An architecture for multimedia communication and real-time collaboration

by B. K. Aldred
G. W. Bonsall
H. S. Lambert
H. D. Mitchell

The paper presents the requirements for real-time multimedia communication in a collaborative environment and describes how the requirements can be met through the IBM Lakes architecture. The initiatives of the ITU-T through the T.120 series of recommendations are described, and the interoperability of Lakes with these recommendations is discussed.

Computers attached to networks are commonly used for collaborative activity; this is supported by a variety of mail, messaging, and database products. In most cases these applications are either person-to-person information exchanges where the parties are working together, but not simultaneously, or are person-to-machine interactions. The first category is illustrated by mail applications, where electronic documents are processed first by one party and then by the other; although the turnaround time can be very short, the essence of the application is alternate activity. The second category involves only one person directly; therefore there is no concept of a natural human dialog to be sustained.

In contrast, real-time collaboration has two essential elements: people are directly involved with each other, and simultaneous activity by these people is the essence of the interaction. Examples include desktop conferencing, distance learning, help desk operation, remote presentations, brainstorming, and shared document editing. Real-time collaboration requires multimedia communication, because the traditional data exchange between workstations needs to be enhanced with audio to allow conversation among the participants for effective human interaction. An alternative, equally valid perspective is to consider the exchanges as data enhancements to telephony. In addition to data and audio, live video can be justified for some, but not all, applications. In real-time collaborative activity, natural interaction between people requires low-latency transmission so that responses are not noticeably delayed. This aspect is much more demanding than is normal in existing messaging, mail, and related networked applications. Support for the audio and video streams also demands isochronous communications to prevent distortion, and this is not commonly available in data networks.

A current topic for research and development activity is the efficient provision of appropriate multimedia communication services. Much of the ef-

©**Copyright** 1995 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8670/95/$3.00 © 1995 IBM

fort has focused either on the human factors of computer-supported collaborative work or on the associated computing science aspects.[1-3] Other activities have explored the social and organizational implications.[4,5] Rather less academic interest has been directed toward the design and specification

New technologies based on ATM combine isochronous capabilities with high bandwidth.

of generic application programming interfaces, although this has been a topic for computing and telephony organizations. Many individual and collective developments are underway, although little has been published for competitive reasons.

From a standards viewpoint, the initiative by the International Telecommunication Union-Telecommunication Standardization Sector (ITU-T) through the draft recommendations of the various T.120 committees is of considerable interest, with the development of the Multipoint Communication Service (MCS)[6] and the associated Generic Conference Call (GCC).[7] Previous to this work the H.320 recommendation provided the basis for audio and video communication over the integrated services digital network (ISDN) and allowed the development of videoconferencing services. The requirements met by the H.320 recommendation were for an endpoint attached to the network to exchange a single audio stream, and optionally a single video stream, with other users; multipoint operation was being provided by multipoint control units within the network itself. Compatibility with the existing equipment was of primary importance and this dictated call setup and the audio formats. Also recognized in the H.320 recommendation was the need for data communication such as file transfer and document exchange, and therefore provision for multiplexing was included for data.

The H.320 recommendation has been widely adopted for videoconferencing and provides interoperability between different manufacturers' equipment; furthermore it has encouraged the deployment by network operators of standard multipoint control units for multiway conferencing. However, the absence of any definition of the contents of the data channel has precluded interoperability for data services and this deficiency has given rise to the T.120 series of recommendations. Hence MCS provides multiple logical data channels to a user and allows multiparty operation, thus laying a foundation for collaborative data services. GCC builds on MCS to provide call management for conference setup and tear-down, while other T-series recommendations define application protocols for file transfer and shared whiteboards, with others to be added in the future.

The combination of the H.320 and T.120 recommendations will be important in allowing interoperability between desktop conferencing users over public switched networks. However, the real-time collaborative opportunity is much greater than the scope of H.320 and T.120 combined; some examples will illustrate the problems still to be addressed.

From a personal computer (PC) as opposed to a telephony perspective, audio and video support is rich and varied. Compact disk standards provide the basis of PC audio, and many video technologies are already in use, with quality approaching and moving beyond that of television. The G.711 audio and H.261 video recommendations of H.320 are not always acceptable substitutes, and the loss in quality will meet resistance from customers whose expectations are set by domestic television. PC networks are characterized by extreme diversity and do not normally have the isochronous capabilities of telephony networks; although their bandwidths are typically orders of magnitude greater, their high latency and jitter are problematic. New technologies, such as those based on asynchronous transfer mode (ATM), address these deficiencies and combine isochronous capabilities with high bandwidth. Such capacity allows video compression without loss of information, combining high quality with low latency.

Simple hierarchical topologies as envisaged by the authors of the ITU-T recommendations are not typical of installed corporate computer networks; complex meshes exist and the constituent links have widely varying, and often unpredictable, characteristics. Frequently the challenge is to exploit what exists while allowing new capabilities

to be selectively integrated into the infrastructure. Multiple audio and video streams per user are required for some applications; for example, reviewing of television commercials, and synchronization between multiple audio, video, and data streams is desirable. An architecture that classifies streams into audio, video, or data fails to recognize that the true distinction is to be made on the basis of the communications requirements, or quality of service needed, for the stream, independent of its content.

From an application perspective, the requirement is not normally for multimedia communications but for distributed multimedia device connectivity. With few exceptions, most multimedia streams originate from devices, and the need is to transport the output of a source to one or more remote destination devices and provide end-to-end, device-to-device services. Many of the problems are to be found at the end points, where the devices are coupled to communication networks using the shared services of an operating system, processor, and bus.

Today the telephone is the natural choice for real-time collaboration and no viable alternative exists for most people; as demand increases for existing telephone calls to be enriched with data exchanges, and ultimately with video services, the personal computer is destined to become the instrument for personal communications. This can be seen either as an evolution of the telephone, or as an evolution of the personal computer, and although the result is the same in both cases, the process is very different. Equally interesting is the nature of the network to which the multimedia personal computer is attached, which can be the telephony network enhanced to supply the necessary bandwidth, or a computing data network enhanced to support low-latency isochronous communication.

This section introduced the need for real-time collaboration using multimedia communications. Following sections clarify the requirements and describe the IBM Lakes architecture designed to meet these requirements.

The real-time collaboration requirements

The detailed requirements for a real-time multimedia communications platform arise from three sources: the needs of the application programmer, the nature of the networks that form the collaborative infrastructure, and the demands of users and product developers. Requirements from each source follow.

From an application programmer perspective, it is preferable if applications can be developed in a platform-independent manner, thus encouraging portable code that can be migrated between systems. Also, applications should be independent of the physical communications network and thus able to perform correctly when changes occur in that network. In other words, applications should be programmed in terms of the logical, rather than the physical, properties of a network, typically expressed in quality-of-service parameters such as throughput, latency, and jitter.

The environment should support applications simultaneously interacting with one or more other applications in either an independent or a dependent way. As an illustration, an audio application may wish to handle multiple simultaneous calls, where each call involves one or more other audio applications.

The interaction between applications should be based upon peer-to-peer responses since there can be no certainty that a central application will be available to mediate behavior. Peer-to-peer design in a multiuser environment allows client/server support to be implemented as a special case, because a client can assume the responsibility for providing common services.

The applications should be able either to handle all aspects of application-to-application collaboration themselves, or to delegate functions to others, for example, the association and disassociation phases between applications. Such delegation simplifies the provision of a consistent user interface for call control, yet allows individual applications to specialize in particular collaborative functions, for example, in shared text editing.

Applications should be able to collaborate freely, without restricting themselves to particular partners. Such a desirable objective is inherently unachievable but flexibility can be included through careful design, for example, by the use of self-describing data streams whose format and content description are available independently of the data.

Any applications already written and installed should be able to be used collaboratively without

modification. This requirement can normally be only partially satisfied, but nevertheless useful functions can be offered. Thus an existing single-user application can rarely be made fully multiuser without access to the source code, but its inputs and outputs can usually be made accessible to remote users.

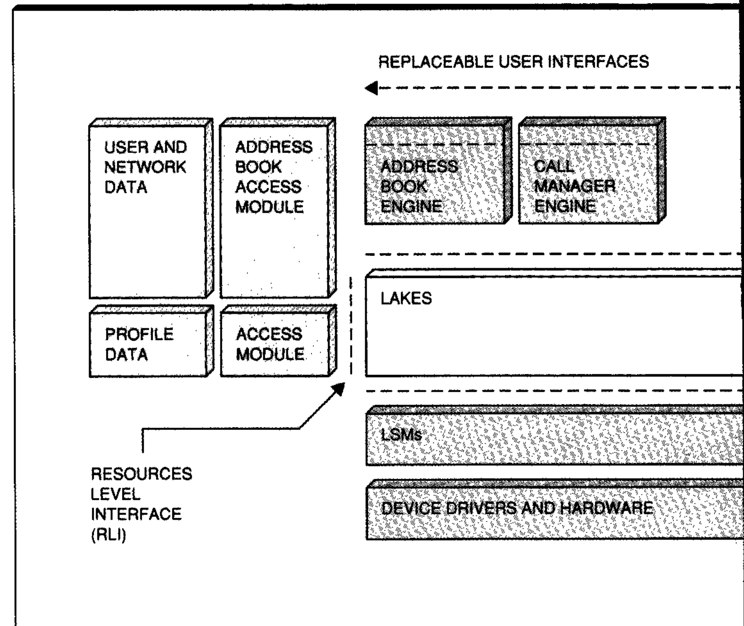
Other requirements arise from a networking perspective. Both digital and analog networks should be supported. Analog audio and video capabilities are frequently available at lower cost than their digital counterparts, although this is slowly changing. Ideally, applications should be unaffected by network migration from an analog to a digital communications base. Multiparty communications are also required and this capability needs to be constructed out of the various underlying transport networks available and described through entries in address books or directories.

Network traffic in collaborative situations is increasingly multimedia in nature, containing not just coded data but one or more audio and video streams. This requires a smooth handling of these continuous data flows, with low latency to ensure good usability. A telephony system model, where data flows are directed and connected, is more suited than one in which the applications themselves move data between links and devices. Collaboration frequently involves both computer and telephone networks and devices, and the distinction between these is of little interest to users and should therefore be hidden. Intelligent networks can be exploited, allowing functions such as data serialization to be removed from applications and implemented more efficiently within the network itself.

A third set of requirements emerges from the needs of customers, users, and product developers. Standards compliance, for example with H.320 and T.120, is mandatory to ensure guaranteed interoperability. An open specification is required, with published interfaces and protocols sufficiently detailed to allow independent implementations or extensions.

This paper describes the IBM Lakes architecture, which is intended to meet the above requirements.[8,9] It is based on experience gained with the IBM Person-to-Person* desktop conferencing product[10,11] and studies designed to extend the range of applicability into the broader fields of real-time collaboration.

