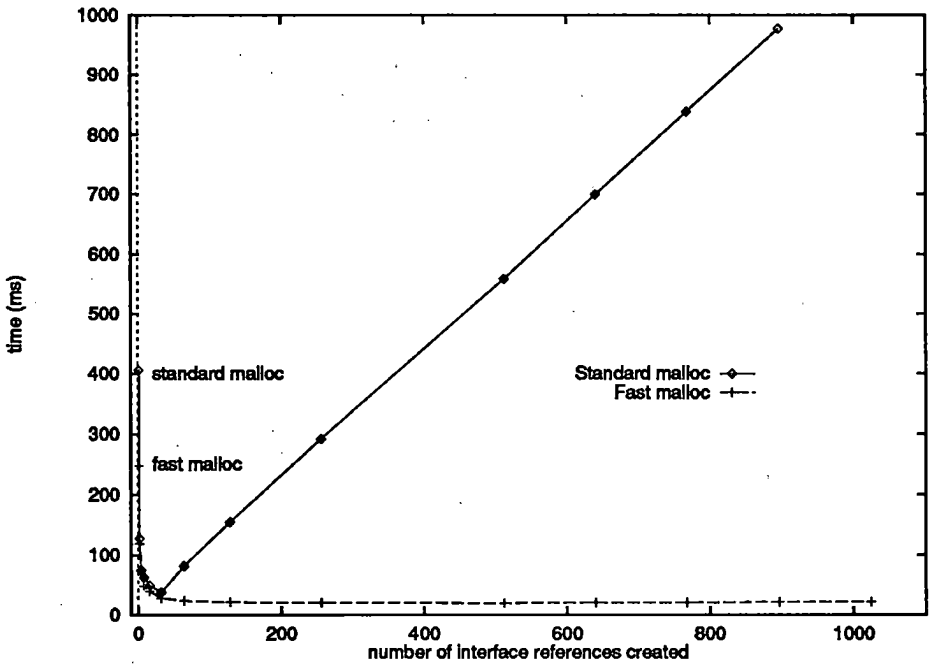

Figure 9.6: Performance of Interface Reference Creation

1 invocation	1000 invocations
189.20*ms*	74.76*ms*

Table 9.4: QoS Bind Times

- 2 invocations to identify a QoS layer: one each for the execution and MPS QoS layers.
- 8 requests to evaluate constraint expressions: one at each layer for each of the 3 *different* QoS specifications, plus an QoS empty request which is issued by the QoS cache to determine all of the available offers at each layer.

This number could be further reduced by extending the configuration files used to store the QoS offers supported by a capsule to include constraints known to match those offers; in other words, by moving more of the QoS negotiation algorithm to compile time.

Table 9.4 shows the time to create a binding for the `QEcho` operation of the `QoSTest` interface; this time includes:

- the execution of the client QoS negotiation algorithm.
- any invocations on the QoS Manager.
- the invocation of the server's `Bind` operation.
- the invocation of the operation itself.

Operation	Echo Size (bytes)						
	1	128	256	512	768	896	1024
Echo (ms)	8.88	10.13	11.26	13.61	15.82	17.05	17.18
Echo with QoS (ms)	8.89	10.13	11.23	13.59	15.82	17.06	17.16

Table 9.5: Echo Timings With and Without QoS

The QoS cache ensures that the QoS Manager will only be invoked the *first* time that a particular QoS binding is created, i.e. when the operation is first invoked. A total of 6 QoS Manager invocations are required as follows:

- 2 invocations to identify a QoS layer: one each for the execution and MPS QoS layers.
- 4 requests to evaluate constraint expressions: one at each layer for the QoS request used, plus an QoS empty request which is issued by the QoS cache to determine all of the available offers at each layer.

Therefore, the time for a single bind includes the QoS Manager overhead, whilst the time for 1000 bindings does not. The time to destroy a binding only involves a single call to the server's `UnBind` operation, regardless of however many times it is called, and was measured at $14.58ms$.

9.1.3 QoS Performance

Table 9.5 shows that the timings for the `Echo` and `QEcho` operations, with and without QoS are virtually identical. The QoS used simply specified the use of a datagram transport protocol (i.e. `"Name == 'DataGram'"`) and selected the use of UDP.

Figure 9.7 shows the performance of the `Echo` and `QEcho` operations when implemented by the *same* server, and invoked simultaneously by *one* client each. That is, a single server is receiving invocations from two clients, one of which is calling `Echo` and the other `QEcho`. `QEcho` is invoked with a datagram QoS request. This clearly shows that the prototype implementation is able to prioritise QoS based communication and to provide better performance than for non-QoS based communication. Although the `QEcho` operation performs better than `Echo`, it is still worse than the performance for a single client invoking operations on a single server. This is an inevitable result of the fact that the implementation of IMAC 3.0 over UNIX does not support any form of pre-emptive scheduling and hence even QoS based communication must wait for any other communication to be processed to completion.

9.1.4 Conclusion

The performance evaluation has shown that the performance of IMAC 3.0 is very similar to that of the original version of the Testbench on which it is based. In addition the implementation and use of QoS has no detrimental effect on RPC performance, with the cost of QoS being concentrated at interface creation and bind time. It has also showed that it is possible to prioritise QoS based communication and thus provide better performance than for non-QoS based communication in the case of multiple clients. The extended interface references make heavy use of `malloc` and as a result the performance of IMAC 3.0 is highly dependent on that of the `malloc` implementation used.

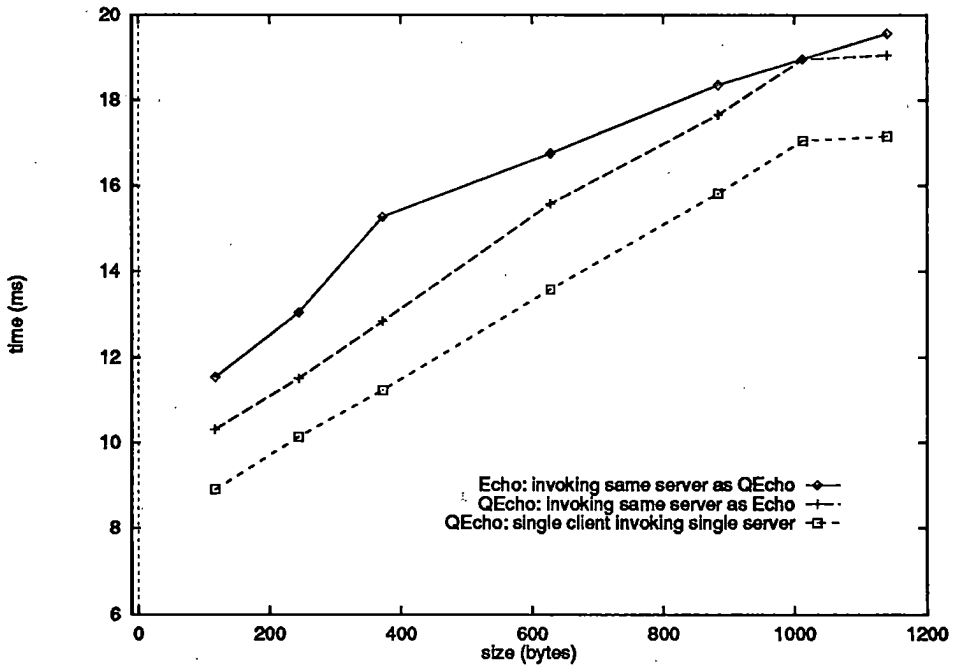

Figure 9.7: QoS Performance Using Two Clients

9.2 Evaluation

The evaluation presented in this section examines how well the requirements identified for multimedia in section 4.2 have been satisfied by IMAC and if the anticipated benefits have been realised. The principle features of IMAC are evaluated in turn.

9.2.1 IMAC Streams and Devices

IMAC streams were designed to support the synchronisation of multiple, related, media and IMAC devices to provide a uniform interface for managing device and protocol heterogeneity. The principal benefits sought were as follows:

- to support stream synchronisation *without* multiplexing.
- to support a wide variety of media and application specific synchronisation schemes.
- to allow the system to evolve as technology progresses and as system performance and configuration changes.
- to support the construction of portable applications.
- to support the construction of open systems as defined in section 4.2.10.1.

Logical Synchronisation Frames, defined in section 5.2.3.3 and illustrated in chapter 6, define a generic interface for synchronising multiple media streams. This interface allows streams to be synchronised whilst being kept separate, thus avoiding the use of multiplexing to synchronise streams

and allowing the advantages offered by the use of separate communication channels, (described in section 4.2.4.4) to be realised. Synchronisation is provided in a device, stream and media independent manner and stream interfaces represent candidates for standardisation. By keeping each media stream separate it is also possible to add new media, and remove existing media, from the set of media being synchronised. The two-level, LSF-PSF, structure allows the synchronisation granularity to be varied and applications to choose the level of synchronisation they require. In this way, it is possible to vary the degree of real-time processing implemented within the application and thus allow applications to evolve along with technology and changes in system configuration.

A potential weakness of this scheme is the cost of communicating the LSF structure to the controlling application; i.e. the cost of invoking synchronisation operations may be too great. However, as shown by section 9.1, even the un-optimised RPC implementation of IMAC 3.0, provides sub $10ms$ performance and can be used to implement synchronisation at a granularity coarser than $10ms$. In addition, RPC performance will only improve as technology progresses whilst user related synchronisation requirements will remain static. If this level of performance is inadequate then two courses of action are available:

1. synchronisation can be implemented at a coarser granularity by increasing the LSF *duration*,[6] thus reducing the performance demands.
2. the communication delay between the controlling and controlled application components can be reduced; for instance they could be co-located on the same machine or even in the same address space.

Whichever solution is adopted the stream and device interfaces remain unchanged,[7] and consequently the application need not be altered in any major way. In addition, if performance improves, then it is possible to decrease the synchronisation granularity or distribute the application components - again without requiring major application change. This is made possible by the use of well defined interfaces for streams, stream synchronisation and devices, which in turn allow synchronisation granularity and distribution to be viewed as *configuration* and not *design* options.

IMAC devices provide an effective means for managing three forms of heterogeneity:

Implementation heterogeneity: a single device interface may have multiple implementations, thus allowing for implementations of the same interface over a range of platforms.

Device heterogeneity: conformance allows devices which are of different, but conforming, types to be used in an identical fashion and in place of one another. If device X conforms to device Y, then X may be used in place of Y, thus providing a means of managing heterogeneity among device interfaces.

Communications heterogeneity: the underlying communication system is hidden and the protocol used to transport invocations is transparent to the application. Even though QoS has been introduced to allow control of the communications system, great care is taken to ensure that QoS is specified in a declarative fashion and using application level terms; section 9.2.2 discusses this in more detail.

The potential problem associated with this approach is that the management of streams and devices is likely to be complex and difficult to implement. The concept of *orchestration* was developed specifically to tackle this problem, and as demonstrated by the prototype implementation

[6] For an isochronous stream, the duration is that for every LSF used; for a non-isochronous stream the duration is that for the LSF which consumes the minimum amount of time.

[7] Except for the addition of a new LSF if a suitable one is not already available.

(chapters 7 and 8) provides an effective solution to the problem. In particular, the use of language extensions provides a clean and powerful interface for managing streams and devices. The principal omission is the lack of an automated means for implementing stream synchronisation operations; section 9.4.2 presents a design for such a scheme. Even so, as shown in chapter 8, the orchestration interface provided is effective at reducing the complexity visible at the application level.

The pervasive use of well defined interfaces for all aspects of synchronisation and heterogeneity management naturally leads to the construction of an open system. The result is a system which allows the interoperation and interconnection of its components and which can be incrementally extended by the addition of new components without disturbing existing components. In addition, the configuration of the system, both in terms of the physical location and the logical interoperation of its components can be changed without changing the components themselves.

9.2.2 QoS

IMAC QoS provides the mechanism for expressing media and application specific requirements. The benefit for applications is that they can directly specify their communication requirements and expect them to be guaranteed. Similarly, the communications system is better able to efficiently manage its resources given an indication of application requirements. The pitfall associated with QoS is that applications will become tied to particular, implementation specific, QoS offers. However, as shown in sections 5.2.7 and 7.3, IMAC and its implementation provide an effective scheme for avoiding this pitfall. That is, QoS is specified in a declarative fashion, using application terms, and a mechanism is provided for mapping application QoS requirements into lower-level communication requirements - thus hiding the details of how the application's QoS requirements are satisfied.

Other important features are that applications can determine the QoS currently in use and vary their behaviour accordingly. Applications may specify and dynamically vary QoS at run-time and are presented with a simple programming interface for using QoS. QoS is implemented on an end-to-end basis. The performance evaluation presented in section 9.1 shows that QoS can be implemented at minimal cost. In addition, the scaling properties of the system are preserved for communication not requiring QoS.

The process of designing and implementing QoS produced a number of interesting results and observations:

- the design and implementation of QoS is complex.
- end-to-end QoS negotiation was surprisingly easy to implement.
- care had to be taken to preserve the performance and scaling properties of the resulting communication system.
- there is a fundamental relationship between QoS and minimising multiplexing.
- QoS is a powerful tool for managing system configuration.

The representation for QoS offers and constraints and within the extended interface references was complex to implement and hard to debug. The Testbench communication system had to be extensively modified and new interfaces designed and implemented before QoS could be accommodated. In particular, the *local channel resource* and *dynamic QoS negotiation* interfaces were devised specifically to support QoS. Local channel resources are fundamental to the successful implementation of QoS since they provide the basic mechanisms for associating resources with

a given QoS offer and for preserving the performance and scaling properties of non-QoS based communication.

Once the in-capsule code was functioning at the client and server, it was relatively straight forward to implement end-to-end QoS. This was largely due to the recursive use of the Testbench invocation scheme to communicate the QoS requirements of the client to the server. This out-of-band approach preserved the performance of the existing RPC system by moving all QoS processing off the critical path, to when interface references and new bindings are created and traded.

The relationship between QoS and minimising multiplexing can be seen when:

- QoS offers are viewed as defining a set of protocol stacks, any of which can be used to provide the required QoS.
- communicating parties, employing minimal multiplexing, are viewed as using their own, private, protocol stack.

The algorithm for QoS specification (see section 5.2.7.1) is responsible for creating this set of protocol stacks and can be viewed as identifying all of the available non-multiplexed communication paths. In this way QoS provides the mechanism for identifying and selecting between non-multiplexed protocol stacks, whilst minimising multiplexing allows QoS to be implemented by reducing performance and QoS cross-talk, (see section 4.2.4.2).

As described in section 7.3.1.2, it is possible to use QoS offers to describe the configuration of the communications system. This configuration is then encoded in an interface reference and thus made available to any client wishing to communicate with the interface provider. The client is then able to match the configuration of its own communication system with that of the server; i.e. to avoid impossible protocol combinations or to choose the most efficient protocol. The advantage offered by this scheme is that decisions regarding which protocol to use are deferred until run-time, and are made with a knowledge of the server's communication system. The disadvantage is that the extended interface references are large and expensive to communicate, but fortunately, such communication is relatively infrequent.

9.2.3 Unsatisfied Requirements

This dissertation has not addressed many of the requirements made for the implementation of multimedia communications protocols and for resource management policies suitable for multimedia applications. This is partly due to a lack of time and resources, and also to the fact that these areas are already the subject of a great deal of research. However, by examining these requirements in detail and taking them into account when designing IMAC and its implementation, a framework has been provided within which such protocols and policies can be incorporated. In addition, the performance penalty imposed by this framework has been shown to be minimal (see section 9.1) and as a result any performance advantages offered by advances in protocol design and resource management should be reflected by a corresponding improvement in future IMAC implementations.

9.2.4 Other Architectures

This section compares the flexibility and suitability of IMAC to other multimedia architectures; finally, the relationship of IMAC to OSI is discussed.

9.2.4.1 VOX and CMEX

VOX and CMEX have very similar architectures, which are in turn based on that of the X Window System [**Scheifler86**]. In this architecture *all* system functions are implemented by a *single* server and applications are implemented as clients of this server. The protocol used for communicating between clients and this server defines the functions provided by the server and is highly specialised for the job in hand. The problem with this approach is that all system functions are implemented in a single, monolithic, piece of software, which must be modified whenever system functions, however minor, are to be modified or extended. The degree of modification required depends on how closely the newly designed functions match those anticipated by the original design and implementation of the server. Such a monolithic structure is too restrictive for the distributed environment assumed by IMAC. Therefore, IMAC represents separate system functions as separate devices which can be implemented, modified, added and removed, independently of one another. The communication protocol used is far more general and accommodates a far wider range of applications than those used for VOX and CMEX.

The server based approach does offer the potential advantage of being able to easily synchronise operations across multiple devices. It is relatively straight forward to implement such synchronisation provided that all operations pass through the same server *and* all of the devices to which the operations apply are implemented within the server, (i.e. are a fixed communication delay away from the server).

However, IMAC being an extension of a distributed system, and in particular of ANSA, is able to make use of any general purpose distributed computing mechanisms for managing and synchronising operations across multiple, distributed devices. In particular, some form of group execution or reliable multicast protocol, as provided by ISIS [**Birman87**] for example, could be easily incorporated into IMAC. Incorporating such a protocol into VOX or CMEX would be inordinately difficult.

9.2.4.2 Extending UNIX

The work carried out by Leung et al, described in section 3.11, shares a common goal with IMAC: namely, the desire to provide a powerful programming interface for multimedia applications. The differences between the two lie in the initial architectural assumptions and in the means chosen to achieve this goal.

IMAC assumes a highly heterogeneous environment, both in terms of hardware and software, whilst Leung assumes the use of a single operating system, UNIX, and a single protocol architecture. In Leung's system separate streams are multiplexed over a *single* communication channel to provide synchronisation. Therefore, synchronisation policy is implemented within the communications system, which is in turn implemented within the UNIX kernel.

Leung's programming model is based on asynchronous message passing and is implemented by the Non-Procedural Programming (NPL) language. IMAC on the other hand provides a procedural model based on the invocation of operations provided by interfaces. The model adopted by IMAC, inherited from ANSA, allows effective heterogeneity management and compile-time type checking, whilst the event driven model does not. In particular, the event driven model used assumes that the same data *representation* is used by all system components.

9.2.4.3 IMAC and OSI

The scope of IMAC, although considerably larger, does overlap with that of OSI. This is most easily seen if IMAC is viewed as an MMCS.

The IMC and Communications components of IMAC overlap directly with the OSI protocol stack, whilst the User Interface component had no corresponding component in OSI. Some portions of the DPC may overlap with the session and presentation layers, however the DPC provides many functions in addition to session and presentation functions. The IMAC application component is intended to support arbitrary, user created applications, as opposed to a standardised set of common applications as specified by OSI.

The Communications component has the greatest correspondence to OSI, since it can viewed as providing the network, transport and optionally the session and presentation layers if they are not part of the DPC. Whether layers 5 and 6 are in the DPC or the Communications component is an implementation issue. The major difference is that IMAC provides comprehensive support for QoS, not only within the Communications system, but also allows the DPC and application components access to QoS.

The IMC component may make use of the Communications component to transport its control operations and even to transport multimedia data. Therefore it appears as part of the applications layer in OSI.

9.3 Requirements for Future Systems

This section presents requirements for the design and implementation of future systems which are better able to support IMAC. Although the prototype implementation was severely restricted by the use of UNIX, this section is not intended as a critique of UNIX and the discussion is kept as general as possible. Each MMCS component is taken in turn.

9.3.1 Information Media Component

As mentioned in section 6.4, if full use is to be made of device conformance then devices supporting multiple media should allow the use of a single medium in isolation of the others.

Multimedia hardware devices must provide sufficient information regarding their current state to allow the implementation of IMAC streams. In particular, the structure of the information media must be visible at the device interface and be accessible whilst the medium is being communicated.

The problems discussed in section 4.1.1 of inappropriate workstation design could be solved by adopting a workstation architecture with multiple, concurrent, data paths, usually referred to as a switch fabric, between high speed devices, as illustrated in figure 9.8.[8] Ideally full connectivity should be possible, allowing for any device to be connected to any other. Such an architecture is better able to support multiple, guaranteed communication paths and leads to a fully integrated workstation which can process multiple information media as easily as current workstations process text. For lower speed devices a single bus may be used to reduce cost. Unfortunately, such switch based architectures are unlikely to be widely available for some years yet.

The use of the Fairisle network (section 3.1.6) for constructing such a switched multimedia work-

[8] Note that devices which can act as data sources and sinks appear on both sides of the switch

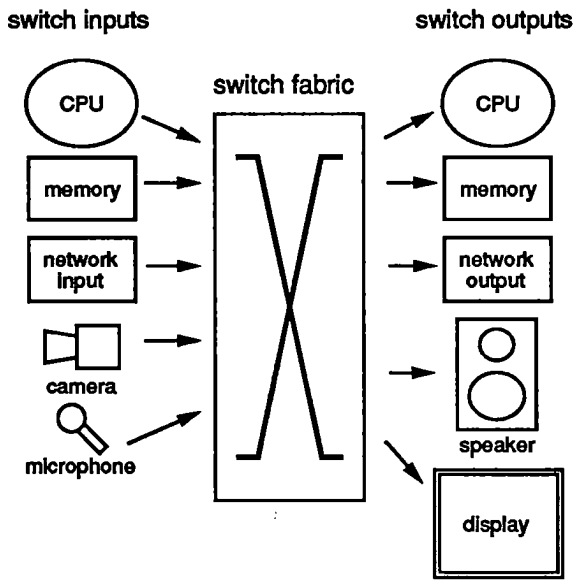

Figure 9.8: Switched Multimedia Workstation

station is currently being investigated at the Computer Laboratory [**McAuley90**].

9.3.2 Communications Component

The principal requirement for future multimedia communications protocols are described in detail in section 4.2.1, and can be summarised as the ability to support multiple service requirements and the guaranteed provision of application specified QoS on an end-to-end basis. As shown in section 3.3 this area is already the subject of much research.

The operating system interface presented to the underlying communication system must preserve the QoS of the communications beneath it. This requires the provision of concurrency within applications and the ability for each concurrent thread to wait for receptions on a different communications channel. UNIX does not support any such concurrency and forces all network receptions to be multiplexed via a single call to `select`. This makes it very difficult to take full advantage of any QoS provided by the underlying protocols and introduces additional multiplexing.

9.3.3 Distributed Processing Component

Of the problems outlined in section 4.1.3 heterogeneity has been directly addressed by IMAC, but only very simple resource management policies have been implemented.

If multimedia applications and devices with predictable performance are to implemented then some form of real-time[9] scheduler is required. Applications need to service incoming synchronisation operations within strict timing constraints, whilst devices need to transmit and receive stream data, and issue synchronisation invocations within strict time constraints. The communication

[9]The term *real-time* is used somewhat loosely to refer to schedulers which support some notion of time and schedule processes accordingly.

between devices and applications requires predictable performance, the QoS facilities provided by IMAC can be used to specify and implement these communication requirements. Although IMAC has not provided any support for CPU scheduling, the LSF structure used for streams does provide an application independent means for expressing scheduling requirements. That is, applications can specify their timing constraints in terms of LSF's and leave it to the system implementation to map these to the parameters used by the underlying scheduler; commonly used scheduling parameters include deadlines and thread priorities. The mapping of LSF's to scheduling parameters should be specified in a declarative manner and can be viewed as part of the system configuration.

9.3.4 Application Component

The application designer and implementor is faced with three major problems. This dissertation has addressed the first of these; namely, the almost complete lack of distributed system support for the construction of multimedia applications.