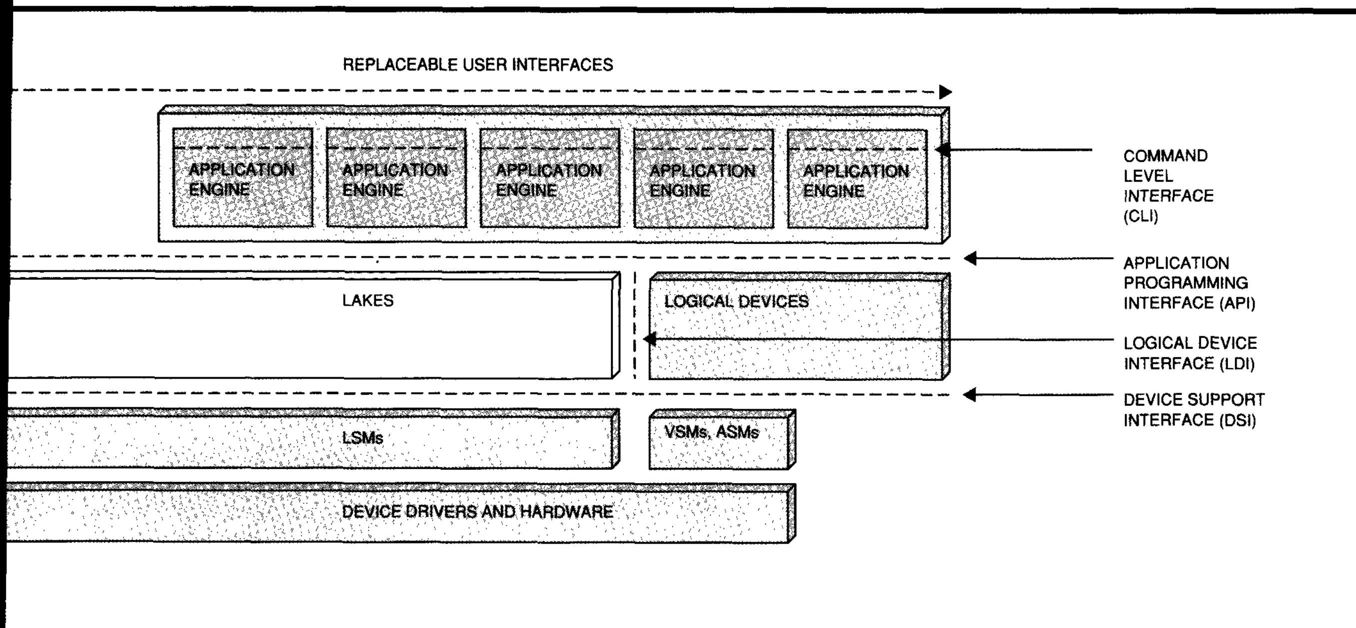
Figure 1 Principal Lakes interfaces



A key decision in the provision of generic programming platform support for collaboration is the stance taken with regard to call models.[12-15] A call model captures the rules of behavior that are to be enforced between the collaborating parties. Some questions that a call model can answer are:

- How is a call established?
- What equipment is to be interconnected?
- What facilities are required to be available?
- Who is in control?
- How is it known who is in control?
- How is a call extended?
- How does a call end?
- How are the actions of users made known?

Call models in desktop conferencing products are based on those in conventional telephony. It is possible for the application programming interface to reflect one or more of these call models, or even to allow models to be tailored to suit user preferences; alternatively the call model can be regarded as being above the interface and imposed by one or more applications. Lakes takes this latter approach because of the greater flexibility provided. The Lakes application programming interface is based only on the assumption that a network of



computers exists and that application instances either are running at these computers or may be caused to run. The model therefore is a distributed application-based model where the applications dominate and determine behavior. Concepts of calls and other abstractions are built on top of this distributed model and accessed through higher level interfaces.

The overall structure provided by Lakes is shown in Figure 1, with the key interfaces identified.

Four important open programming interfaces exist: the *application programming interface* (API), which allows applications to request Lakes services; the *command level interface* (CLI), which allows applications to be controlled via commands; the *device support interface* (DSI), which is provided to allow support to be added for different software and hardware subsystems, such as communication systems, video and audio devices, etc.; and the *resources level interface* (RLI), through which Lakes can be integrated into existing network and user databases containing phone book and related information.

Other interfaces and components illustrated in Figure 1 are described later.

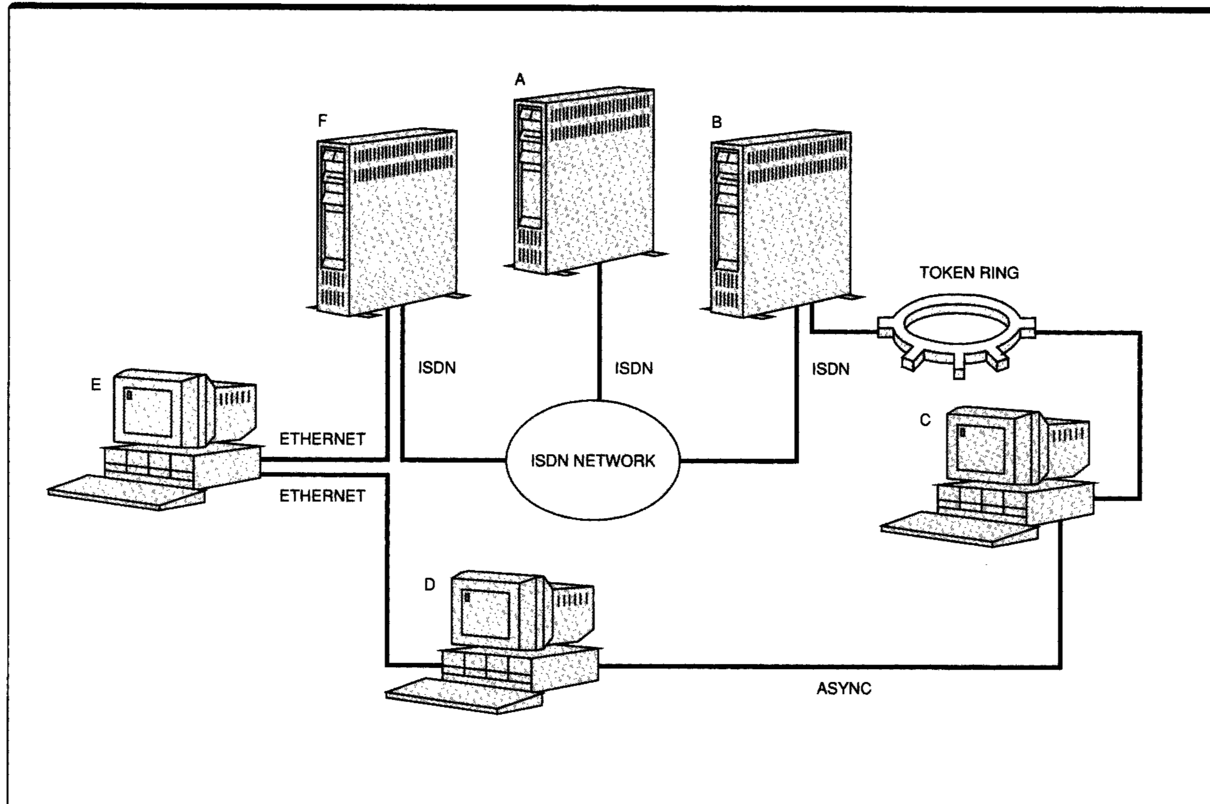
Principal components of the architecture

This section introduces the individual components of the Lakes architecture and describes how they interact to meet the requirements of multimedia communication and real-time collaboration.

Network, nodes, and applications. At the highest level, the Lakes programming model presents a representation of a network to an application as a series of interconnected nodes. A *node* is the addressable entity in Lakes representing a user. It comprises an instance of the Lakes API and typically a set of resources, such as application programs, data, devices, etc. Usually a node is a dedicated programmable workstation capable of communicating with its peers; in a multiuser system a node is associated with each user. Nodes are identified by name; ideally all node names are unique but duplicates can be tolerated as long as their associated nodes are never required to intercommunicate. The node-naming scheme is not prescribed by the architecture, but a system such as that defined by the IBM Open Blueprint* directory services has many benefits.

A collection of intercommunicating Lakes nodes is called a *Lakes network*. It is fundamental to the

Figure 2 A possible Lakes network



architecture that any node, independently of any others, can dynamically join or leave the network. The structure of the physical network itself is not presented to applications and cannot be deduced. It is a necessary condition that all nodes can be physically interconnected, but no connection pattern is prescribed or excluded. Thus multiple connections between nodes can exist and will be exploited where possible, and the indirect access to one node via other intermediate nodes is possible. The links between nodes can vary in type, capability, and protocol. An example of such a Lakes network is shown in Figure 2.

In the figure, workstations F, A, and B all have ISDN capability and can connect in a pair-wise fashion using the public ISDN network. B and C are also equipped with token-ring adapters and can communicate with each other over the local area network. C and D have asynchronous adapters and a dedicated link between their ports. F, E, and D form an Ethernet local area network. Lakes applications on all nodes will have logically direct access to all other nodes. Lakes itself may realize a connection from, say D to A, either by using E and F or by using C and B; the choice will be governed by the specific quality-of-service requirements, the state of the network, and by the customization, or *profile*, data accessed through the resources level interface and the profile access module as shown in Figure 1.

Although Lakes nodes are generally workstations running applications, it is sometimes convenient to regard other equipment as a Lakes node without normal capabilities. A Lakes implementation at one node can allow certain attached devices such as telephones or video telephones to behave as though they are attached Lakes nodes. It provides this by simulating a *virtual application* at a *virtual node* to represent the device. The advantage of this approach is that an appropriate application pro-

gram written to communicate with real Lakes nodes will now also communicate with simulated nodes without application device-specific code. From a user perspective, telephones and video telephones can thus participate in desktop conferencing and other applications, with a Lakes node performing the role of a media server, mixing audio and selecting the video for viewing.

In order for Lakes to be fully active at a node, one particular application must be running at that node. This application plays a unique role and is known as the *call manager*. The name is misleading and suggests that the job of this application is to provide call management facilities. From a purely Lakes standpoint this is not the case; the job of the call manager is to respond to certain events that are generated by Lakes and to supply installation, application, or user-generated information. The call manager is involved in resource management and gives permission for applications to reserve communications bandwidth. Many call managers may be available for execution at a particular node, but by definition, only one instance can perform this role at any time. This is in no way restrictive, since call manager responsibility can be transferred from one application to another. Alternatively, it is possible for the call manager role to be combined with an application function, if appropriate. The limit of one active call manager at a time does not limit the number of call models that can be simultaneously in use.

In responding to Lakes events, the call manager is controlling the behavior of Lakes and is therefore, in some sense at least, implementing policies and thereby one or more call models. This does not mean that the call manager needs to establish a dialogue with the workstation user and be the application that establishes and terminates calls. Other applications may do this, and may also enforce additional rules of behavior and build on the base created by the call manager. Although all this is possible, simplicity and consistency generally require that only one application, which may be the call manager, provides the user interface and allows the setting-up and tearing-down of calls and the launching and termination of applications within those calls. Other applications may provide user functions within this framework.

A Lakes implementation may request that the resources of one node be made available for Lakes communication between two otherwise unconnected nodes; this is termed *passive operation* and involves the call manager, since permission must be granted at the passive node for this to take place.

The most fundamental concept in Lakes is that of the *application sharing set*. This describes a collection of applications that have agreed to collaborate. Before joining an application sharing set, applications can reference each other only by name; once they become a member of an application sharing set, each has direct addressability to the others in that set. A further consequence of membership is that existing members are informed, through events generated by Lakes, of any arrivals and departures. Application sharing sets are identified by name, and any application can be a member of any number of such sets. It will be seen that set membership is key to many of the other facilities offered by Lakes for application collaboration.

One way in which applications can join a sharing set is by initiating a share request using the LakShareApp call and naming an application sharing set, a target application, and a destination node. This request is first passed by Lakes to the call manager at the sending node, which will typically transfer it to the call manager at the destination node. Usually this second call manager will launch the requested application and transfer the share request to it, and if successful, the source application will be informed. The participation of the call managers in this process allows local control of the sharing process. It also allows other actions to be initiated if necessary. These are, for example, user authentication procedures or the delegation of share requests to another node. The call managers also play a vital role in resolving the names used by applications to identify other nodes and applications. The symbolic name references are translated into specific node identities and applications. The sharing mechanism can be cascaded; for example, if two applications are already sharing, one of them can initiate a share with a third application specifying the same sharing set, with the result that all three applications are then sharing with one another.

Applications may also make local share requests on behalf of other applications. It is this ability that can be exploited to create applications that handle call management on behalf of other applications: a third party can put the first and second party into communication with each other. Facilities exist for

either the issuer or the target of the share request to name the application sharing set.

Information on nodes in the Lakes network is assumed to be held in an *address book* that Lakes accesses through the resources-level interface and the replaceable address book access module shown in Figure 1. The address book may optionally contain a variety of node-related information. The API call LakAddressBookFind allows an application to search the address book and retrieve parameters, for example, node names of interest.

An application initially requests addressability to a remote node by using the node name in a Lakes API call. This name is first passed to the local call manager, which has the option to modify it. The resultant name is then used by Lakes to determine connectivity information. This requires access to the externally-held network and user database, using the facilities of the Lakes resources interface. A *node handle* is returned to the application to reflect this resolution of the node name. Addressability from one application to another requires the resolution of an *application name*. If both applications are local to a node, then the resolution involves the call manager at that node; if one application is remote, then both call managers are involved. The resolution results in the target application being identified to the source application by an *application handle*. Subsequent application requests using handles require no name resolution and are transferred by Lakes directly to the target application.

Channels and ports. Lakes provides two distinct mechanisms for application-to-application communication. The simplest form uses *signals* and is intended for the exchange of commands, control information, and text strings. Two forms of application signaling are supported; the LakSignalApp request provides unidirectional communication, and LakSignalAppWithReply supports a response. Neither of these requests is restricted to members of the same application sharing set. This signaling mechanism has many uses. Among them is communication between components of a single application; for example, a processing component (engine in Figure 1) and a user interface component of the same application might exchange information via signals. Such a design enables applications to share a common processing component.

More sophisticated intercommunication is supported between applications in an application sharing set. This requires the establishment of data communication links between the applications. These logical links are known as channels. *Channels* are logically dedicated, unidirectional pipes, with application-specified transmission characteristics. Unidirectional channels are used as the basic communications building blocks to efficiently support two-way communications, since the quality-of-service requirements are often different in each direction. Thus in a broadcast application, no return flow is required; in a movie-on-demand application, only simple control data are required to select the movie; in telephony, full capabilities are required in both directions.

Lakes channels are always defined by the sending application and go from it to a receiving application. This approach is used because only the sending application can be aware of the properties of the data, which dictate how they should be transmitted. The ends of channels are known as *ports*; thus each channel has one sending port and one receiving port. A *sending port* sends data *blocks* down the channel; a *receiving port* receives data blocks in the order in which they were sent down the channel. Both sending and receiving ports can be shared between different channels. There may not be a direct mapping between the logical channel structure seen by the Lakes applications and the physical communication network in existence between the nodes; the mapping that does exist is not identifiable by the applications. A complex example of channels, ports, and applications is shown in Figure 3.

An application is expected to establish multiple channels to another application as a convenient way to separate data traffic of different types. Thus if an application wishes to send audio, video, and image application data, it would be normal for the application to create three such channels. The characteristics of each of these channels would be specified differently to suit the intended traffic. If the destination application wished to send data back, then it would have to establish another set of channels to move data in the opposite direction. Lakes may map some or all of the logical channels onto a single physical link, but this will be invisible to the application. The interface to physical communication links is provided through one or more link support modules (LSMs) as shown in Figure 1; information in the address book identifies which LSMs

provide access to a particular node and supplies the connection details.

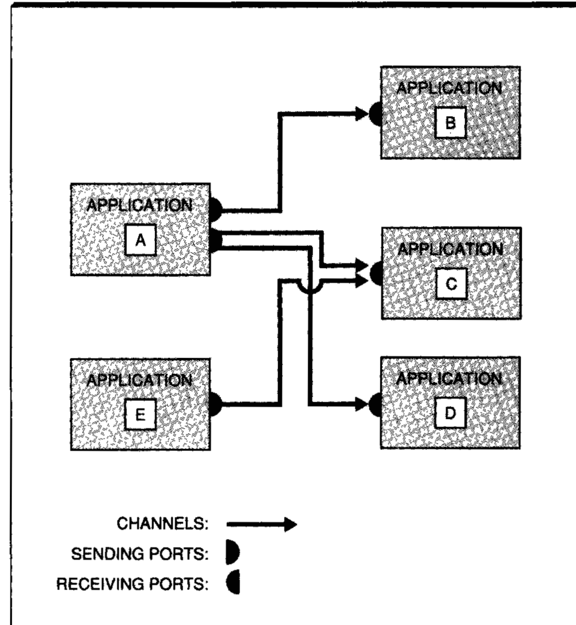
The capabilities of a channel are governed by its *quality-of-service* (QOS) characteristics, initially negotiated with Lakes during the creation process, which allow data transmission capabilities to be tailored to the requirements of the expected traffic. Such parameters are defined according to the *signal type*, which distinguishes analog from digital data. Typically for digital data, QOS is defined in terms of parameters such as throughput, latency, and jitter. Rather than expressing these parameters in the applications directly, a symbolic reference to collections of values, known as the *data class*, should be used, thus allowing the values to be changed without affecting the applications. Such a data class might be, for example, "G.711 Audio" for an ITU-T G.711-compliant audio stream, with the translation into detailed communications characteristics defined elsewhere.

Channel characteristics can be renegotiated after channel creation. Channel QOS may also be left undefined; this allows channels to be created whose operational characteristics depend upon the resources available when data are being sent down the channel. This reflects the reality of many existing data networks where any particular QOS cannot be guaranteed but is wholly dependent upon the behavior of the other simultaneous users; token-ring and Ethernet local area networks exhibit this form of behavior.

A feature of Lakes is that channels may be collected into named sets; these sets are known as *channel sets* and each must also be associated with a channel set type. Four such types exist: standard, merged, serialized, and synchronous. The names assigned to channel sets are local to an application sharing set; thus duplicates may exist among sharing sets.

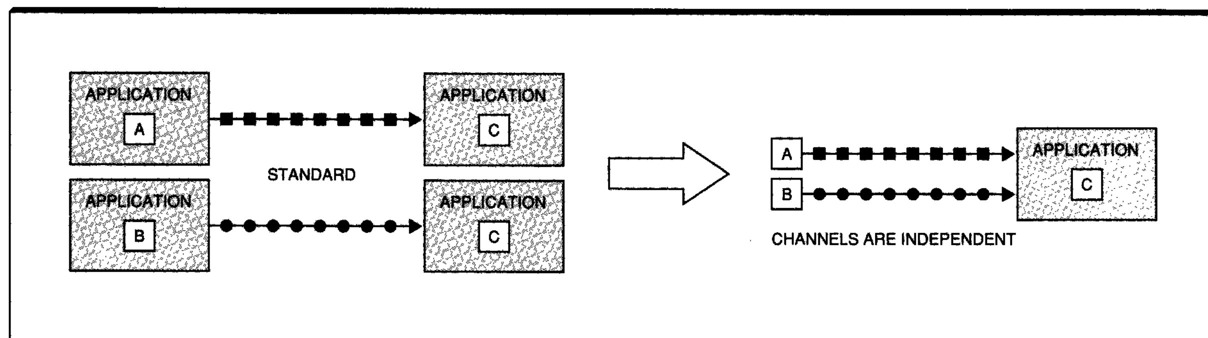
Standard channel sets provide a convenient way of referring to a collection of channels, for example for program channel management. This grouping does not change the individual behavior of the constituent channels in any way. Thus, in Figure 4, application A is sending squares to application C while application B is sending circles. At C two ports exist: one terminates the channel established by A and from it C receives squares; the other terminates the channel established by B and from it C receives circles.

Figure 3 Channels and ports



The *merged channel set* groups a collection of channels together in a way that, for any single application, the sending ports and receiving ports for all channels in that set are each combined into a single such port. A typical use of the facility is illustrated by an application that sends data to multiple destinations; this can be achieved by making all the channels to those destinations members of a merged channel set; then the data presented by the application at the single sending port will go to all destinations. More importantly, an application can often arrange for channel creation in a merged channel set to be handled automatically by Lakes, and so, as receiving applications come and go, the data are sent to all the correct receiving ports without further programming. This feature makes merged channel sets particularly attractive. For example, in a collaborative chat application it is sensible for each application to establish a channel to itself; down the channel it sends its own user's contribution, and writes to the screen the information it receives back from the channel. As the application is shared with others it will receive their contributions as well as its own; as others leave, their contributions will disappear. Figure 5 illustrates two examples of the merged channel set; the first is identical to the example used earlier for

Figure 4 A standard channel set



a standard channel set, where applications A and B are sending squares and circles respectively to application C. The specification of a merged channel set means that the receiving port at C is used for the second channel, and thus both circles and squares are available at C. The identity of the sender is not lost through the use of a merged channel set; each data block is presented with the originator identified.

The second example in Figure 5 is more complicated in that A and B are sending data respectively to C and D through the same merged channel set. This is interpreted to mean that all the senders of the data are attempting to send to all the receivers; thus both C and D receive all the blocks sent by A and B. There is no attempt to interleave the data in an identical way, and therefore, although C and D receive all the data and they both receive data from A and B in the order in which the data were sent, it may be that the interleaving of the blocks is seen differently by C and D.

The *serialized channel set* is closely related to the merged channel set and performs the same functions. It has the additional property that the interleaving of data from the various sources is performed identically for all receivers. If serialization could be implemented without affecting QOS, then the serialized channel set could always be used in place of the merged channel set. However, since this is not the case, it is necessary to offer both capabilities to applications. A measure of the quality of the implementation of data serialization is the size of the added delay as determined against a universal clock and the fidelity of the data ordering with respect to its actual sequence. Implementation of serialization need not involve sending the data to a common serialization process; instead tokens can be used to represent data blocks. The tokens can be serialized and the order returned to the delivery mechanism. Algorithms have also been developed[16–18] to distribute the serialization process, and the selection of nodes to be used can have dramatic implications on performance. The Lakes architecture does not dictate the algorithms to be used; this is an implementation decision. Figure 6 illustrates a serialized channel set.

The fourth channel set type, the *synchronized channel set*, exists to allow applications to specify that data blocks on multiple, otherwise independent, channels are to be tied together in time and therefore delivered together, but through the individual ports belonging to their respective channels. A number of implementation strategies exist; the data blocks from the constituent channels can be tagged with identifiers and then either interleaved down a single communications link or transmitted through independent links and then sorted and sequenced. Normally synchronized channels are used for multimedia streams such as audio and video; in these cases the QOS characteristics will have been set to ensure low latency and jitter. The synchronization performance is therefore established by the QOS for the constituent channels. Figure 7 illustrates a synchronized channel set.