The second problem is the absence of application level concepts and building blocks for use in applications; this is a direct result of the short time that research in this area has been active. MUSE (section 3.7.2) is one system that provides such building blocks.

The final problem is that of deciding "what an application should do"; that is, what functionality to provide, how to provide it, and also what not to provide. This problem can only be overcome by building working systems whose use can be evaluated and the results of this evaluation fed back into the refinement of existing systems and the design of new ones.

A large application area for multimedia communication will be CSCW. Such applications require the ability to communicate the same data, or media stream, to multiple locations; i.e. they require one to many, *multicast*, communication. The term multicast is used in two conflicting ways: the first refers to reliable group multicast (as exemplified by the ISIS system [**Birman87**]) whilst the second refers to multicast delivery without any implicit reliability or any other QoS requirement. CSCW applications require both types of multicast. Reliable multicast can be used to maintain the consistency of the shared workspace and conference membership, whilst unreliable multicast may be used to provide efficient multi-site delivery of voice and video which do not require high reliability or consistency.

9.3.5 User Interface Component

People are adept at communicating and manipulating multiple information media simultaneously; they are also capable of identifying and understanding the relationships between such multiple media. The problem facing the user interface designer is how to integrate new information media into a single coherent user interface and how to capture, represent, communicate and recreate the relationships between multiple media. The word *seamless* is often used to describe the level of integration sought by user interface designers.

A central aim of multimedia communication is to allow a single user to use a computer as a tool for communication with several other, physically distant, users. This means that the next generation of UIMS' which will implement user interfaces to such multimedia communication must extend their view of human computer interaction beyond the current situation of a single user interacting with a single computer. Therefore, the UIMS must cope with multiple sources of human input and the very much larger class of errors introduced by the presence of a network and distribution. These errors include communication errors due to the network itself and partial system failures which occur when part, but not all, of the distributed application managing the communication fails. In addition, new UIMS' will have to support the variations in the degree of interactivity required

by different applications, and even by the same application when using a different communication QoS.

CSCW, as shown by the problems encountered by MMConf (section 3.8.4), places even greater demands on existing user interfaces and user interface management systems. In particular the ability to customise a user interface to personal taste must be carefully managed in a shared workspace to avoid interference between multiple users.

Capturing and presenting the relationships between multiple information media is another problem for current systems since they often assume that separate user activities are independent of one another. Some means of expressing these relationships in an efficient and non-intrusive manner must be found.

9.4 Future Work

This section presents designs for un-implemented portions of IMAC for extensions to the prototype implementation, and suggestions for improvements if the current system were to be re-designed and re-implemented.

9.4.1 Multi-Channel Synchronisation

Section 5.2.7.5 described the need for multi-channel synchronisation and outlined the various levels at which it might be implemented. Within ANSA, work is already underway on the implementation of atomic transactions, concurrency control and a group execution protocol; all of which require some form of multi-channel synchronisation. The design presented in this section is for a lower level, stub code, implementation and could be used to implement the higher level mechanisms mentioned above.

This design requires the extension of IDL to allow the definition of *synchronisation groups*, to which operations requiring multi-channel synchronisation must belong. Such groups are represented as two components: a sequence counter and a set of operations to be sequenced by this counter. A new IDL statement is required to define such sequence counters and the existing syntax for operations extended to include a list of the sequence counters, and therefore synchronisation groups, that each operation belongs to. The client stubs are responsible for implementing sequence counters as monotonically increasing variables and for generating new values for each operation invocation. This new value is passed to the server as part of operation invocation. The server maintains a record of the last sequence number received and will only dispatch an invocation for the next sequence number; invocations with sequence numbers ahead of the expected one will be buffered. The process of ordering invocations at the server is called *collation*. It should also be possible to specify a maximum time for which the server will wait for the next expected invocation if a future invocation has been received and buffered. If this timeout is exceeded then a communication error should be reported.

In order to support a range of different collation algorithms applications should be able to specify the use of application supplied collation algorithms or modules.

9.4.2 Implementing Synchronisation Operations

As mentioned in chapter 8, there is no automatic support for implementing synchronisation operations within the control component[10] of a multimedia application, nor are there any associated synchronisation primitives. This section presents an outline design for such a scheme.

The requirements for these synchronisation primitives are that they allow the application to wait for multiple synchronisation operations simultaneously and allow a timeout[11] to be associated with such a wait. In addition, some form of *synchronisation expression* is required for specifying which operations are to be waited for; for instance a statement of the form "wait for operation x or (y and z)" should be supported. The points below outline `WaitFor` and `ReturnInvocation` synchronisation primitives to wait for, and return from, synchronisation operations respectively, and provide the same functionality as that assumed by the pseudocode used in chapter 6.

- each event and stream synchronisation operation is allocated an area of memory, which has sufficient room for the following information:
  - an indication of the operation using this memory.
  - room for the operation's arguments.
  - room for the operation's results.
  - a semaphore which the operation will wait upon for its results to become available in its memory area.

  When invoked, each operation will place its arguments in the memory area provided for it and then wait on its semaphore before returning.[12]
- a pseudo operation is used to represent a timeout and is treated in the same way as any other synchronisation operation. When the timeout expires this operation is invoked; it has a single argument, identifying the timeout which has expired. The timer management facility, section 7.2.1.2 can be used to implement this.
- a `WaitFor` statement is provided and waits on a single semaphore which can be signaled by *any* synchronisation operation.
- when this semaphore is signaled the implementation of `WaitFor` must examine *all* of the synchronisation operation memory areas to determine which ones have been invoked. Using this information it must decide if the synchronisation expression has been satisfied. If it has then `WaitFor` can return and pass control back to the application, if not then it simply waits on its semaphore once more.
- the application is now released, to examine the arguments of the invoked synchronisation operations, to perform any processing it requires and to set values to be returned as results. The `WaitFor` statement must return an indication of where these arguments and results are to be found and set.
- the `ReturnInvocation` statement simply signals the synchronisation operation's semaphore and thus allows it to return.
- STUBC is responsible for generating the body of the synchronisation operations and procedures for accessing and setting the arguments and results.
- PREPC implements the `WaitFor` and `ReturnInvocation` statements.

[10] That is, the component of an application responsible for *receiving* synchronisation operation invocations.

[11] Required to detect communication and synchronisation errors.

[12] Announcements need not wait on this semaphore.

9.4.3 Device Operation and LSF Synchronisation

The wide variety of mechanisms which can potentially be used to transport stream data makes it impossible to design an automatic scheme for synchronising device operations to LSF boundaries which can be used in all implementations.

One approach is to use the `WaitFor` statement described in the previous section and to view the portion of the device responsible for recognising LSF boundaries as a synchronisation operation and assign it an area of memory in the same way as these operations. When an LSF boundary is detected, an indication of which LSF has just been seen will be written into the argument portion of this memory area and `WaitFor`'s semaphore signaled. This code will have to be written by hand. The device operation code may then use `WaitFor` to wait for such LSF boundaries. Depending on the style of synchronisation required (i.e. blocking or non-blocking, see section 5.2.4) the device operation may wait until the LSF boundary is reached before returning, or return immediately and issue a synchronisation operation when the LSF boundary is reached.

9.4.4 Streamed Invocations

There are some applications which do not need to return any results for many of their invocations; that is, they use asynchronous invocations, (i.e. announcements). Window systems, and in particular X11, are examples of such applications. The fact that announcements do not return any results means that multiple announcements can be buffered and transmitted in a single network packet. The X11 window system protocol uses this approach to buffer large numbers of small requests and achieves better performance than that currently possible using synchronous invocations. The use of LCR's within the communications system allows such buffering to be implemented across multiple invocations and to be controlled via the QoS interface. That is, a QoS request can be used to specify the amount, if any, of buffering required. However, such applications still need to occasionally synchronise client and server and rely on the use of a synchronisation invocation to achieve this. That is, a synchronous invocation (interrogation) forces any buffered announcements to be transmitted immediately and before the interrogation. If the announcement and interrogation use the same encompassing QoS, then they will also use the same communication channel and hence the required synchronisation is provided by multiplexing all communication over this single channel. In the more likely event of using a different QoS for announcements and interrogations then the scheme for multi-channel synchronisation described in section 9.4.1 can be used.

9.4.5 Re-design and Re-implementation of IMAC 3.0

The basic structure of IMAC 3.0 appears sound and provides a strong framework for experimentation with multimedia protocols and resource management policies to support multimedia. In particular the use of LCR's to represent QoS resources, channels to represent addressing and sessions to tie the two together works well.

Work is required to design and implement a more compact representation of QoS for encoding in interface references. The current interface reference structure is clumsy and verbose, and any future implementation should improve on this. Any new representation should also provide some form of *type* field to allow multiple types and versions of the interface reference structure to be distinguished and used simultaneously.

The current implementation, both of 3.0 and of the extensions made as part of IMAC 3.0, have paid little attention to performance. Significant speed improvements could be achieved, simply by re-implementing the same design whilst paying attention to performance issues. In particular

greater use of pre-allocated, per-module memory allocation and improved marshalling code would increase performance. The pre-allocation of resources supported by the LCR interface is currently not used; its use, especially for pre-allocating memory buffers would also improve performance.

The User Locator could be extended to support more sophisticated location and access policies. In particular, the person being located should be able to control whether or not they can be located and by whom they can be located.

9.5 ANSA and IMAC

The ANSA architecture has proved an excellent basis for IMAC, and no major modifications were required to support IMAC. As a result, IMAC can be viewed as stylised usage of ANSA to support multimedia applications. The architectural impact on ANSA has been confined to three areas:

- the extension of interface references to include QoS information in addition to addressing.
- the extension of the conformance relationship to include devices and streams.
- the incorporation of per-operation QoS to control the underlying communication system.

The implementation of ANSA provided by the Testbench required extensive modification to support the prototype implementation of IMAC. These modifications were concentrated on the removal of communications assumptions which were invalid in a multimedia environment and on the implementation of QoS. The language based approach taken by ANSA and the Testbench has been instrumental in the successful implementation of IMAC. Not only does it provide a clean and powerful interface for the application writer but allows the underlying system to be developed independently of this interface and therefore of applications. The latter advantage helped reduce the time taken to implement IMAC 3.0.

9.6 Future Research

This section outlines some broad directions for future research into multimedia communication systems. More detailed suggestions have been made as part of the requirements for future systems discussion. These can be categorised as follows:

- resource management policies, especially CPU scheduling, to support multimedia. Promising areas of research are the use of the LSF structure provided by IMAC streams to specify application timing requirements and the extension of the cost functions provided by the prototype implementation to represent a wider variety of resources.
- continuing research into multimedia communication protocols and in particular into the practical use of QoS.
- multi-cast and multi-drop streams and in particular the detailed connection, QoS, synchronisation and translation management of such multiway streams.
- research into multimedia applications and user interfaces, as discussed in sections 9.3.4 and 9.3.5.

The contribution of IMAC is to provide an architectural framework for this research, and its prototype implementation can be used as a starting point for a wide variety of multimedia applications. This future research can only be carried out as part of a wider and longer-term effort to build usable and used multimedia applications. Therefore, the next logical step is to use IMAC to build such multimedia applications and thus extend its evaluation to conditions of practical use.

10

Conclusion

The thesis of this dissertation is that an architectural approach is effective at increasing functional integration and that this will reduce the difficulties currently encountered in constructing multimedia applications.

The survey of background and related research clearly showed that each of the majority of such projects has concentrated on one very well defined area of multimedia systems, and paid little or no regard to other system components. This piecemeal approach has led to the lack of functional integration discussed in the introduction to this dissertation.

The IMAC architecture represents an attempt to pull together the results of this disjointed research into a coherent architectural framework and thus provide a sound and necessary basis for continued research into multimedia applications.

A detailed investigation of the problems and requirements for multimedia was conducted and is presented in chapter 4. The results of this investigation led to the formulation of the new architectural principles of media separation, choice and evolution, and the already recognised principle of modularity. These were then used to guide the design of the IMAC architecture.

The principle features of IMAC are the support it provides for the synchronisation of multiple, related, media streams, for the management of heterogeneity and the expression of media and application specific requirements. The result is a distributed, open, architecture providing extensive support for the construction of multimedia applications.

Evaluating an architecture is a difficult and drawn out process. The approach taken in this dissertation has been to construct as complete a prototype implementation as time and resources allowed and to use it to evaluate the feasibility, and identify the strengths and weaknesses, of the architecture. A complete evaluation, and in particular the construction of a suite of usable, and used, multimedia applications would take a far greater amount of time than that available for this dissertation.

The results of this evaluation are presented in chapter 9 and show that the architecture is feasible, in that it can be implemented and implemented efficiently. The performance of the prototype implementation is comparable to that of the Testbench version on which it was based, and also

preserves the scaling properties of the Testbench. Care had to be taken when implementing complex features such as QoS and the associated QoS negotiation algorithms to preserve performance and scaling properties. The resulting implementation is therefore well placed to take full advantage of the performance advantages promised by advances in communications protocols, networks and resource policies designed to provide the guaranteed, real-time performance required for multimedia.

The strengths of the architecture and its implementation lie in the well defined interfaces provided for stream synchronisation, heterogeneity management and QoS expression. The result is a system which can be used to implement a wide variety of media and application specific synchronisation requirements and which can be re-configured to suit the current implementation environment and available technology.

The weaknesses lie in the lack of attention paid to the design and implementation of resource management policies to support multimedia applications. However, section 9.3.3 makes requirements for future implementations of CPU schedulers. In addition, the potential weaknesses of poor performance, complexity for the application programmer and non-portable QoS requirements have been avoided as discussed in sections 9.2.1 and 9.2.2 respectively.

The result is an architecture and prototype implementation of that architecture which are successful in increasing functional integration and provide a framework for the implementation of *all* components of a multimedia communication system. In this way, the process of constructing multimedia applications is made considerably easier.

A

ANSA Summary

This appendix provides a brief summary of the ANSA architecture and of an implementation of that architecture called the ANSA Testbench. It is by no means exhaustive and only covers aspects of ANSA which are of direct relevance to this dissertation. A more complete description of the architecture and the Testbench can be found in the ANSA Reference Manual [**ANSA89b**] and the Testbench Implementation Manual respectively.

The material presented in this appendix is largely taken from the ANSA submission to the Open Software Foundation's (OSF) Distributed Computing Environment Request for Technology [**ANSA89c**].

Sections A.3 to A.12.3 cover particular aspects of the ANSA architecture and Testbench in greater detail.

A.1 ANSA Architecture

ANSA consists of five related models called the enterprise, information, computational, engineering and technology models. This dissertation is only concerned with the computational and engineering models and this summary concentrates on these.

The importance of distinguishing between a distributed computing environment as seen by an application programmer and as seen by a system programmer is of primary importance to ANSA. Many environments have an "application programmer's interface" as the dividing line between the two views. In practice this interface often turns out to be clumsy and lets too much of the system detail show through. Moreover, checks must be made at run time to ensure that application programmers are using the interface correctly. ANSA has taken a programming language view; that is, distributed computing concepts should be represented by extra syntactic constructs to be added to existing programming languages. These can then be compiled directly into calls at the systems level. The advantages of this approach are threefold:

- a simple programming model for applications programmers.
- checking at compile time rather than run time.
- independence of application programmer view from system view.

The first two increase the confidence that application programmers have in their programs. The third provides for separate evolution of the two views of the environment, allowing for modifications to be made to one without unduly disturbing the other and helping to make applications and systems more "future-proof". For example, generation of stub routines from an interface specification permits error free marshalling/unmarshalling of application data between interacting objects, and allows the marshalling/unmarshalling operation to be modified independently of the interface and transparently as far as the application writer is concerned.

A.1.1 Computational Model

The computational model defines the programming structures and program development tools that should be available to distributed application programmers, in whatever application programming language they choose to use.

The purpose of the ANSA computational model is to provide a framework for describing the structure, specification and execution of distributed application programs. Distribution brings additional constraints and complexity to application programs but also provides the means for enhancing certain qualities such as performance and reliability.

The design philosophy for the computational model has been to find the smallest number of concepts needed to describe distributed computations and to propose a declarative formulation of each concept when the model is realised in a programming language. In comparison to an imperative approach, this declarative approach provides greater scope for compile-time checking of program safety, the automatic generation of support code and optimisation of special cases by compilers and other development tools. These are important attributes of a distributed computing environment, since they help to reduce the additional complexity that is brought into the application programmer's world.

The computational model is expressed in terms of objects which interact by passing messages. These objects may be supported by separate processors, thus introducing the possibility of both concurrency and independent failures. This gives rise to the need to resolve conflicts of access and failures; this can be done either by careful programming, or by assuming special transparency mechanisms. Transparency can be selectively controlled to allow the designer to choose whether to deal with conflicts and failures explicitly or not.

The model distinguishes between interfaces (what an object does) from object construction (how it does it) so that evolution of the system can be achieved smoothly - e.g. the removal of failed or obsolete components and the introduction of repaired or enhanced components.

This model addresses the topics of:

1. modularity of distributed applications.
2. naming and binding of module interfaces.
3. access transparent invocation of operations in interfaces.
4. parameter passing scheme.

5. configuration and location transparency of interfaces.
6. concurrency and synchronisation constraints on interfaces
7. atomicity constraints on interfaces.
8. replication constraints on interfaces.
9. extending existing languages to support distributed computing.

Maximum engineering flexibility is obtained if all computational requirements of an application are expressed declaratively. This permits tools to be applied to the specifications to generate code satisfying the declared requirements.

A.1.1.1 Computational Model Concepts

A distributed system consists of a collection of objects that interact at well-defined interfaces. Objects are the units of structure, while interfaces are the units of binding.

Objects interact at interfaces; each interface is provided by one object and may be used by one or more objects. Interfaces are defined in terms of possible interactions. An interaction at an interface consists of a sequence of requests and responses where the requests are chosen by the user(s) of the interface and the responses by the provider.

All requests and responses must be part of an invocation. There are two kinds of invocation: those that consist of a request and a response are called interrogations, while those that consist of only a request are called announcements. There is no distinction between remote and local invocations. The notion of "locality" is not part of the computational model, apart from the distinction between an object's own interfaces and those of other objects, and variations in the fault models that may apply to invocations of different interfaces.

An operation is defined in terms of the way that the response in an interrogation is related both to the request and to other invocations - those in this particular interaction and also those in other interactions with the same object. Every invocation permitted at an interface must be in terms of one of the operations of that interface.

Every invocation of an operation must have the same structure. This rule both depends upon and allows the construction of a type system for the computational model. This structure describes how many elements are passed in a request or response and what can be done with them. The description of this structure is called the signature of the operation.

An operation may not be invoked in both an interrogative and an annunciate style; this may be considered a special case of the "same structure" rule.

Request structures are very similar to argument structures of function calls in typical programming languages. All the requests in invocations of an operation contain the operation name and a fixed number of elements of known types.

The rule for permitted responses to an operation invocation is less conventional; it provides both for a choice from a defined set of structures and, for each choice, it allows a choice-specific fixed number of elements of known types. The possible responses to an operation invocation are partitioned into groups which are called terminations; each termination is given a name. All responses in a termination have the same structure, the termination name plus a fixed number of elements of known types.

Failure of an object supported by an independent mechanism becomes visible when an operation in an interface provided by that object is invoked. Responses reporting failure can be added as one or more standard terminations for every operation. The question of exactly how many terminations of this kind should be added and what elements (if any) they should carry depends upon the degree of distribution transparency selected for the interface.

Closure of the interaction model demands the ability to pass interface references as arguments or results. The ability to pass more traditional data types (e.g. constants, integers, structures) is also modeled as passing references to interfaces which contain operations that can be invoked to read/write the value of the object. Treating these values as interfaces means that the elements that are passed in requests and responses are all interface references; the interaction model is uniform and relatively simple. This does not mean that the model must be mechanised in this uniform way. Moreover, the model becomes one of "information sharing" which is a more general paradigm that can be extended to include concepts such as bulletin boards, mailboxes and shared memory.

The type of an interface is given by the names and types of the operations in the interface, together with a definition of how operations influence one another. The operational structure part of an interface type is defined by a set of (operation name, signature) pairs.

The operational structure part of an operation type is given by the signature. An interrogation signature is a fixed-length list of interface types together with a response type. An announcement signature is a fixed-length list of interface types. A response type is a set of (termination name, list of interface types) pairs.

The computational model also addresses the following topics:

1. configuration and location constraints on interfaces.
2. concurrency and synchronisation constraints on interfaces.
3. atomicity constraints on interfaces.
4. replication constraints on interfaces.
5. naming and binding constraints on interfaces.

These constraints on interfaces appear as attributes of interfaces, of operations within interfaces, in interface specifications. These attributes cause specific engineering model mechanisms to be incorporated when realising a distributed system which includes a constrained interface.

A.1.2 Engineering Model

The engineering model defines a set of logical compiler and operating system components that realise the computational model in heterogeneous environments, namely:

1. thread and task management.
2. address space management.
3. inter-address space communication.
4. distributed application protocols.
5. network protocols.

6. interface locator.
7. interface traders.
8. configuration managers.
9. atomic operation manager.
10. replicated interface manager.

The engineering model shows the system designer the range of engineering trade-offs available when providing a mechanism to support a particular function defined in the computational model. By making different trade-offs the implementor may vary the quality attributes of a system in terms of its dependability (reliability, availability, performance, security, safety), performance and scalability without disturbing its function. This is an important feature of the ANSA architecture since it decouples application design from technology to a significant degree. By conforming to the computational model, a programmer is given a guarantee that his program will be able to operate in a variety of different quality environments without modification of the source. The engineering model gives the system implementor a toolkit for building an environment of the appropriate quality to the task in hand. In other words, by making this separation it is possible to identify what forms of transparency are required by a distributed application and to be able to choose the most appropriate technique for providing the required transparency for each application.

A.1.2.1 Engineering Model Concepts

The computational model requires a virtual machine (VM) which enables interactions between computational objects. The engineering model describes components from which this VM can be built in an environment of networked computers. The engineering model explains how to distribute the VM over a communication network. The VM is broken up into a nucleus object and transparency mechanism objects; it is also provided with processing and memory resources and connection to a network. The transparency mechanisms communicate with one another via the nucleus and the network to achieve the desired transparency. The collection of computational objects, transparency mechanisms, interpreter and nucleus forming a node of a network is called a capsule.

The nucleus is the engineering object which encapsulates all of the heterogeneity of processor and memory architectures. It provides the functions needed to support concurrency and access transparent interaction between computational objects. The nucleus is defined in terms of conventional hardware resources: processors, memory, inter-processor message systems and clocks. Different capsules may consist of different kinds of processors, different kinds of memory and have different representations for computational objects.

Many systems provide the means to partition resources between address spaces so that each address space can be treated as an independent node. This is modeled as the creation of capsules which transparently share resources with one another. A capsule is mapped to the corresponding abstraction in the local operating system - e.g. a UNIX process.

A capsule interpreter consists of one or more virtual processors executing the instructions of the object and a distinguished processor for synchronising the other processors and interpreting inter-object interactions (e.g. `call`, `cast`, `fork`, `join`).

A thread is a sequence of instructions within a computational object that can be evaluated in parallel with other threads, subject to synchronisation constraints. A thread represents a unit of potentially concurrent activity. A task is a virtual processor which provides a thread with the resources it requires. To make progress, a thread must be bound to a task.

In a distributed application there may be many threads (e.g. 100's or 1000's); it is important only to allocate resources to a thread when there is a processor available to run it. All capsules are multi-threaded; a capsule may optionally be multi-tasking. Threads only identify potential concurrency, while tasks provide the resources for real concurrency.