Channels, and their associated ports, can be created explicitly or implicitly. Explicit creation uses the LakCreateChannel request, specifying the required channel characteristics; likewise new channels can be added to an existing port through the

Figure 5 A merged channel set

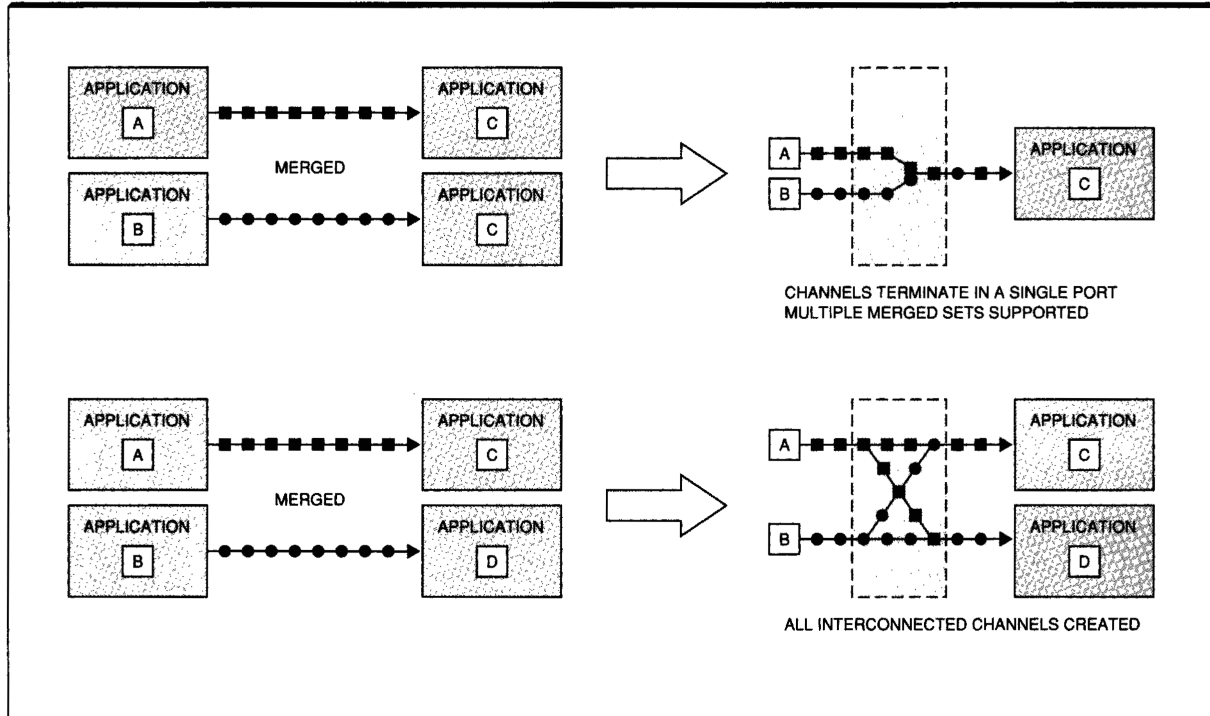
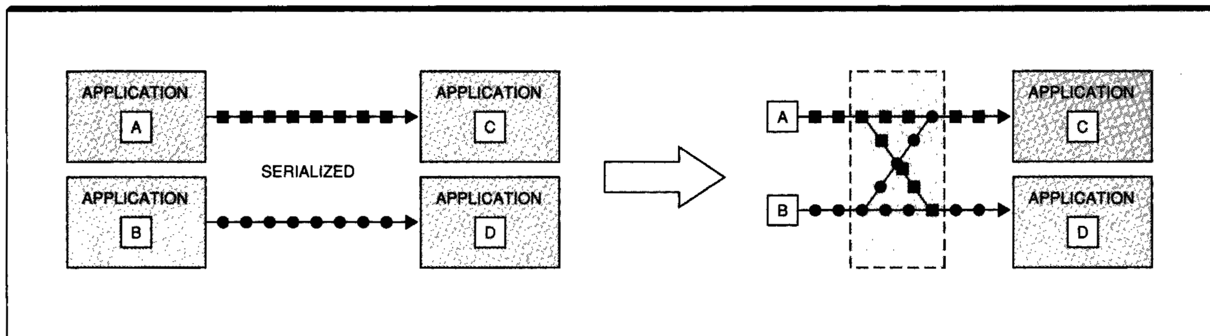


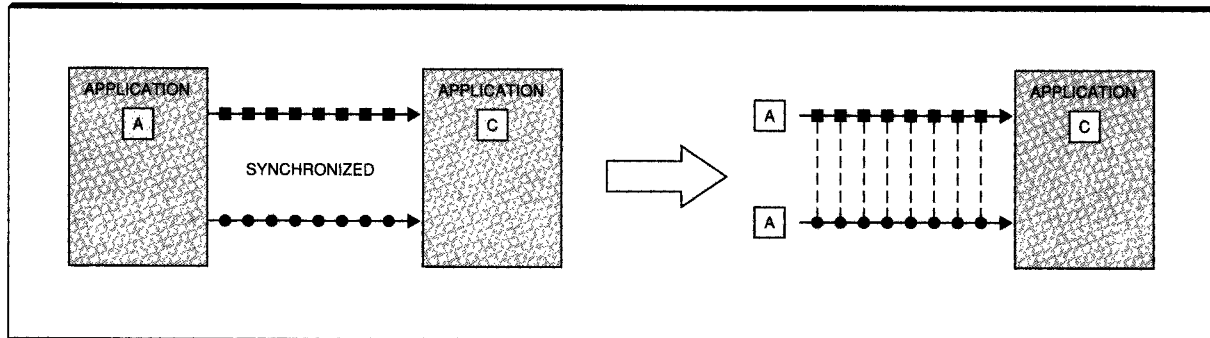
Figure 6 A serialized channel set



LakAddChannel request. This mechanism also allows a port to be shared across channels belonging to different channel sets; for example, data can be sent from a single port to one set of destinations belonging to a merged channel set and to a second set of destinations belonging to a serialized channel set.

Channels can be implicitly created as a consequence of an application being, or becoming, a member of an application sharing set. For example, if unshared applications already have a merged or serialized channel, and the channel set name used is identical across these applications, then when the applications share with each other, the

Figure 7 A synchronized channel set



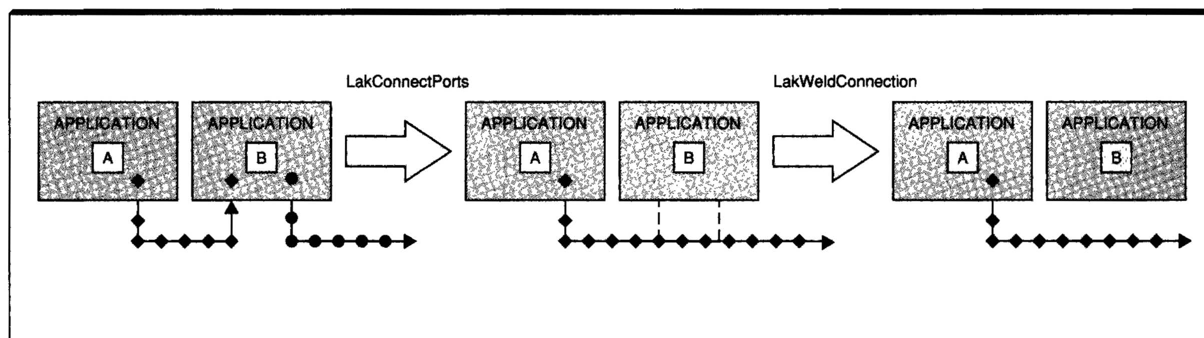
additional channels required will be created automatically. The result obtained is independent of whether the share precedes or follows the explicit channel creation. Applications are notified of channels implicitly created in this way and have the same ability to accept or reject them that they have for explicitly created channels. A channel, however created, can be deleted through the LakRemoveChannel request; the channel to be deleted is uniquely identified by specifying both its sending and receiving ports. Certain QOS characteristics of channels, such as throughput, can be changed after channel creation through use of the LakChangeChannel request.

The sending and receiving ports that terminate channels have an assigned *connect type*—event, command, or null—and are associated with a *port event handler*. *Event ports* generate an event when data are either available or required; *command ports* allow the application to drive the receipt or supply of data to the port via explicit commands. *Null ports* are ports that are unable to supply data to an application. An example is the sending port, associated with an analog channel, of a video camera. A port can be controlled through commands specified in a LakSignalPort request. Signal commands are issued to the local port and can be passed to the next port or the end port in the channel or channel set. Normally the signal commands for channel ports will be sent to the port event handlers of the applications supplying or receiving data, and may be used to stop, start, decrease, or increase the data flow. The order of signals between a source and a target is maintained. Signal commands sent to receiving ports in a serialized channel set are serialized with the data, so that all destinations receive the same sequence of commands and data. A receiving port can cause the sending port to stop sending data down the channel by issuing a LakSuspendPort request, with the option to either discard or deliver the remaining data in the channel. Suspended data transmission can be restarted by a LakResumePort request.

User exits can optionally be associated with ports. These allow monitoring or manipulation of the data, either after the data have been supplied to a sending port or before they have been presented by a receiving port. In the case of synchronized channels, synchronization is carried out from the time the data leave the sending port user exit to the time data are presented to the receiving port user exit.

Ports are associated with a *data class* that specifies the *data type* and *data subtype* that are sent by a sending port down the channel or received by a receiving port. The data type identifies the nature of the data, for example, audio, video, file, etc. Data type also distinguishes analog data from digital data. The data types themselves are further divided into subtypes according to the precise format of the data; examples of audio subtypes are G.711, G.721, G.722, etc. The list of recognized data types and subtypes is accessed by each Lakes node from a body of customization information available to that node. The data class may be queried by an application independently of the data stream itself; this facility assists application interaction. The data subtype may be specified differently at the sending and receiving ports of a channel, with Lakes performing the conversions below the API. Certain characteristics of ports, such as

Figure 8 Port connection and welding



data class, event handler address, user exit, and user information, can be changed after port creation through the LakChangePort request.

Ports can be *connected* together, via the LakConnectPorts request, to establish channel-to-channel communication, so that an application may redirect its inputs to another application for processing. When ports are connected, if no user exits have been established, no further application involvement in the data flow is required, but it is allowed, for example, with LakSignalPort. Connected ports allow the *streaming* of data between applications, between devices, and between an application and a device. Connected ports can be subsequently disconnected using the LakDisconnectPorts or may also be *welded* using LakWeldConnection, so that the connection is permanent and persists even when the local application has terminated. Welding removes any user exits that may be present on the connected ports. The resulting channel behaves as though it had been originally created to run from its source directly to its destination, and may be physically reestablished if appropriate. Figure 8 illustrates port connection and welding.

Connection and welding of channels allows the transport of data to drop below the API. Lakes has the option, in some cases, of effecting the connection either at a very low level at that node or rerouting the flow away from that node.

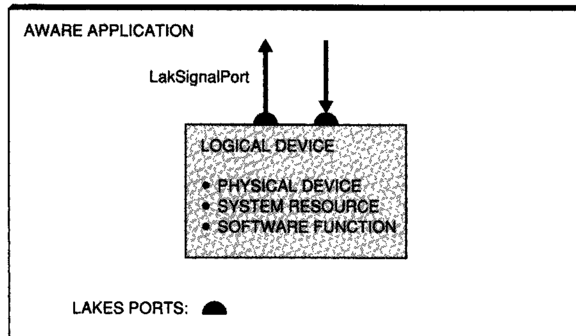
Negotiation of channel quality of service. Applications communicating through channels have many requirements for both QOS and bandwidth negotiation and control. Accordingly Lakes provides mechanisms to allow applications to: request unspecified QOS and be given whatever is dynamically available, request a particular QOS, and manage their own communications resources. Another mechanism allows the call manager to manage communications resources across applications.

For those applications that have unspecified QOS characteristics, channels may be established using the LakCreateChannel request, leaving the QOS parameters empty. Throughput and other characteristics will not be guaranteed by Lakes and may change dynamically, depending upon availability. Such channels will take from whatever bandwidth is currently available; this includes taking from bandwidth that may have been reserved but is not actually in use.

Certain applications have fixed QOS requirements for the channels that they need to communicate with other applications. In these cases each channel may be established directly, using the LakCreateChannel request. The resources are allocated by Lakes if they are available. If the channel characteristics are changed or the channel is deleted, any freed resources are returned to Lakes.