In addition to the implicit synchronisation among threads caused by references to the instructions of the interpreter's distinguished processor, explicit synchronisation among threads occurs when the threads employ event count objects to achieve mutual exclusion. These operations are generated from concurrency constraints written in interface specifications. Successful mutual exclusion also relies upon the use of sequencer objects.

The communications model is based upon remote procedure call (RPC) protocols extended to include stream-like interactions. A clear separation is maintained between the programming language aspects of RPC (which feature in the computational model), the service primitives to the RPC protocol, and the design of the protocol itself. This permits alternative language representations of RPC and enables the protocol to be operated over widely differing kinds of networks.

An assumption that underpins the engineering model is that bursts of simple interactions will be more common than sustained bulk data transfer. This means that the end-to-end latency of interactions is the key factor affecting performance. Consequently communications are closely integrated with the scheduler so that processors can be assigned quickly to respond to the arrival of incoming messages, and so that the path between a computational interface and the network involves as few steps as possible and the least amount of intermediate buffering.

Many communications systems suffer from the problem that they consume excessive local resources such as buffer space and timers as the number of network connections increases. This presents a severe scaling problem for computational models that emphasise concurrency where there may be tens or hundreds of parallel invocations outstanding at a time. The solution to these problems is to allow the greatest possible multiplexing of interactions over connections and to minimise the dependency on end-to-end network connections. This is achieved in the ANSA engineering model by dividing communications into three layers.

At the lowest layer are a number of Message Passing Services (MPS) that manage connection and disconnection, as well as the transmission and receipt of messages between nodes. All message passing services conform to the same interface.

Above them are the execution protocols that map computational model invocations onto message exchanges via the message passing services. All execution protocols have the same interface. Two execution protocols are currently described: REX (Remote Execution protocol) which is a protocol for single endpoint to single endpoint communication, and GEX (Group Execution protocol) which is a protocol for multi-endpoint to multi-endpoint communication.

The coordination between protocols and threads is the responsibility of session objects, which make up the third layer. A session represents local state about interactions with another object's interface. Both the client and the server maintain session information during an interaction and the function of a protocol, in addition to transporting data, is to maintain session state between the client and server session objects.

A.1.3 Overall Structure

The way in which the aspects of ANSA fit together is shown in figure A.1. The host systems represent the computers and networks used to resource the distributed computing environment. The nucleus components take these basic resources and extend them to provide a basic distributed computing environment. The nucleus components interwork to provide a basic support platform

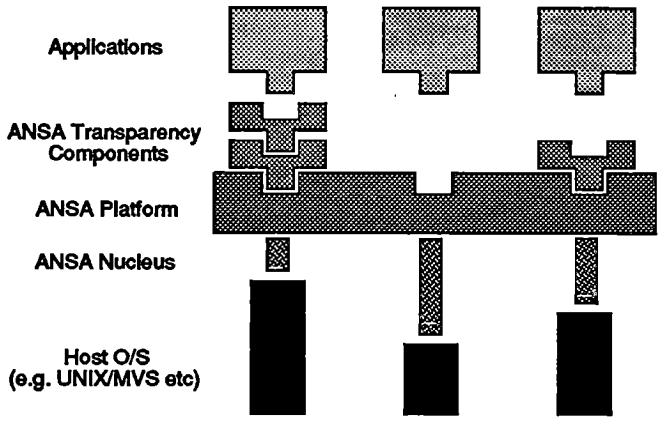

Figure A.1: An ANSA Distributed System

for distributed computing. The transparency components provide the various forms of distribution transparency as additional functions to those provided by the platform. The engineering model specifies the mechanisms needed to provide the various kinds of transparency and the protocols for interaction between nucleus components on different hosts. Application components are structured according to the computational model and the distributed computing aspects of the application are compiled into calls on the interfaces to the appropriate transparency and platform components.

The engineering model can also be taken as a template for the implementation of the nucleus, platform and transparency components, although this is not mandatory for either application portability across implementations, nor for interworking between them.

It is possible to conceive of multiple implementations of the architecture which make different engineering trade-offs. To provide interworking between systems that have made different implementation choices it will be necessary to provide gateway functions, but this will be confined to simple interface adaptors that match the different engineering trade-offs rather than changes to the applications themselves.

Many hosts will provide a range of functions and resources beyond those needed by the platform and may wish to contribute them to the distributed computing environment as potential application components. This can be achieved by extending the nucleus with additional distributed computing environment interfaces that map onto the locally available functions.

A.2 ANSA Testbench

The ANSA Testbench is a suite of C programs that conform to the architecture described in section A.1. These programs represent an instantiation of parts of the architecture intended for porting across the current generation of operating systems and network protocols.

The software includes the following modules

1. a threads management package.
2. an address space management package.
3. an inter-address space communications package (the "interpreter").

4. remote execution protocol (REX).
5. a group execution protocol (GEX).
6. an interface description language (IDL).
7. an IDL processor.
8. an application description language (PREPC).
9. a PREPC preprocessor for C.
10. a trader.
11. a factory.
12. a node manager.

The function of the threads management package is to provide for concurrency within an address space, if it is not provided by the host. Concurrency is needed a) so that servers can respond to multiple clients in parallel and b) so that clients can distribute computation in time (i.e. perform parallel tasks) as well as space (i.e. perform remote tasks), or both (run remote tasks in parallel).

The function of the address space management package is to complement the threads package with facilities for managing multiple stacks, communications buffers and a shared heap within a single address space. Multiple stacks are needed to be able to support true concurrency.

The function of the interpreter is to provide an implementation-independent standard interface for interactions between threads in separate address spaces.

The REX protocol provides for the transport of messages to implement the communications requirements of the inter-address space communications package. It provides functions of transport, error recovery, fragmentation of large messages and control of optional end-to-end connections.

IDL is used to describe interfaces between application components. It is derived from the Courier language developed by Xerox.

The IDL processor reads interface descriptions and generates libraries of stub procedures in C to convert the arguments and results of remote operation requests into a canonical format for transmission between heterogeneous systems.

The PREPC preprocessor for C extracts statements that augment a C program to bind to interfaces and invoke remote operations and translates these statements into calls of the appropriate stub procedures and inter-address space communication package calls.

The trader is a distributed application component which acts as a directory and management facility for distributed application components.

The factory provides a host-independent interface for the creation and destruction of new services.

The node manager provides a host-independent way of managing application components, particularly servers, on a per-host basis.

A.3 Trading

In a distributed environment it is necessary to provide a means by which the separate parts of a distributed application can rendezvous. This is called trading in ANSA and is an interface provided

to every object. Trading gives access to a directory structure that can be searched by path name, or by property values, or by some combination of both. A server can export an interface reference to the trading service to make it accessible to other programs. An import operation is provided to clients so that they can retrieve interfaces from the trader.

The trader performs type matching of imports and exports. This is done by maintaining a type name space in the trading system. The name space is an acyclic graph showing the conformance relationship between types. The trader provides operations for programs to add and retrieve types by name from the type graph. Imports and exports are typed: trading operations are parametised by types and the trading service will only search through exports of the required type and its conforming types when trying to match on path and properties.

The import operation returns an interface reference to the client. These references are unambiguous in a trading domain (a trading system can be structured as a federation of autonomous trading domains, managed by separate trading authorities, and a domain can be partitioned into a hierarchy of sub-domains). The importer can retain the reference for as long as required. If the distributed computing environment has location transparency enabled, the system will be searched to find the object. If location transparency is not enabled, an address hint in the interface reference is assumed to be absolute.

On export the interface provider may specify a list of (name, value) pairs, called properties, for that reference. Values may be textual strings, sets of textual strings or numbers. Import requests include a constraint expression to be applied to the property list of any offers found matching the specified type in the specified context. Constraint expressions may include tests for equality and inequality of string values, set membership and numeric comparison. Boolean connectives may be used to link sub-expressions, whilst minimum and maximum operators are provided for numeric expressions.

A.4 Configuration

There are two styles of distributed application programming. In the first the application is treated as a single large program. The program may be divided up into separately compiled components for reasons of efficiency and modularity. The program, once compiled, is then loaded into an appropriate configuration of computers and allowed to execute. In the second style, an application is treated as a number of separate programs which are independently compiled and loaded into individual computers. Programmers who work in this style often refer to "server" programs and "client" programs to indicate whether a program expects to be invoked by others, or whether it is responsible for invoking others. This second style is dependent upon some form of system directory (i.e. a trader) that enables servers to register their presence in the network and for clients to locate servers. In some systems the directory is itself a server, in others it is decentralised and broadcast algorithms are used to locate servers.

The distinction between these two styles is one of early versus late binding.

The first program style is potentially more flexible than the second, since the programmer can change the configuration arbitrarily by altering the assignment of components to computers. However, this style does not permit an application to be developed in which some components are developed independently of the others. This requirement is inevitable in an open systems context, because it is unlikely two organizations will be willing to lock together their programming environments in order to interwork over a network! On the other hand the second style tends to lead to a rather rigid assignment of components to computers, since externally visible names have to be invented and this can be inconvenient if the system supporting the application is restructured.

It can be concluded that both styles are appropriate in different circumstances, and the ANSA computational model provides for both.

The ANSA Testbench predominantly supports the client/server style of configuration, although an application programmer does have access to the facilities required to provide the distributed program style for himself.

A.5 Interfaces

The components of a distributed program may be written in different languages by different programmers in different places at different times. In order for a component to be constructed independently of another component with which it is to interact, a precise specification of the interactions between them is necessary.

This interface specification can be used to generate the interaction code as well as to independently check that one component is correctly interacting with another. Later on, when the program is finally assembled a check can be made that each pair of interacting components is using the same interface specification.

An interface specification requires an action specification, a data specification and a property specification. The action specification defines what are the actions that one program component may request another to perform. The data specification defines what types of data may be passed with each action request and reply. The property specification, expressed as a series of attributes, defines what transparencies and constraints are to be associated with each action, or the whole interface.

In general, an interaction specification may be bi-directional and specify what actions each of a pair of program components could request the other to perform. For simplicity, the ANSA computational model only contains uni-directional interface specifications which directly support the client/server style of interaction. A bi-directional interaction can easily be specified as a composition of two uni-directional interface specifications in opposite directions.

A program component acting as a client may request a number of other components to perform actions and thus needs a different interface with each of them. Equally, a program component acting as a server may perform actions requested by a number of client components. There is no reason to restrict a server to provide the interfaces with the same specification to each of its clients. Allowing a server to provide multiple interfaces with distinct specifications enables the computational projection to directly model the different roles of the enterprise model, especially with regard to access control. Multiple interfaces also enable knowledge of other components to be more tightly scoped. This conforms to the need to know principle required for program evolution and component re-use and is an important feature of the ANSA computational model.

The ANSA Testbench includes an interface definition language (IDL) for action and data specification. Support for attributes has yet to be implemented.

A.6 Operations

The action part of an interface specification could be functional or procedural. Distributed programs have separate and concurrent components which need to communicate in order to interact. Because it is hard to express communication in the functional style, actions are most naturally expressed as procedures in distributed programs and this approach is adopted for the ANSA com-

putational model.

In most programming languages procedures can have multiple arguments. Only a few programming languages permit procedures to return multiple results. Single results are asymmetrical and restrictive, especially in a distributed system where computational level interactions must be turned into message passing at some lower level with performance more dependent on message latency than message size. Consequently the ANSA computational model assumes multiple results.

Actions defined as procedures with multiple arguments and results provide the protocol part of interface specifications and are known as operations. The data specification requirement is provided by the definition of the argument and result types.

In the ANSA computational model, properties, such as transparency or synchronisation constraints, are specified declaratively for each operation or the whole interface specification and automatically inserted by the invocation mechanism.

The Testbench IDL provides for multiple arguments and results of various canonical concrete data types and composite types; a more general interface reference type can be used to convey abstract data types.

A.7 Invocations

Operations can only be invoked via their enclosing interface. Because the program component providing the interface may be remote an operation invocation must be via an interface reference in order to preserve access transparency.

The results of an operation are normally required before the client can proceed. In a distributed and therefore concurrent system this is achieved by blocking the client until the server has performed the operation and delivered the results. Thus the client and server are synchronised by the invocation. The local optimisation of a synchronous operation is the procedure call.

Where the client does not require an operation to deliver any results, the synchronous invocation suffers from latency and a reduction of concurrency in distributed systems. Asynchronous operations remove the latency and preserve the concurrency when immediate results are not required. The engineering level can make further optimisations by concatenating messages. Some systems describe such invocations as being "streamed".

There is no confirmation that asynchronous operations have terminated or even started, but if they are serialised with synchronous operations in the same interface then the result of a synchronous operation can indicate which of the preceding asynchronous operations failed. Thus a synchronous operation can be used to re-synchronise a client and server after a stream of asynchronous operations has transferred data at full speed (i.e. with no latency and a concurrent client and server). This kind of synchronisation must be explicit in the specification of the interface and therefore a conformance requirement for an implementation.

Distributed computing systems have unpredictable delays and partial failures, which may be silent. A client requires some way of indicating the urgency with which it requires a server to perform an operation and whether or not it is to keep trying forever or give up at some point so that corrective action can be taken.

Time is the only universal means of measuring urgency. Therefore a deadline may be required by each operation. Soft deadlines only affect the scheduling of an operation. Hard deadlines also prematurely terminate an operation when the deadline is reached.

The ANSA computational model provides for synchronous operations, asynchronous operations, streaming and deadlines.

The ANSA Testbench does not yet provide support for deadlines.

A.8 Terminations

There is not necessarily a "right answer" for every operation. An often used example is the popping of an empty stack, say of integers. A pop operation on a stack of integers normally returns an integer but if the stack is empty there is no integer that can be returned. The pop operation needs to return some other response that is distinguishable from the responses that indicate integers so that different actions can be taken.

Operations therefore require multiple responses (each of which may consist of multiple results). In the ANSA computational model, these responses are distinguished by name and known as terminations. Mechanisms are required for raising these terminations from within an operation and for changing the sequence of actions taken after an invocation of an operation depending on the termination it returns.

In any operation invocation one termination will cause no changes to the sequence of following actions. This termination is distinguished by not having a name and may be thought of as the "normal" response of the operation.

This termination mechanism can also be used by the engineering support environment for reporting engineering or transparency failures to the invoker of an operation.

The ANSA Testbench includes a limited form of the termination features defined in the computational model.

A.9 Objects

It is very hard in a networked system to achieve a workable, let alone efficient, implementation of global distributed storage. It is therefore necessary to look for a programming model which partitions and encapsulates state in order to describe the components of a distributed program. Such a model is common to the object-oriented programming languages such as Emerald and Argus. In these languages each object provides a set of operations by which it can be manipulated. Externally these operations are known by their names. The binding of operation names to computations that perform operations is an internal property of each object. Thus it is possible for different objects to respond to the same operation names, but to have different implementations of those operations.

This indirection from operation name to implementation has useful properties for distributed computations. Firstly it allows for heterogeneity: two interacting objects need not share the same infrastructure; they merely require communication between their infrastructures. Secondly the indirection provides a point to transparently insert the mechanisms that provide for communications. Thirdly the indirection makes it possible to substitute replacement objects without requiring any actions by the users of the object's operations. This has important benefits for software maintainability and evolution.

The ANSA (object-oriented) computational model is specialised for distribution by packaging sets of operations into interfaces to restrict the scope of operation names as tightly as possible and by

always accessing interfaces indirectly so as to preserve location transparency.

All data is stored in objects and accessed indirectly via interfaces. Thus the ANSA computational model only deals with interface references. It makes no statements about how values are represented. The obvious optimisations can be made when invoking references to the interfaces of local (co-located) objects, especially trivial ones such as integers and booleans, but such optimisations are definitely not part of the computational model and are an issue for the mechanisation of the model which is considered in the engineering model.

A.10 Type Checking

Because the component parts of distributed applications programs are separated in both space and time, extra care needs to be taken when composing them into a whole for final evaluation. The computational model concentrates on those checks designed to ensure that the assumptions made by the programmers in different places and at different times are still valid. These checks are generally known as type checks and will validate such things as the use of operators, types of data items, ranges of data items, representations of data, matching of operations used and provided, and allowed ordering of operations.

In addition to type checks, the computational model is also concerned with checking that components of an application that interact with each other have compatible transparencies. Other checks such as access controls and consistency constraints are passed through the computational model from information models for the system in question.

Checks can be performed in various epochs but steps must be taken to ensure that early checks are still valid during the final evaluation epoch.

In the ANSA computational model, interfaces are typed. An interface type describes both the operations in the interface and the properties of those operations in terms of transparency and security attributes.

In the ANSA Testbench some type checking is performed by the IDL and PREPC processors; the remaining type checking is deferred to the application programming language.

A.10.1 Type Conformance

An interface type defines the requests and responses permitted in an interaction where a client uses an interface of that type.

An interface type is defined by a set of signatures.

A signature specifies:

- the name of an operation.
- the number and interface types of argument parameters.
- whether the operation is an interrogation or an announcement.
- in the case of an interrogation, the response type for that operation.

A response type defines the set of permitted responses for an interrogation. A response type is defined by a set of terminations.

A termination specifies:

- the termination name.
- the number and interface types of result parameters.

Interface type X conforms to interface type Y if no interaction errors can arise from the use of an interface of type X as if it were of type Y.

An interaction error occurs:

- when a server receives a request which is not in the set of requests defined by the type of the interface provided by the server.
- when a client, having invoked an interrogation, receives a response which is not in the set of responses defined by the response type for that interrogation.
- when a client, having invoked an interrogation, receives no response.
- when a client, having invoked an announcement, receives a response.

For interface types that are not defined recursively, the conformance relation can be described in terms of signatures and response types as follows:

An interface type X conforms to an interface type Y if:

1. for every signature in Y there is a signature in X which defines an operation of the same name.
2. for each signature in Y the signature in X with the same operation name defines an operation with the same number of arguments.
3. each interface type in each Y signature conforms to the interface type in the same position in the corresponding X signature.
4. for every signature in Y which defines an announcement the signature in X with the same operation name defines an announcement.
5. for every signature in Y which defines an interrogation the signature in X with the same operation name defines an interrogation with a response type which conforms to the response type in the Y signature.

A response type X conforms to a response type Y if, for every termination in X:

6. there is a termination with that name in Y.
7. for each termination in X the termination in Y with the same name has the same number of parameters.
8. each interface type in each X termination conforms to the interface type in the same position in the corresponding Y termination.

These rules do not cover the cases where a type refers to itself as the type of an argument or result. In such cases, following these rules leads to situations where a type X conforms to a type Y if X conforms to Y. A more detailed definition of conformance, which can handle such recursive types, can be found in the ANSA Computation Model [**ANSA90a**].

A.11 Factory

Factories are the objects in a distributed system which facilitate the dynamic creation of other objects. A mapping from object factories to the facilities provided by the engineering model must be found in order to realise an ANSA system.

A.11.1 Computational Model Considerations

Object factories are implicit in the computational model. The computational model must define some means of delineating object boundaries in order to permit state to be encapsulated. Upon encountering one of these object boundaries, interaction with an object factory of the correct object type will occur, resulting in a new object instance. Attributes associated with the object-defining syntax may affect how the object is instantiated.

The arguments and results to an object instantiation depend upon the object type; as a result, there is a separate factory for each object type. The results from a successful object instantiation is a list of interface references which may subsequently be invoked by the creator or other objects.

A.11.2 Engineering Model Considerations

Capsules are engineering model concepts; they act as carriers of objects. Since one object may wish to create another, each capsule which supports objects of a particular type must provide the ability to cause another instance of that object type to come into existence. Likewise, it must support the ability to terminate an object instance.

The support environment for a distributed system must provide the following basic facilities:

- the creation/destruction of capsules
- the creation/destruction of objects within a capsule
- the creation/destruction of interfaces within an object

When an object is instantiated, it is expected that an initial sequence of instructions are executed; this initial sequence may cause interfaces to be instantiated and some initial interaction with other objects in the system (e.g. traders, authentication servers, time servers). The pseudo-computational statement:

```
create instance of Foo;
```

should cause the following things to happen:

1. check for the existence of a capsule which supports objects of type `Foo`; if not found, instantiate one using the capsule factory
2. instantiate an object in that capsule, returning the list of interface references generated by the `Instantiate` operation

A capsule factory permits the instantiation and termination of managed capsules - i.e. those that permit object instantiation/termination.

A.12 Node Manager

The node manager provides an architectural interface for the creation, simple monitoring and destruction of ANSA services on a single node. Both static and dynamic services may be created; static services may be declared as persistent in which case they will be automatically restarted if they terminate. The node manager makes use of the proxy export facility provided by the trading service to provide dynamic services; it uses the factory service to create service instances, and the notification service is used to inform the node manager of the termination of any services previously created by the node manager.

The functions provided by the node manager can be split into three principal components; namely a database for describing services, the creation of services and the creation of dynamic services via the trader proxy export facility. Once a service has been created it is referred to as an activation.

A.12.1 Service Database

In order to create services it is first necessary to provide a means for describing such services; the node manager implements a persistent database for such descriptions. Each service description is identified by an alias string and each alias is unique within an individual node manager. Each service description consists of the following information:

- alias string uniquely identifying this service description
- maximum number of activations allowed for this alias
- trader interface name
- trader context name
- trader properties string
- template string
- arguments string
- environment string

In order to provide simple resource management the Node Manager allows the specification of a maximum number of activations for each alias. This limits the number of service objects which can be created for each alias and therefore also places an upper bound on the number of capsules which can be created. The section on Activation management below describes how this parameter may be used and also explains the policies applied by the Node Manager when the maximum number of activations is reached.

The trader interface, context and property strings are used to describe the service within the trading system. The template, argument and environment strings are passed to the factory service and contain sufficient information to create an instance of the service. The template string is the name of an executable file, the argument string represents the command line arguments to this executable and the environment is a sequence of variable assignments which can be accessed using the `getenv()` C library function.

Operations are provided for installing new descriptions and for deleting existing ones. The node manager will refuse to delete an alias if any activations of the service are still in existence. In order to avoid the race between removing an alias, and the creation of any new activations which

could cause such a removal to fail, it is possible to inhibit any further activations of the alias to be removed using the mask operation.

No operation is provided to modify an existing description since this can be easily implemented outside of the node manager by removing and reinstalling the description to be changed.

Operations are provided to list all of the installed aliases and for displaying the full description for a given alias. A complete listing of all of the service descriptions maintained by the node manager can be obtained by first requesting the list of installed aliases and then for each alias in the list requesting the full service description.

Finally all updates to the service description database are logged to stable storage as they are executed and read back from stable storage whenever the node manager is executed. In this way the database persists across successive invocations of the node manager and also across node crashes.

A.12.2 Activation Management

Each active instance of a service is called an activation; activations may be created by invoking the node manager's run operation which creates a new instance of a service given its alias name. Such activations may be designated as being persistent, in which case the node manager will automatically restart them should they terminate.

Activations may be destroyed by the kill operation; note that kill will terminate both persistent as well as non-persistent services. Given that an alias may have multiple activations it is necessary to provide a means of identifying a specific activation from the list of all activations for that alias. Therefore each activation is assigned an activation identifier which is unique within the context of that activation's alias; i.e. any activation can be uniquely identified by an alias name and an activation identifier. The kill operation therefore requires both an alias name and activation identifier as arguments.