Some applications are more flexible in their QOS requirements and need to determine whether an acceptable range is available to a particular node before creating the channel. Although this can be accomplished through the LakQueryResource request by specifying the target node, unfortunately if used in this way, the call returns only what is dynamically available, and there is no guarantee of availability when a subsequent LakCreateChannel is issued. One way to avoid this situation is to issue only the LakCreateChannel to attempt to es-

Figure 9 A Lakes logical device



tablish an appropriate channel and to reissue the request with a lower QOS specification if the attempt fails. Lakes attempts to return an indication of the available QOS when a channel create fails.

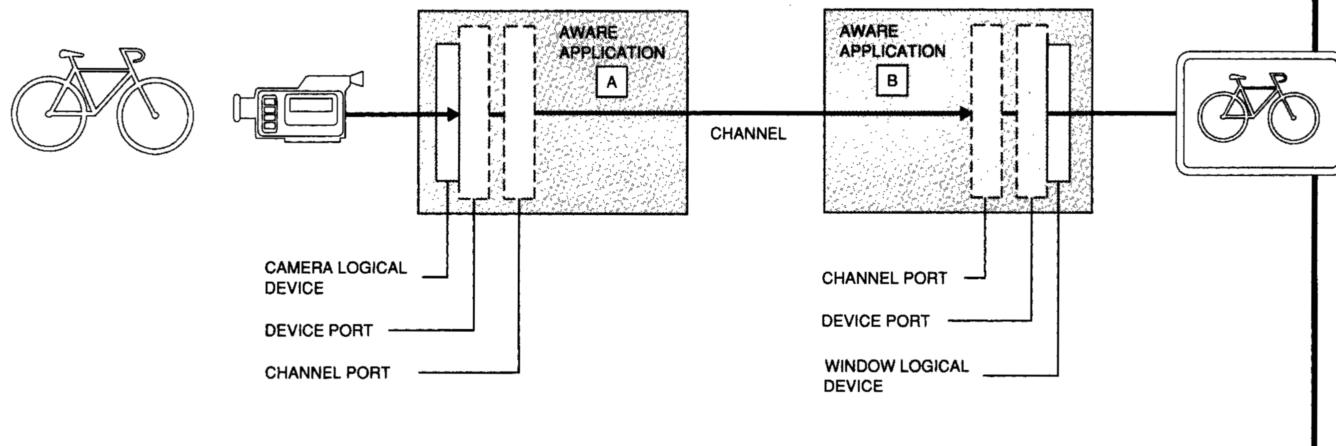
Other applications have flexible QOS requirements but need the ability to manipulate the allocation of communications resources. For example, a desktop conferencing application may wish to use all the bandwidth available all the time and allocate it to a combination of video and audio channels; however, when data services are required, it may wish to temporarily reallocate some video bandwidth to data, restoring it to video use later. If the channels are requested directly from Lakes, there is no guarantee that released bandwidth will be available to that application later. This problem is overcome by the concept of named *resource sets*. Applications can request that communications resources to a remote node be allocated to an application-owned resource set; such resources are identified by the appropriate QOS parameters. Channels are then allocated from this set and any resources freed are returned to the resource set. Only the call manager can request resources directly from Lakes; other applications must request resources from those acquired by the call manager, either directly or indirectly, from other applications. This gives the call manager the ability to manage competing requests for resources between applications. This need arises when multiple independent applications are involved in a desktop conferencing session, for example, chat, chalkboard, video, and audio applications. Each of these is unaware of the others, but all request bandwidth to the target node. The call manager can monitor their resource requests and allocate resources effectively. In situations of this kind the call manager will typically have acquired the resources to the remote node before a LakShareApp request has been issued to applications at that node, so that the resources are reserved prior to the activation of the collaborating applications. The alternative approach, where resources are requested after the collaboration has been started, is also permissible.

In some cases, the call manager needs to negotiate with applications to determine what resources are available and to defer any reallocation until the negotiations are complete. For example, a movie application starting part way through a call may require a certain minimum bandwidth that is not available. Other applications may be already running that, between them, are using more than that bandwidth and can be flexible in their usage. Lakes provides requests that enable the call manager to ask applications to prepare to free the bandwidth and later to actually free it, if sufficient total bandwidth is available.

Logical devices. Channels, as described previously, provide application-to-application communication but do not address the problems of coupling data sources and sinks, which are normally devices such as video adapters, audio adapters, and others. In some cases obvious physical entities are not involved, but windows, clipboards, and other software elements play a similar role. Lakes provides device access through a notion of *logical devices*, which are abstractions of these entities. Logical devices are independent of the specific details of the particular devices and provide program independence. Lakes does not determine the number or nature of the logical devices but presents a prototype API through which all may be controlled, or device data supplied or accessed. In Figure 1, these calls are illustrated as an extension of the Lakes API. The logical device interface (LDI) provides the mechanism through which such logical devices are integrated into a Lakes implementation. A *logical device* can be exploited in Lakes, not only to allow easier access to system resources and devices, but also to allow end-to-end data streaming. One use of a logical device is to provide access to the inputs and outputs of non-Lakes applications and thus to allow such applications to be used collaboratively.

A logical device can be represented as shown in Figure 9.

Figure 10 Connecting a Lakes logical device to a channel



Logical devices are identified by name. When *opened* with the LakOpenDevicePort request, they present a port to the application; a single logical device can have multiple such ports, and a device can simultaneously present ports to different applications at the same node. The LakOpenDevicePort request allows characteristics to be established peculiar to that device, for example, the data formats to be used. Opened logical devices can be controlled through commands, specific to the particular logical device, sent via the LakSignalPort request.

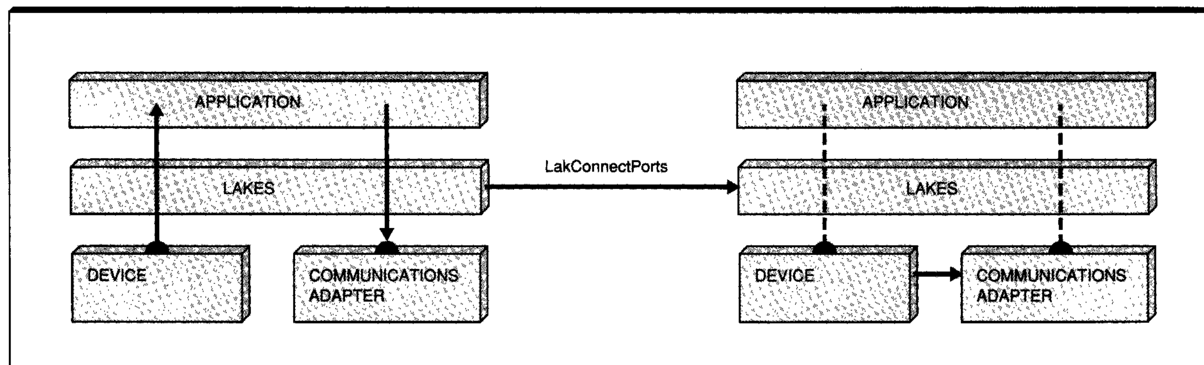
Applications can connect a port on a logical device to a channel port; this enables data to flow to or from the device and across the channel. This data flow does not require further application involvement once the connection has been made. This is illustrated in Figure 10.

Data from the camera are streamed through the camera logical device, across the channel, and then displayed by the window logical device. The application can control the two logical devices via the LakSignalPort call; when the transmission is no longer required the application can disconnect the ports, close the devices, and remove the channel. Figure 11 illustrates the data paths involved in this example at the sending node; without port connection, device data are sent to the application from a device port, and from the application to the channel with Lakes, using the full communications stack, labeled LAKES. With port connection, low-level paths between the device and the communications adapter can be exploited, such as multivendor independent protocol connectors or bus mastering, with the application still retaining control to the device and channel ports.

Certain logical devices present standard interfaces so that existing drivers can be more easily accommodated; the audio and video logical devices exploit this with a concept of replaceable audio and video support modules (ASMs and VSMs in Figure 1), thus enabling the standard logical devices to be interfaced to existing audio and video subsystems.

Tokens. Collaborative activity frequently requires that resources owned by a node, for example, a printer device, be shared with other nodes. Such resources are considered to be global, and application access to them is controlled through *global tokens*. Other resources are local to application sharing sets, for example, a shared pointer, and access to these resources is managed through *application tokens*. A token owner determines the significance of a token and allocates it on request. At the discretion of the owner, queued requests may be permitted, and more than one concurrent holder of a particular token may be allowed. Token owners can optionally force holders to hand back tokens.

Figure 11 Data flows on port connection



Global tokens share a common name space throughout the Lakes network, but since applications are expected to know the identity of the node holding the required globally available resource, duplicate global token names are permitted. Facilities for the broadcasting of global token availability information are not provided; instead, the call manager at the node with the global resource is responsible for resource management and therefore owns any global tokens. Such tokens may be held by an application on an exclusive or shared basis; token ownership cannot be transferred via Lakes. Requests for a global token may be queued, with the queue being held above the API and managed by the owning-node call manager. Access to global tokens is not restricted to a sharing application set.

Application token name space is restricted to the application sharing set. A token may be owned by a member application, and ownership can be transferred. Application tokens may be held exclusively or shared. Requests for tokens may be queued, with the queue held above the API and managed by the current application token owner.

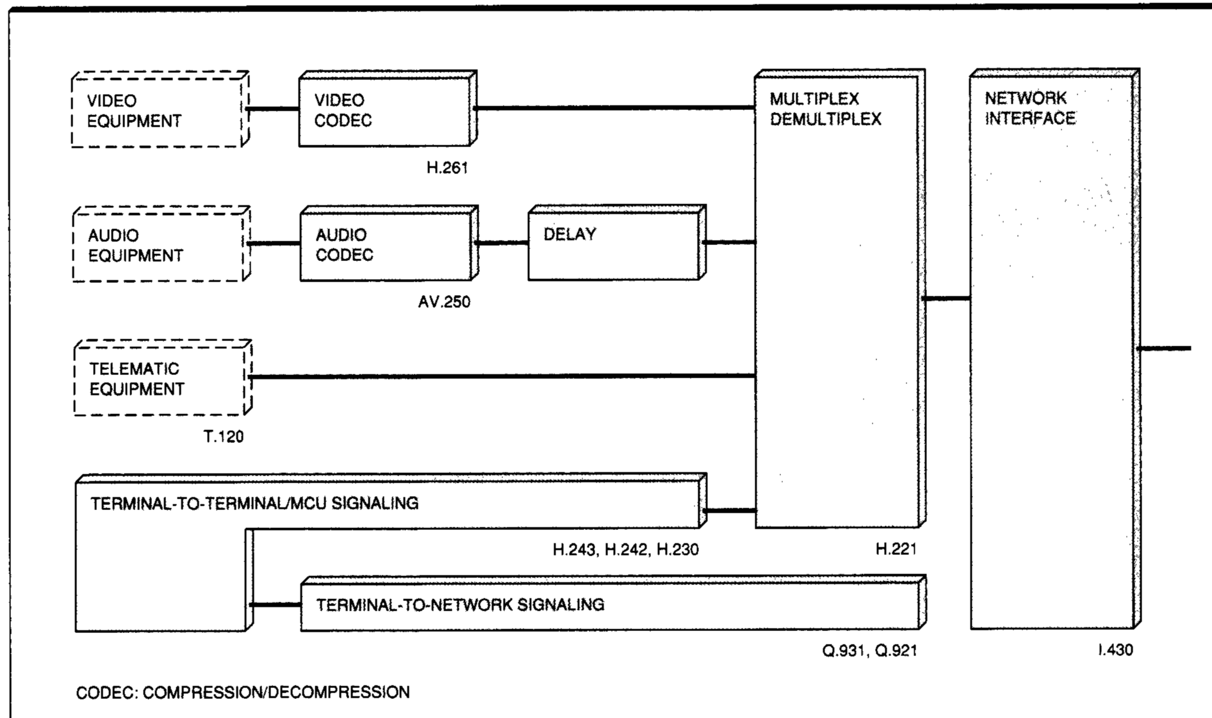
System considerations

Whereas the previous section described individual components and interactions, this section describes services defined by the Lakes architecture that apply across an entire system.