When the maximum number of activations is reached, the Node Manager applies one of two policies depending on whether the activation request originates from a run operation or a trader lookup on a proxy export. In the first case, the Node Manager simply refuses to create any new activations and returns an appropriate error code from the run invocation. In the case of the trader lookup, the interface reference from an existing activation is used to satisfy the request; a round-robin policy is used to decide which activation's interface reference to return. It is possible to bypass this resource management by specifying the maximum number of activations to be zero, in which case no upper bound is place on the number of activations.

Operations are provided for listing all of the aliases which have at least one activation and for listing all of the activations associated with a particular alias. Again it is left to a client program to obtain the list of active aliases and then to obtain the list of activations for each alias in turn in order to display all of the activations currently in existence. The use of separate operations does introduce a race condition, however the effects of the race condition are not sufficiently harmful to warrant the complexity required to overcome them.

Creating an activation involves interacting with the factory service to instantiate a capsule and then interacting with the capsule to instantiate an object. The result of an object instantiation is an Object Identifier and a sequence of Interface References representing the service provided. Because a capsule may be capable of instantiating multiple types of objects the first argument in the argument string is conventionally used to denote the type of the object required. The current implementation of the Node Manager imposes a one-to-one mapping of objects to capsules, i.e. it will always create a new capsule when creating a new activation.

A.12.3 Proxy Export and Dynamic Services

The node manager makes use of the trading service's proxy export facility to provide dynamic service creation. Operations are provided to post proxy exports for a given alias and to remove a previously posted proxy export. Once posted, the node manager will receive any trader lookup operations for the service identified by the alias, and on receipt of such lookup requests it will either create a new activation or return an existing one as follows:

- if the maximum number of activations has been reached then cycle through all existing activations in a round-robin fashion.
- if the maximum number of activations has not been reached, or an unbounded number of activations has been specified, (i.e. a maximum of zero), then create a new activation.

In both cases the result of the lookup request is an interface reference which can be used to invoke the service instance created. As mentioned above the result of instantiating an object is a sequence of interface references, however the lookup request only returns a single interface reference, therefore a convention is used whereby the first interface reference in the sequence returned by an object instantiation is the one which can be returned by the lookup request and subsequently used for interacting with the service.

Activations created by the proxy export mechanism are treated in an identical fashion to those created directly using the node manger run operation. All activations may be listed and killed in the same way. Objects created dynamically will terminate of their own accord if they are idle for longer than some period of time.

B

Trader Constraint Language

The Trader constraint language consists of the following items:

- superlative functions: min, max
- comparative functions: $==, !=, >, >=, <, <=, in$
- constructors: and, or, not, $\rightarrow$ (is restricted by)
- property names
- numeric and string constants
- mathematical operators: $+, -, *, /$
- grouping operators: $(,), [,]$

The following precedence relations hold in the absence of parentheses, in the order lowest to highest:

- + and -
- * and /
- or
- and
- not

The comparative operator in checks for the inclusion of a particular string constant in the list which is the value of a property. The is restricted-by constructor (→) permits the client to specify that a superlative function should be applied to the set of instances which match the preceding portion of the constraint expression.

As a simple example consider the following:

1. The following QoS offers are made:
   - "Name TCP Rate 100"
   - "Name TCP Rate 1000"
   - "Name UDP"
2. The constraint expression "Name == TCP" would return:
   - "Name TCP Rate 100"
   - "Name TCP Rate 1000"
3. Alternatively "Name == TCP and → max [Rate]" would return:
   - Name TCP Rate 1000

The full BNF for the constraint language follows:

```
<program>     := <empty>
               | <expr>
               | <expr> -> <superlative>
               | -> <superlative>

<empty>       :=

<expr>        := <expr> or <expr>
               | <expr> and <expr>
               | not <expr>
               | ( <expr> )
               | <nexpr> in <nexpr>
               | <nexpr> == <nexpr>
               | <nexpr> != <nexpr>
               | <nexpr> < <nexpr>
               | <nexpr> <= <nexpr>
               | <nexpr> > <nexpr>
               | <nexpr> >= <nexpr>

<superlative> := min [ <nexpr> ]
               | max [ <nexpr ]

<nexpr>       := <term>
               | <nexpr> + <term>
               | <nexpr> - <term>

<term>        := <factor>
               | <term> * <factor>
               | <term> / <factor>
```

```
<factor>      := <identifier>
               | <constant>
               | ( <nexpr> )
               | - <factor>

<identifier>  := <letter><characters>

<letter>      := a | b | c | d | e | f | g | h | i | j | k | l | m
               | n | o | p | q | r | s | t | u | v | w | x | y | z
               | A | B | C | D | E | F | G | H | I | J | K | L | M
               | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<characters>  := <empty>
               | <character>
               | <characters><character>

<character>   := <letter>
               | <digit>

<digit>       := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<constant>    := <floatnumber>
               | <string>

<floatnumber> := <mantissa><exponent>

<string>      := '<chars>'

<mantissa>    := <digits>
               | <digits>.
               | <digits>.<digits>
               | .<digits>

<exponent>    := <empty>
               | <sign>e<digits>
               | <sign>E<digits>

<sign>        := <empty>
               | - | +

<digits>      := <digit>
               | <digits><digit>

<chars>       := <empty>
               | <char>
               | <chars><char>

<char>        := <letter>
               | <digit>
               | <other>

<other>       := ` | ~ | ! | @ | # | $ | % | ^ | & | * | (
               | ) | - | _ | = | + | [ | { | ] | } | ; | :
               | " | \\ | | | , | < | . | > | / | ?
```

Bibliography

The pages on which each reference is cited are listed in parentheses after the reference.

[Ades86] Stephen Ades, Roy Want, and Roger Calnan. *Protocols for Real Time Voice Communication on a Packet Local Network.* In Proceedings of the International Conference on Communications, pages 525–530, Toronto, June 1986. IEEE. (50)

[Ades87] Stephen Ades. *An Architecture For Integrated Services On The Local Area Network.* PhD thesis, University of Cambridge Computer Laboratory, September 1987. Technical Report No. 114. (29, 30)

[Aguilar86] L. Aguilar, J. J. Garcia-Luna-Acheves, D. Moran, E. J. Craighill, and R. Brungardt. *Architecture For A Multimedia Teleconferencing System.* ACM Computer Communications Review, 16(3):126–136, August 1986. (35)

[Anderson89] David P. Anderson, Shin-Yuan Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews. *Support for Continuous Media in the DASH System.* Report No. UCB/CSD 89/537, University of California at Berkeley, Berkeley, CA 94720, October 1989. (40)

[Anderson90a] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. *An Internet Protocol for Resource Reservation to Achieve Guaranteed Performance Communication.* Technical Report, International Computer Science Institute, Berkeley, CA 94720, February 1990. Technical Report. (40)

[Anderson90b] David P. Anderson, Ramesh Govindan, George Homsy, and Robert Wahbe. *Integrated Digital Continuous Media: a Framework Based on Mach, X11, and TCP.* Technical Report No. UCB/CSD 90/566, University of California at Berkeley, Berkeley, CA 94720, March 1990. (40)

[ANSA89a] ANSA. *ANSA: An Engineer's Introduction.* APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, November 1989. (4)

[ANSA89b] ANSA. *ANSA Reference Manual.* APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, March 1989. Release 01.00. (3, 4, 171)

[ANSA89c] ANSA. *Open Software Foundation Submission: Distributed Computing Environment.* CO.025.00, APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, October 1989. (171)

[ANSA90a] ANSA. *The ANSA Computational Model.* RC.205.1, APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, 1990. (84, 184)

[ANSA90b] ANSA. *ANSAware Implementation Manual.* APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, August 1990. (4,108)

[Arons89] Barry Arons, Carl Binding, Keith Lantz, and Chris Schmandt. *The VOX Audio Server.* In Second IEEE Comsoc Internatonal Multimedia Communications Workshop, Ottawa, Ontario, April 1989. IEEE. (38)

[Berglund86] E. J. Berglund. *An Introduction to the V-System.* IEEE Micro, pages 35–52, August 1986. (35)

[Birman87] K. Birman and T. Joseph. *Exploiting Virtual Synchrony in Distributed Systems.* ACM Operating Systems Review, 21(5):123–138, November 1987. SIGOPS SOSP 11. (38,159,162)

[Birrell84] Andrew. Birrell and Bruce. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions On Computer Systems, 2(1):39–59, February 1984. (18)

[Buxton85] William Buxton, Ralph Hill, and Peter Rowley. *Issues and Techniques in Touch-Sensitive Tablet Input.* In SIGGRAPH '85, pages 215–224, New York, July 1985. ACM. (60)

[Buxton86] William Buxton and Brad A. Myers. *A Study in Two-handed Input.* In Proceedings HCI'86 Conference on Human Factors in Computing Systems, pages 321–326, New York, April 1986. ACM. (60)

[Calnan87] Roger S. Calnan. *ISLAND: A Distributed Multimedia System.* In Globecom 1988, pages 744–749. IEEE, November 1987. (29,30)

[Calnan89] Roger Calnan. *The Integration of Voice within a Digital Network.* PhD thesis, University of Cambridge Computer Laboratory, February 1989. (29)

[Casner90a] S. Casner, K. Seo, W. Edmond, and C. Topolcic. *N-Way Conferencing with Packet Video.* In Third Internation Conference on Packet Video, Morristown, New Jersey, March 1990. (to appear). (23)

[Casner90b] Stephen Casner, Charles Lynn, Philippe Park, Kenneth Schroder, and Claudio Topolcic. *Experimental Internet Stream Protocol, Version 2 (ST-II)*, October 1990. RFC-1190. (23)

[CCITT-X.400] CCITT-X.400. *Message Handling Systems: System Model-Service Elements.* X400. (25)

[Chow90] C. H. Chow and M. Adachi. *Achieving Multimedia Communications on a Heterogeneous Network.* IEEE JSAC, 8(3):348–359, April 1990. (28)

[Christodoulakis86a] S. Christodoulakis, F. Ho, and M. Theodoridou. *The Multimedia Object Presentation Manager of MINOS: A Symmetric Approach.* In Proceedings ACM Sigmod, pages 295–310, Washington DC, May 1986. ACM. (26)

[Christodoulakis86b] S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria. *Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and a System.* ACM Transactions on Office Information Systems, 4(4):345–383, October 1986. (26)

[Clark88] D. Clark. *The Design Philosophy of the DARPA Internet Protocols.* In IEEE SIGCOMM, pages 106–114, August 1988. (16)

[Crowley89] T. Crowley and H. Forsdick. *MMConf: The Diamond Multimedia Conference System.* In Proceedings Groupware Technology Workshop, August 1989. IFIP Working Group 8.4. (36)

[Day83] J. D. Day and H. Zimmerman. *The OSI Reference Model.* Proceedings of the IEEE, 71(12):1334–1341, December 1983. (15,66)

[Ferrari90] D. Ferrari and D. C. Verma. *A Scheme for Real-Time Channel Establishment in Wide-Area Networks.* IEEE JSAC, 8(3):368–379, April 1990. (40)

[Forgie79] James W. Forgie. *ST - A Proposed Internet Stream Protocol.* IEN 119, September 1979. (23)

[Gerla84] M. Gerla and R. A. Pazos-Rangel. *Bandwidth Allocation and Routing in ISDN's.* IEEE Communications Magazine, 22(3):16–26, February 1984. (9)

[Gifford88] D. Gifford and N. Glasser. *Remote Pipes and Procedures for Efficient Distributed Communication.* ACM Transactions on Computer Systems, 6(3):258–283, August 1988. (18)

[Greaves90] David Greaves, Dimitris Lioupis, and Andy Hopper. *The Cambridge Backbone Ring.* In Infocom 1990, San Fransisco, 1990. IEEE. (22)

[Gusella83] Ricardo Gusella and Stefano Zatti. *TEMPO: Time Services for the Berkeley Local Network.* Report No. UCB/CSD 83/163, University of California, Berkeley, CA 94720, December 1983. (80)

[Hamilton84] K. G. Hamilton. *A Remote Procedure Call System.* PhD thesis, Cambridge University Computer Laboratory, December 1984. Technical Report No. 70. (18)

[Harita89] Bhaskar R. Harita and Ian M. Leslie. *Dynamic Bandwidth Management of Primary Rate ISDN to Support ATM Access.* In ACM SIGCOMM. ACM, September 1989. (23)

[Herman87] Gary Herman, Michael Ordun, Christine Riley, and Leland Woodbury. *The Modular Integrated Communications Environment (MICE): A System for Prototyping and Evaluating Communications Services.* Bell Communications Research, Morristown, New Jersey 07960, USA, March 1987. (27)

[Hill86] Ralph D. Hill. *Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction-The Sassafras UIMS.* ACM Transactions on Graphics, 5(3):179–210, July 1986. (60)

[Hodges89] Matthew E. Hodges, Russel M. Sasnett, and Mark S. Ackerman. *A Construction Set for Multimedia Applications.* IEEE Software, 6(1):37–43, January 1989. (32)

[Hopper86] Andy Hopper and Roger M. Needham. *The Cambridge Fast Ring Networking System.* Technical Report No. 90, University of Cambridge Computer Laboratory, June 1986. (22)

[Hopper90] Andy Hopper. *Pandora - an Experimental System for Multimedia Applications.* ACM Operating Systems Review, 24(2):19–34, April 1990. (32)

[Hudson87] Scott E. Hudson. *UIMS Support for Direct Manipulation.* Computer Graphics, 21(2):120–124, April 1987. (60)

[IFIP-WG6.579] IFIP-WG6.5. *Reports of the 2nd and 3rd Meetings of the System Environment Subgroup*, 1979. IFIP-WG6.5 N16, N17. (25)

[Karmouch90] A. Karmouch, L. Orozco-Barbosa, N. D. Georganas, and M. Goldberg. *A Multimedia Medical Communications System.* IEEE JSAC, 8(3):325–339, April 1990. (37)

[Lampson79] B. W. Lampson and R. F. Sproull. *An Open Operating System for a Single-User Machine.* In Proceedings of the Seventh Symposium on Operating System Principles, pages 98–105, Pacific Grove, California, December 1979. ACM. (66)

[Lantz86] Keith A. Lantz. *An Experiment in Integrated Multimedia Conferencing.* In Proc. CSCW '86: Conference on Computer Supported Cooperative Work, pages 267–275, December 1986. (35)

[Lantz87] Keith A. Lantz, Peter P. Tanner, Carl Binding, Kuan-Tsae Huang, and Andrew Dwelly. *Reference Models, Window Systems and Concurrency.* Computer Graphics, 21(2):87–97, April 1987. (35,60)

[Lazar85] Aurel A. Lazar, Avshalom Patir, Tatsuro Takahashi, and Magda El Zarki. *MAGNET: Columbia's Integrated Network Testbed.* IEEE JSAC, 3(6):859–871, November 1985. (24)

[Lazar86] Aurel A. Lazar, Mark A. Mays, and Kenichi Hori. *A Reference Model for Integrated Local Area Networks.* In Proceedings International Conference on Communications, pages 531–536, Toronto, June 1986. (25)

[Lazar87] Aurel A. Lazar and John S. White. *Packetized Video on MAGNET.* Optical Engineering, 26(7):596–602, July 1987. (24)

[Leslie83] I. M. Leslie. *Extending the Local Area Network.* PhD thesis, Cambridge University Computer Laboratory, February 1983. Technical Report No. 43. (14,59)

[Leung90] W. H. Leung, T. J. Baumgartner, Y. H. Hwang, M. J. Morgan, and S. C. Tu. *A Software Architecture for Workstations Supporting Multimedia Conferencing in Packet Switching Networks.* IEEE JSAC, 8(3):380–390, April 1990. (41)

[Lison89] H. Lison and T. Crowley. *Sight and Sound.* Unix Review, October 1989. (26)

[Mazumdar89] Subrata Mazumdar and Aurel L. Lazar. *Knowledge-Based Monitoring of Integrated Networks.* In First International Symposium on Integrated Network Management, Boston, MA., May 1989. (25)

[McAuley89] Derek McAuley. *Protocol Design for High Speed Networks.* PhD thesis, University of Cambridge Computer Laboratory, September 1989. Technical Report No. 186. (14,23,47,59,113)

[McAuley90] D. R. McAuley, December 1990. Private commmunication. (161)

[Mills89] Devid L. Mills. *Network Time Protocol (Version 2)*, September 1989. RPC-1119. (80)

[Naffah86] Najah Naffah and Ahmed Karmouch. *Agora-An Experiment in Multimedia Message Systems.* IEEE Computer, 19(5):56–66, May 1986. (25)

[Needham82] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System.* International Computer Science Series. Addison-Wesley, 1982. (18,29)

[Nicolaou90] C. Nicolaou. *An Architecture for Real-Time Multimedia Communication Systems.* IEEE JSAC, 8(3):391–400, April 1990. (69)

[Oguet90] F. Oguet, C. Schwartz, F. Kretz, and M. Quere. *RAVI, a Proposed Standard for the Exchange of Audio/Visual Interactive Applications.* IEEE JSAC, 8(3):428–436, April 1990. (17)

[Poggio85] A. Poggio, J. J. Garci Luna Acheves, E. J. Craighill, D. Moran, L. Aguilar, D. Worthington, and J. Hight. *CCWS: A Computer-Based, Multimedia Information System.* IEEE Computer, 18(10):92–103, October 1985. (35)

[Postel82a] J. Postel. *Internet Multimedia Mail Document Format Protocol,* March 1982. RFC-767-revised, MMM-12. (26)

[Postel82b] J. Postel. *Internet Multimedia Mail Transfer Protocol,* March 1982. RFC-759-revised, MMM-11. (26)

[Postel88] Jonathan B. Postel, Greogory G. Finn, Alan R. Katz, and Joyce K. Reynolds. *An Experimental Multimedia Mail System.* ACM Transactions on Office Information Systems, 6(1):63–81, January 1988. (25)

[Redman87] Brian E. Redman. *A User Programmable Telephone Switch.* Bell Communications Research, Morristown, New Jersey 07960, USA, May 1987. (27)

[Reed79] David P. Reed and Rajendra K. Kanodia. *Synchronization with Eventcounts and Sequencers.* Communications of the ACM, 22(2):115–123, February 1979. (145)

[Reynolds85] Joyce K. Reynolds, Jonathan B. Postel, Alan R. Katz, Greg G. Finn, and Annette L. DeSchon. *The DARPA Experimental Multimedia Mail System.* IEEE Computer, 18(10):82–89, October 1985. (25)

[Root86] Robert W. Root and Steve D. Hawley. *DynaMICE: A Direct Manipulation Graphics Interface for Controlling Advanced Telecommunications Services.* Bell Communications Research, Morristown, New Jersey 07960, USA, 1986. (27)

[Ross86] Floyd E. Ross. *FDDI - A Tutorial.* IEEE Communications Magazine, 24(5):10–17, May 1986. (21)

[Salmony89] M. Salmony and D. Shepherd. *Extending OSI to Support Synchronization Required by Multimedia Applications.* IBM ENC Technical Report No. 43.8904, April 1989. (24)

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. *End-To-End Arguments in System Design.* ACM Transactions on Computer Systems, 2(4):277–288, November 1984. (54)

[Sarin85] Sunil Sarin and Irene Grief. *Computer-Based Real-Time Conferencing Systems.* IEEE Computer, 18(10):33–45, October 1985. (34)

[Scheifler86] Robert W. Scheifler and Jim Gettys. *The X Window System.* ACM Transactions on Graphics, 5(2):79–109, April 1986. (159)

[Schmandt88] Chris Schmandt and Michael A. McKenna. *An Audio and Telephone Server for Multi-Media Workstations.* In Second IEEE Conference on Computer Workstations. IEEE, 1988. (31)

[Schmandt89a] Chris Schmandt and Barry Arons. *Desktop Audio.* Unix Review, 7(10), October 1989. (38)

[Schmandt89b] Chris Schmandt and Stephen Casner. *Phonetool: Integrating Telephones and Workstations.* In Globecom 1989. IEEE, November 1989. (27)

[Schneiderman83] Ben Schneiderman. *Direct Manipulation: A Step Beyond Programming Languages.* IEEE Computer, 16(8):57–69, August 1983. (60)

[Schroeder89] Michael D. Schroeder and Michael Burrows. *Performance of Firefly RPC.* ACM Operating Systems Review, 23(5):83–90, December 1989. SIGOPS SOSP 12. (18,149)

[Steinmetz90] R. Steinmetz. *Synchronisation Properties in Multimedia Systems.* IEEE JSAC, 8(3):401–412, April 1990. (53)

[Strathmeyer87] C. Strathmeyer. *Voice/Data Integration: An Applications Perspective.* IEEE Communications Magazine, 25(12):30–35, December 1987. (28)

[Swinehart83] D. C. Swinehart, L. C. Stewart, and S. M. Ornstein. *Adding Voice to an Office Computer Network.* In Globecom 1983. IEEE, November 1983. also available as Xerox Palo Alto Reseach Center Technical Report CSL-86-1, June 1986. (28,50)

[Swinehart87] Daniel C. Swinehart. *Telephone Management in the Etherphone System.* In Globecom 1987, pages 1176–1180. IEEE, November 1987. (28)

[Swinehart88] Daniel C. Swinehart. *System Support Requirements for Multi-media Workstations.* In Proceedings of SpeechTech '88 Conferenece, pages 82–83, New York, April 1988. Media Dimensions Inc. (28)

[Tanner87] Peter P. Tanner. *Mutli-Thread Input.* Computer Graphics, 21(2):142–144, April 1987. (60)

[Temple84] Steven Temple. *The Design of a Ring Communication Network.* PhD thesis, University of Cambridge Computer Laboratory, January 1984. Technical Report No. 52. (22)

[Temple89] Adam Temple, Aurel L. Lazar, and Rafael Gidron. *A Metropolitan Area Network Based on Asynchronous Time Sharing.* In IEEE International Conference on Communications, Boston, MA., May 1989. IEEE. (25)

[Tennenhouse87] D. L. Tennenhouse, I. M. Leslie, R. M. Needham, J. W. Burren, C. J. Adams, and C. S. Cooper. *Exploiting Wideband ISDN: The Unison Exchange.* In Infocom 1987, pages 1018–1026. IEEE, 1987. (23)

[Tennenhouse89a] David L. Tennenhouse. *Layered Multiplexing Considered Harmful.* In Protocols for High Speed Networks, IBM Zurich Research Lab., May 1989. IFIP WG6.1/6.4 Workshop. (57)

[Tennenhouse89b] David L Tennenhouse. *Site Interconnection and the Exchange Architecture.* PhD thesis, University of Cambridge Computer Laboratory, September 1989. Technical Report No. 184. (23,59)

[Terry88] D. Terry and D. Swinehart. *Managing Stored Voice in the Etherphone System.* ACM Transactions on Computer Systems, 6(1):48–61, February 1988. (28)

[Thomas85] Robert H. Thomas and Harry C. Forsdick et al. *Diamond A Multimedia Message System Built Upon a Distributed Architecture.* IEEE Computer, 18(11):1–31, November 1985. (26)

[van Renesse88] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. *Performance of the World's Fastest Distributed Operating System.* Operating Systems Review, 22(4):24–34, October 1988. (149)

[Want88] Roy Want. *Reliable Management of Voice in a Distributed System.* PhD thesis, University of Cambridge Computer Laboratory, July 1988. Technical Report No. 114. (29)

[Wilkes79] M. V. Wilkes and D. J. Wheeler. *The Cambridge Digital Communication Ring.* In Local Area Communications Network Symposium, Boston, May 1979. (29)