Monitoring, audit, and control. The Lakes architecture allows applications to monitor requests and events at a Lakes node. A LakBeginMonitor request starts monitoring; specific classes of activity can be set, and within these classes, individual requests and events selected. Control of one application by another is accomplished through *audit managers* and is provided to allow access, costing, and billing policies to be implemented. At any node a Lakes application may become an audit manager by issuing the appropriate API call. Two kinds of audit managers are supported: local and remote. Both receive information on application activity and may be required to approve certain actions; additionally local audit managers give permission for their node to be remotely audited. Depending upon the scope, almost all activity by an audited application raises either an information event or an authorization event, first at the local audit manager(s), then at the remote audit manager(s). Information events such as removing a channel do not require audit manager approval; authorization events require the approval of all the audit managers.

Intelligent networks. Lakes allows the exploitation of many of the additional services provided by current and future intelligent networks, because it offloads functions from the application such as data cloning, data merging, data conversions, data routing, data serialization, and data synchronization. Although these functions can be implemented at the local node, in some cases they can be more efficiently implemented within the network itself. For example, a simple implementation of data serialization requires that all the data in the serialized channel set be sent to a single point, then cloned and distributed to the receiving nodes. If this function is to be performed at the application level in

Figure 12 The ITU-T H.320 recommendations



a non-Lakes implementation, a decision must be made between the collaborating instances as to which node is selected to implement the function. It is desirable that this node have efficient communications access to the other nodes and the necessary resources available. Most application programmers are not interested in these details, and an arbitrary node is likely to be selected and to remain fixed. From a Lakes application perspective, without serialized channel support, the task would have been even more difficult. The application could not determine the nature of the network and therefore would have no basis on which to make an informed choice. In fact, the optimal location for the serialization process may be a node in the network that is not even part of the collaborating set but has the necessary resources and links. A sophisticated implementation of Lakes, by removing this function from applications and making its existence explicit through the API, can cooperate with other instances across the network to determine the implementation details.

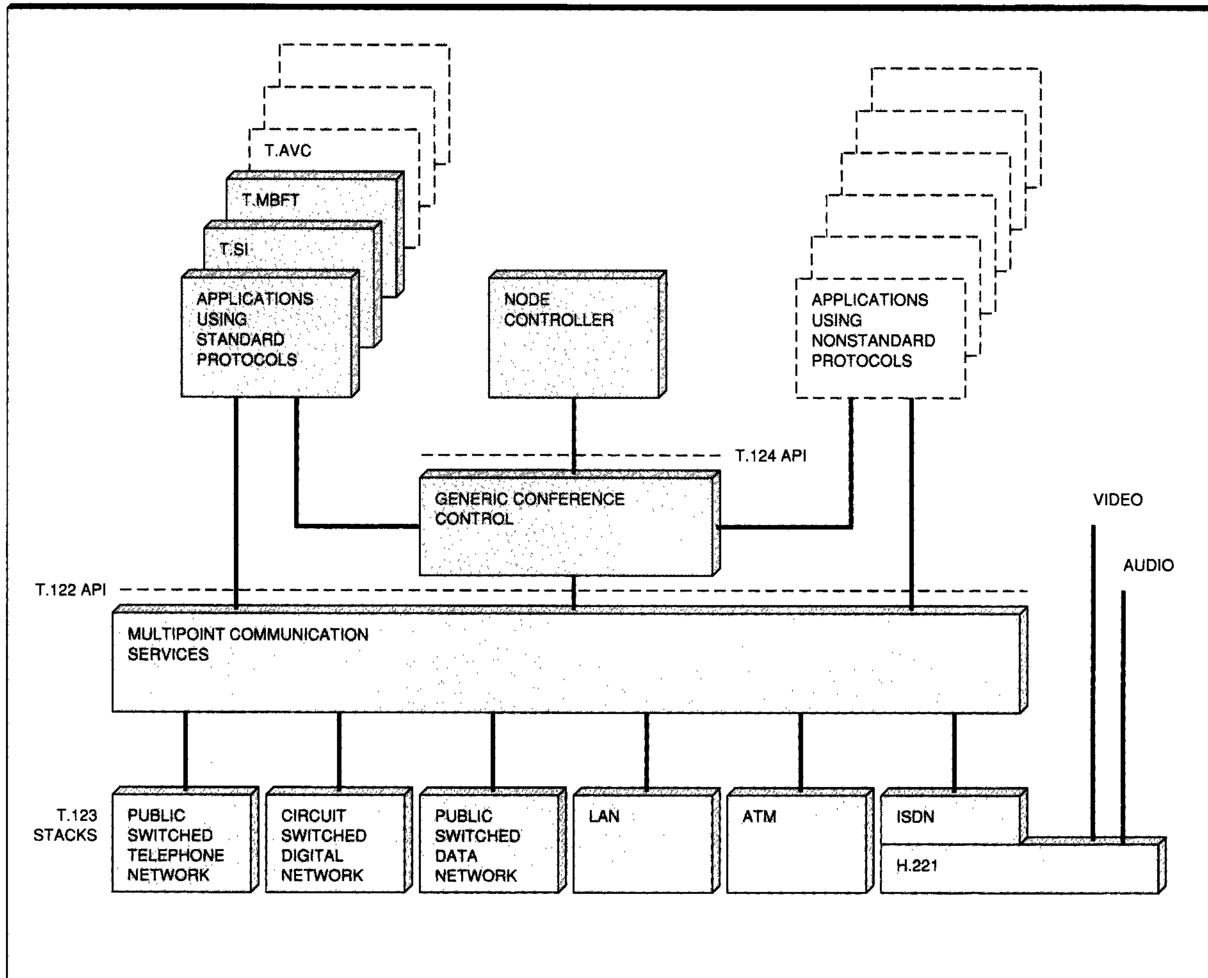
Data cloning provides another example in which the underlying network is more able to determine an efficient implementation than any particular node. When data must be sent to multiple destinations across a heterogeneous network, any one application can send the data directly to each target. In practice, with knowledge of the network and access to intermediate nodes, it would be more efficient to replicate the data at points in the network and execute a tree-structured transmission scheme.

Interoperability with ITU-T H.320 and T.122 recommendations

Because of the importance of the H.320 and T.122 recommendations, they are more fully discussed in this section, which also includes several scenarios illustrating how Lakes supports these recommendations and how it interoperates with an MCS network.

The H.320 recommendation. The H.320 recommendation provides for visual telephony services over an isochronous digital network, for bandwidths up to 1920 kilobits per second. An overview of the

Figure 13 The ITU-T T.120 recommendations



H.320 family of recommendations is shown in Figure 12.

H.320 assumes that three types of information, video, audio, and data, need to be combined and then transmitted. Each of these data types is encoded in a unique way, combined together with the necessary control information, and then sent over the network. The key elements of the H.320 recommendation are

- *Video content*: Encoding is defined by the H.261 recommendation.
- *Audio content*: Encoding is defined by AV.250, a reference to a series of recommendations including G.711, G.722, and G.728. Audio is delayed for synchronization purposes since compression and decompression for video takes longer than for audio.
- *Data content*: This is not defined by H.320 but is specified in the later T.120 series of recommendations.
- *Multiplexing of audio, video, and data*: H.221 partitions the available communications bandwidth to support multiple logical streams.
- *System control and signaling*: Control of the network by the terminal, for example, establishing the call, is transmitted over the D channel for ISDN according to I.400. Control information between the connected terminal equipment is de-

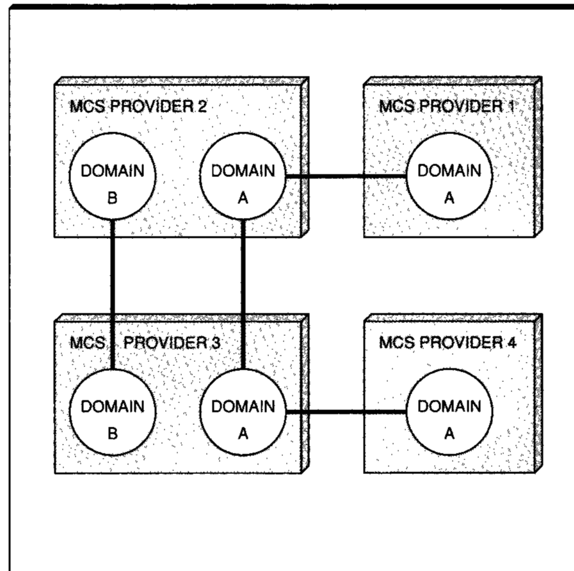
fined in H.320 and uses facilities available in recommendations H.243, H.242, H.230, and H.221. An example is the selection of an audio encoding scheme.

The T.120 recommendation. The T.120 recommendation is designed to support what is described as "audiovisual conferencing services, where a conference refers to a group of geographically dispersed nodes that are joined together and that are capable of exchanging audiographic and audiovisual information across various communication networks." It refers to a collection of subsidiary recommendations, as shown in Figure 13. Although a wide variety of data transports are supported, it is only for digital telephony, using ISDN or switched 56 kilobits-per-second services with the associated H.320 recommendation, that audio and video are supported. Extensions to T.120, to both exploit ATM networks and support the full range of multimedia traffic, are under development.

T.120 requires that the stack from the T.122 API layer and below be used in a compliant conference. Conference control is provided by GCC and this component can be driven by the user, through an application known as the node controller. All applications use MCS for communication and may also issue commands to the GCC conference control component.

Multipoint communication service. The multipoint communication service (MCS) is a generic service designed to support highly-interactive multimedia conferencing applications. The basic concept is that of a *domain* established over a collection of point-to-point MCS connections; within the domain, application clients can send and receive data. MCS connections are therefore logical connections defined across a physical connection; one physical connection can support more than one logical connection. To create an MCS domain, an MCS implementation at a workstation, known as an *MCS provider*, establishes a connection to a remote MCS provider and binds this connection to a domain. Other MCS providers can then establish connections to these providers and bind to the same domain; alternatively MCS providers already with connections to a domain can bind further connections to that domain. In this way a domain can be established, which can vary in complexity from a simple point-to-point connection to a multi-branched tree structure of connections. Figure 14 illustrates the relationship between MCS providers and domains.

Figure 14 MCS providers and domains

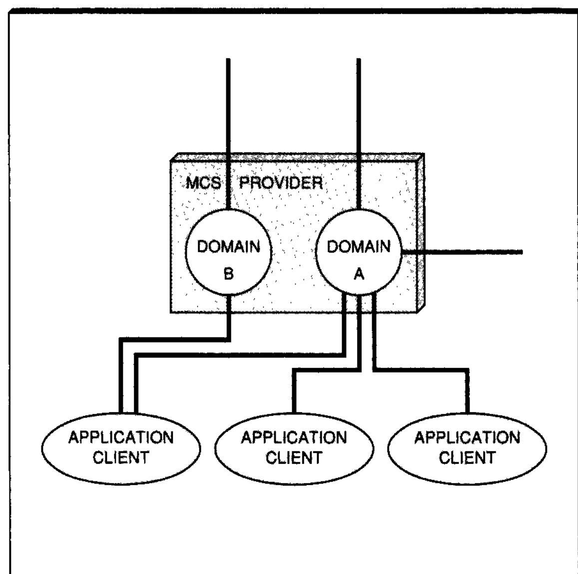


Existing domains can be combined into a single domain. A restriction on the establishment of a domain is that it be created hierarchically; this is enforced by ensuring that, as each binding takes place, the called provider is explicitly positioned as either hierarchically superior or subordinate to the calling provider.

Application instances (known as *clients* or *users*) can attach to domains through their local MCS provider. In this way an application instance can be attached to one or more domains. Figure 15 illustrates the relationship between application clients and domains.

Once domain setup and client attachment is complete, before data can be exchanged clients must join the right combination of interaction *channels*. Channels are effectively domain-wide addresses for the delivery of data to all, or a subset, of the clients of a domain. A client does not have to be joined to a channel in order to send data to it, but must be joined to it in order to receive data from it. Channels are classified as *static* or *dynamic* and have a numeric identifier; dynamic channels are

Figure 15 Application clients and domains



further subdivided into user, private, or assigned channels.

Static channels are permanently available and any clients can join at will. They provide a mechanism for clients to send data to any subset or all of the clients. Thus, for example, the clients in a domain may have a convention to use channel 10 to broadcast to all members; to achieve this they all join channel 10 and send on this channel data to be received by everyone. Particular uses are allocated to individual static channels in other standards, for example, in the GCC and the T-series applications recommendations. *Dynamic channels* require creation before they exist. A *user channel* is created by the original user domain attachment request and can only be joined by the client with that identifier; this therefore provides a mechanism for any of the other clients to send data on a point-to-point basis to that client. *Private channels* are created by a client, who becomes the owner of that channel and is the only user initially joined to it. The owner can invite other clients to join or force clients to leave. Having been invited, other clients can send data on the private channel or join it to receive data from it. *Assigned channels* provide a mechanism for allocating an empty channel to a client; like static channels they can be joined at will, but for these channels the identifier is allocated by MCS, thus ensuring that initially the requesting client is the only client joined to that channel. Clients that join channels may subsequently leave them; similarly MCS domain providers can force clients to leave channels.

Once the participating MCS providers have connected and bound to the domain, and clients have attached and joined the right combination of channels, then the clients are ready to exchange data. Two data services provide the actual transfer of data. *MCS-send-data* sends data packets on the specified channel to all clients, except the sender, joined to that channel. The data packets from any client are delivered in sequence, but the packets from multiple clients may be interleaved in different orders at each receiving client, a consequence of their being delivered using the most direct route up and down the tree of MCS providers. A variant of send data is *MCS-send-data-with-responses*, which requires each receiving client to generate a short response, with the responses from all receivers gathered at each level and traveling back along the distribution path of the original message. *MCS-uniformly-sequenced-data-send* parallels the send data request, but guarantees identical interleaving of data packets. The packets to be serialized are sent to the top MCS provider, and from there are dispatched, in the same sequence, to all receiving clients and also to the sender. Data transmission is associated with a *priority* specification with four normally available: top, high, medium, and low. These priorities are used to resolve the contention issues that may arise during transmission. Associated with each priority are bandwidth and latency specifications.

In addition to data transmission facilities, MCS also provides *token management* services, which may be used by clients to control resources.

Interoperability aspects. Support of the standards described is important for the reasons given earlier. There are three primary cases to be considered:

1. An attached Lakes node is using H.320 standards for its own communication to another Lakes node.
2. A non-Lakes H.320-compliant terminal, such as a video telephone, is attached to a Lakes node and sending audio and video, where:
   a. Lakes applications are unaware that the terminal is not another attached Lakes node.

b. Lakes applications are controlling the remote terminal directly as an attached device.
3. A non-Lakes T.122-compliant device or workstation is sending audio, video, or data to a Lakes node.

Case 1 is important because it allows Lakes workstations to use standard H.320 components for audio, video, and data transmission to other Lakes nodes using the ISDN or related public networks applications. The H.221 data partitions are used by Lakes both to send control data and to realize appropriate Lakes channels. The video and audio partitions are also used by Lakes for those channels with a data class that identifies the traffic as standards-compliant. Many H.320 adapters are incapable of routing audio and video digital streams to the workstation bus but instead have on-board devices that either generate or accept these streams. For example, the received digital video stream may be converted to analog and mixed with the monitor output to create a video display window. From a Lakes perspective, the application controls audio and video through logical devices and connects logical device ports to channels with suitable characteristics. Lakes detects the logical connection of the devices to the audio and video channels and passes this information to the device drivers. Through this mechanism there is no requirement to route the data into the workstation and thus the application is not written in an adapter-dependent manner; an attempt to connect an audio or video channel from such an adapter to a device not realized on the same adapter would necessarily be rejected. Use of H.320 in this way is not visible to applications.

Case 2a, still at the H.320 level, applies to many situations where a standards-compliant device is to participate in an interaction with one or more Lakes nodes essentially as an equal partner, although its function may be restricted; the most common examples are a telephone or a videoconferencing suite where the scenario is some form of conferencing. This is provided for by the virtual node and virtual application components of the architecture, where Lakes simulates the events that a real Lakes node would have generated. The non-Lakes device appears as a virtual node to Lakes nodes that communicate with it, while its various features (video, audio, etc.) appear as virtual applications running at that node. A call from such a device causes sharing with the identified local audio and video applications, just as if a call had been received from a node running those applications remotely.

Case 2b is much less common than 2a, for the H.320 equipment is now a remote, slave device under application control. The local Lakes application accesses it through a suitable logical device. This allows much greater control but requires specific programming to handle the situation.

Case 3 is the most important because as T.120 and H.320 protocols become commonplace, seamless interoperability is required. The requirement is to allow mixed network support so that a Lakes network can freely interoperate with one or more MCS hierarchies. One example is shown in Figure 16.

In order to preserve the integrity of both the Lakes and the MCS programming models, this mixed network must appear from a Lakes viewpoint as a Lakes network, complete with application sharing sets, and from an MCS viewpoint as a hierarchy of MCS providers, with the application instances in a domain. If it is assumed that an application exists at each node, $A_A \ldots A_E$ and $A_1 \ldots A_4$, and that all these instances are collaborating, then the two views can be represented as shown in Figure 17, assuming some allocation of instances to Lakes application sharing sets.

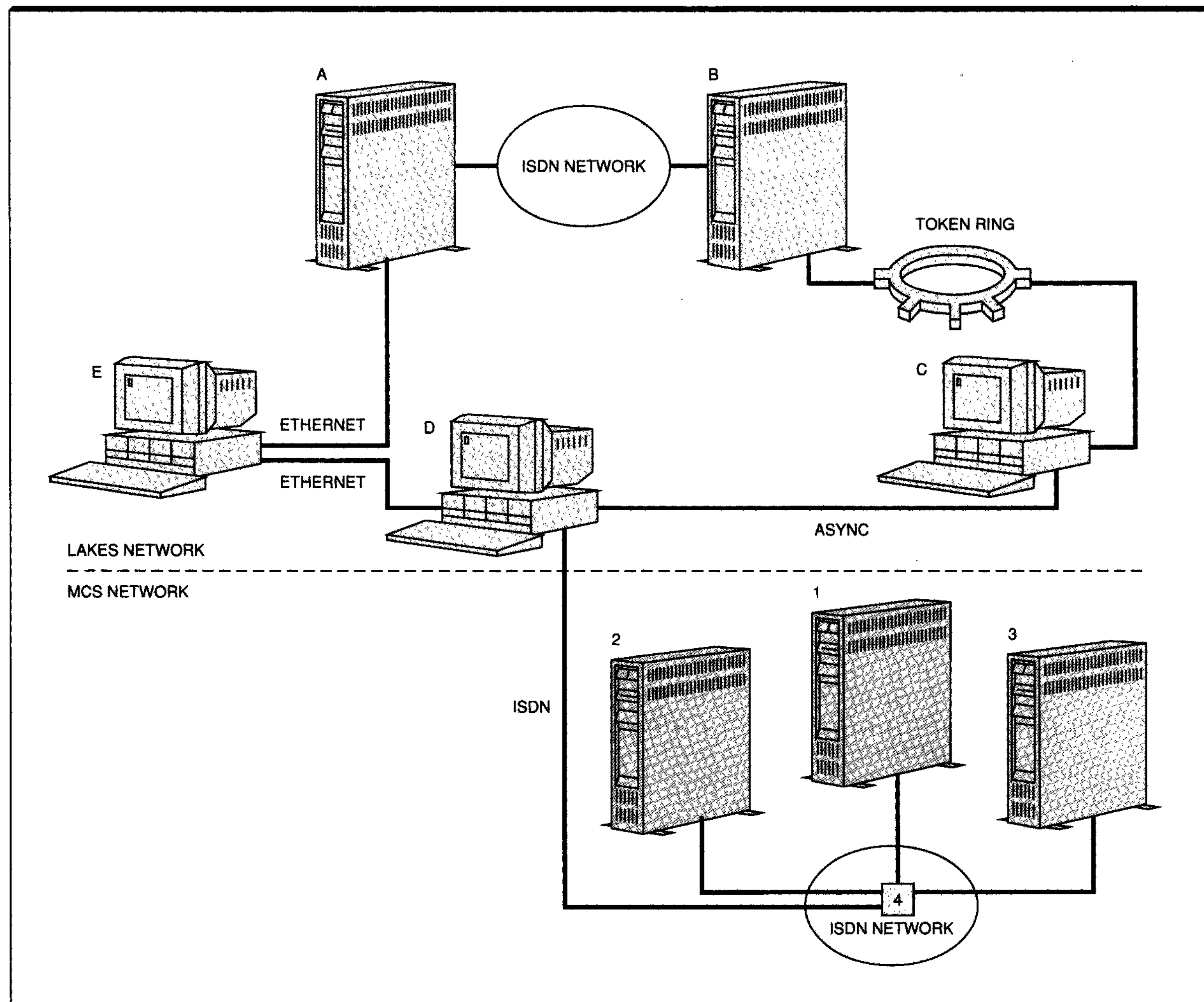
The Lakes node providing the link between the two networks simulates an MCS provider providing multipoint control unit services, and the actual Lakes nodes appear as MCS providers subservient to this bridge. Two views are possible; the preferred one in Figure 17 simplifies the implementation of functions such as data serialization, since they can be carried out at a Lakes node for the entire network.

Since Lakes offers a superset of the MCS capabilities, a Lakes API subset can be defined that is realizable on the MCS nodes; no such subset of MCS need be defined. Given the differences in the protocol streams produced by the two architectures, interoperability is based upon the following elements:

- Application proxies in both environments that simulate the real applications that each environment expects
- A Lakes call manager that establishes the application proxies and controls their operation

Each proxy collects the events that the real application should have received; it then uses the set

Figure 16 A mixed network

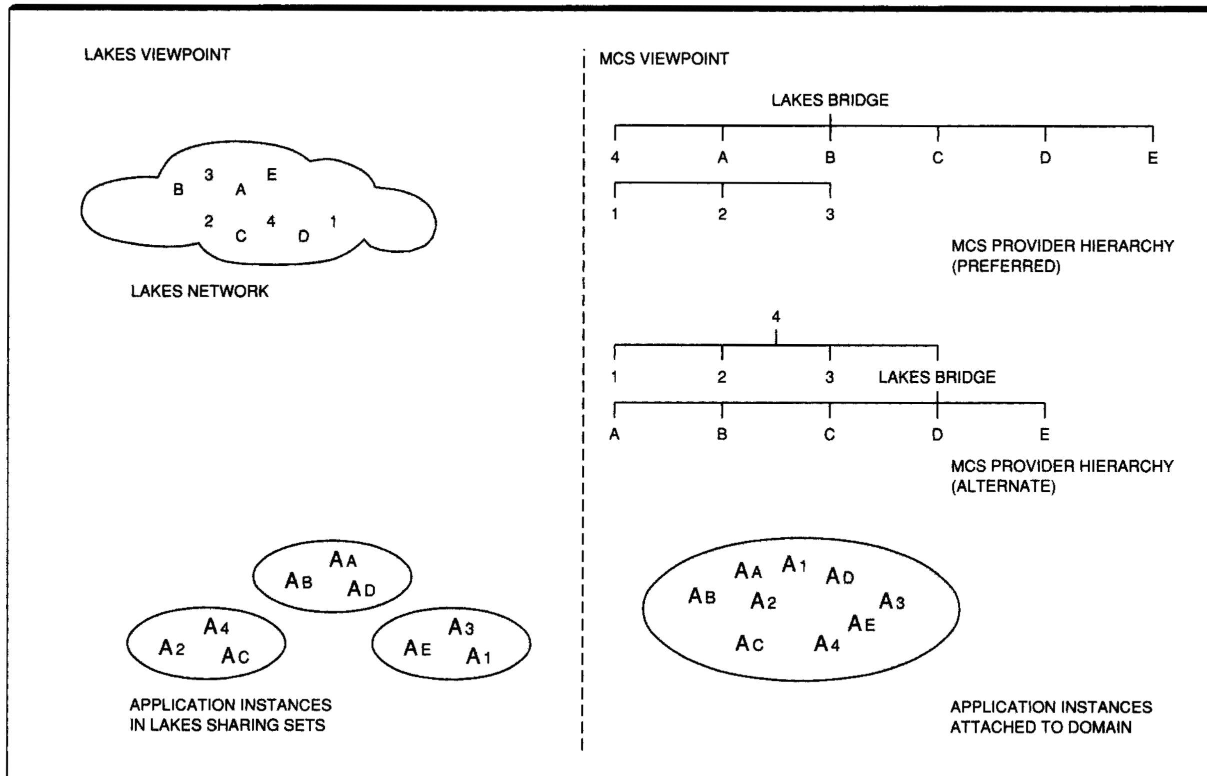


of application proxies in the other environment—each of which represents a different application to itself—to create a request from one of those that will have an equivalent effect in that environment. This mapping process can be controlled by the Lakes call manager at the node supplying the interconnection, and the Lakes audit manager can be used to police the compatible MCS subset of the Lakes API subset. This is illustrated in Figure 18 for a simple mixed network involving the four applications A1, A2, B1, and B2; one at each of the Lakes nodes L1 and L2 and the MCS providers M1 and M2.

This can be logically implemented as shown in Figure 19.

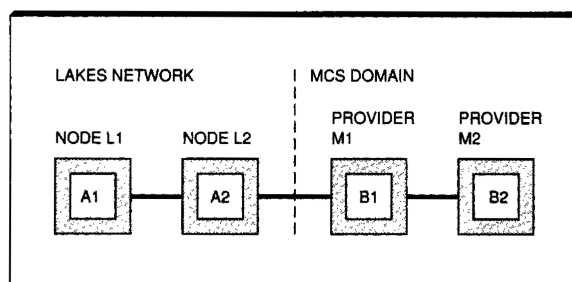
The Lakes network comprises two nodes L1 and L2. At node L1, application A1 is running; at node L2 there are three applications running: A2, b1 and b2. All four applications have registered with Lakes and joined a common application sharing set. From an MCS perspective, node L2 appears as three MCS providers; M3 is a multipoint control unit and M4 and M5 are running applications a1 and a2 respectively, each of which has attached to the MCS domain. Applications a1, a2, b1, and b2 are proxies

Figure 17 Lakes and MCS views of a mixed network



for the real applications A1, A2, B1, and B2. Thus application b1 receives the Lakes events that the real application B1 would have received if it were a Lakes application; instead b1 passes these events through the interface code that translates them into MCS requests, where they are issued in the MCS domain via either a1 or a2 as appropriate. The converse scenario holds true when the process is viewed from an MCS perspective. The proxy applications operate only at the Lakes or MCS entity level. It is not necessary that they understand the application significance of the information that they manipulate; only that they are able to convert between MCS and Lakes primitives.

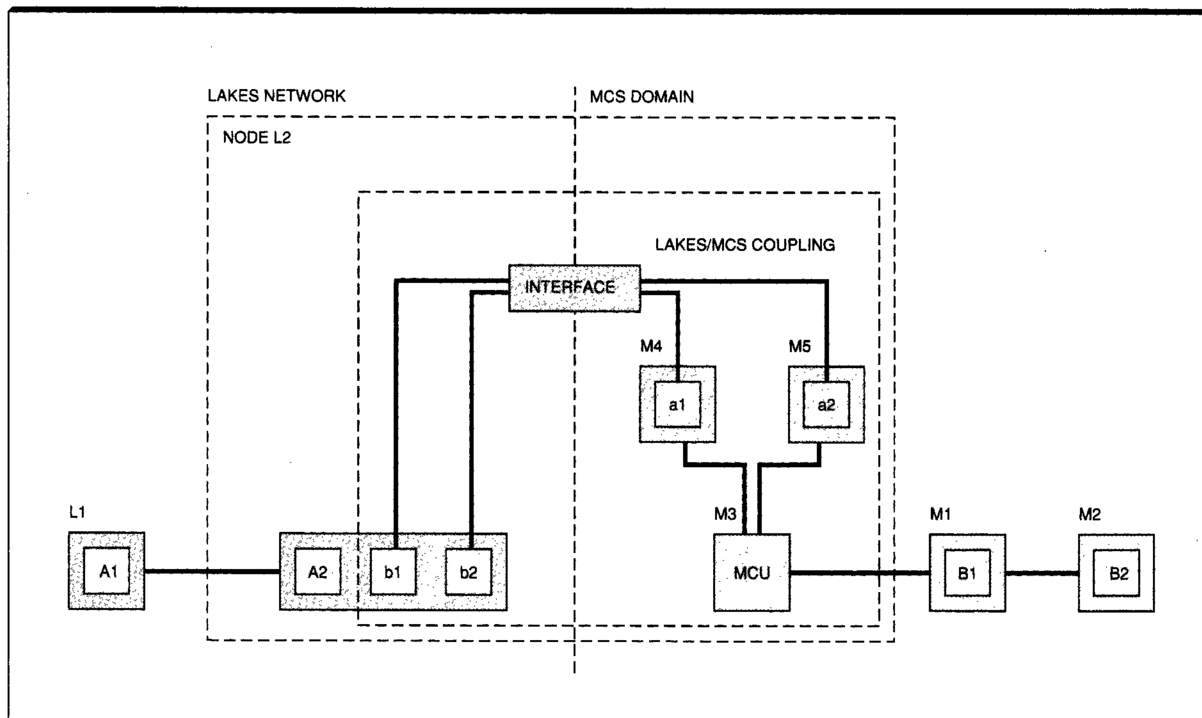
Figure 18 A simple mixed network



Thus, for Lakes applications that limit themselves to the subset API, full interoperability with MCS nodes is possible without any compromise to the Lakes model; moreover the full range of Lakes network configurations is enabled and all the local node capabilities of the architecture, such as logical device support, may be exploited without restriction.

The functional capabilities of Lakes beyond the subset API, such as multiple audio and video streams, synchronization across channels irrespective of stream format, passive operation, resource management, signaling, and certain call management functions and audit operations, can be exploited when interoperability with H.320- or T.120-compliant nodes is not required.

Figure 19 Implementation model



Summary

The Lakes architecture meets the requirements of a broad range of real-time multimedia collaborative applications and permits exploitation of network connectivity, capability, and intelligence. It allows the development of a range of call models above the API and addresses additional opportunities beyond the range of desktop conferencing over digital telephony networks, which has been the focus of the ITU-T H.320 and T.122 recommendations. Lakes also allows full interoperability with these standards, which is an essential prerequisite for any successful product implementation in this field.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. D. Marca and G. Bock, *Groupware: Software for Computer Supported Cooperative Work*, IEEE Press, Los Alamitos, CA (1992).
2. R. Baecker, *Readings in Groupware and CSCW*, Morgan Kaufmann, San Francisco, CA (1992).
3. R. Baecker, W. Buxton, and J. Grudin, *Readings in Human-Computer Interaction: Toward the Year 2000*, 2nd edition, Morgan Kaufmann, San Francisco, CA (1994).
4. J. M. Bowers and S. D. Benford, *Studies in Computer Supported Cooperative Work: Theory, Practice and Design*, Elsevier Science Publishers B.V., Amsterdam, Netherlands (1991).
5. J. Galegher, R. Kraut, and C. Egido, *Intellectual Teamwork: Social Foundations of Cooperative Work*, Lawrence Erlbaum Associates, Hillsdale, NJ (1990).
6. ITU-T Recommendation T.122, *Multipoint Communication Service for Audiographic and Audiovisual Teleconferencing Applications*, ITU, Geneva, Switzerland (1993).
7. ITU-T Recommendation T.124, *Generic Conference Control for Audiographic and Audiovisual Teleconferencing Applications*, ITU, Geneva, Switzerland (1993).
8. B. K. Aldred, G. W. Bonsall, H. S. Lambert, and H. D. Mitchell, "An Application Programming Interface for Collaborative Working," *Proceedings of the 4th IEE Conference on Telecommunications*, Manchester, UK, April 1993, IEE, pp. 146–151.
9. IBM Lakes Team, *IBM Lakes—An Architecture for Collaborative Networking*, R. Morgan Publishing, Chislehurst, UK (1994).
10. T. Baldwin, I. F. Brackenbury, H. D. Mitchell, and H. Sachar, "A Design for Multi-Media Desk-to-Desk Conferencing," *Proceedings of the 4th IEE Conference on Telecommunications*, Manchester, UK, April 1993, IEE, pp. 160–166.

11. *IBM Person-to-Person/2 for OS/2, IBM Person-to-Person for Windows, General Information*, G221-3686-01, IBM Corporation (1993); available through IBM branch offices.
12. B. K. Aldred, H. S. Lambert, and H. D. Mitchell, "Call Management in a Lakes Environment," *Proceedings of the 5th IEEE COMSOC Workshop—Multimedia '94*, Kyoto, Japan, May 1994, IEEE Communications Society, pp. 4.4.1–4.4.6.
13. B. K. Aldred, H. S. Lambert, and H. D. Mitchell, "Community Networking—A Lakes Approach," *Proceedings of the 1st IEEE Workshop on Community Networking*, San Francisco, CA, July 1994, IEEE Communications Society, pp. 11–19.
14. M. Arango et al., "The Touring Machine System," *Communications of the ACM* **36**, No. 1, 68–77 (January 1993).
15. R. Furata and P. D. Stotts, "Interpreted Collaboration Protocols and Their Use in Groupware Prototyping," *Proceedings of the ACM CSCW Conference*, Chapel Hill, NC, October 1994, ACM Press, pp. 121–131.
16. S. Greenberg and D. Marwood, "Real Time Groupware as a Distributed System: Concurrency Control and Its Effect on the Interface," *Proceedings of the ACM CSCW Conference*, Chapel Hill, NC, October 1994, ACM Press, pp. 207–217.
17. A. Prakash and M. J. Knister, "Undoing Actions in Collaborative Work," *Proceedings of the ACM CSCW Conference*, Toronto, Canada, November 1992, ACM Press, pp. 273–280.
18. C. A. Ellis, S. J. Gibbs, and G. L. Rein, "Groupware: Some Issues and Experiences," *Communications of the ACM* **34**, No. 1, 38–58 (January 1991).

Accepted for publication March 8, 1995.

Barry K. Aldred *IBM UK Laboratories Ltd., Hursley Park, Winchester, Hampshire S021 2JN, United Kingdom (electronic mail: bkaldred@vnet:ibm.com).* Dr. Aldred joined IBM in 1969 as a programmer, later moving to the UK Scientific Centre to research relational database systems. In 1980 he transferred to the Hursley laboratory and managed developments in workstation operating systems. More recently he has been responsible for the specification and design of printer subsystems and high-performance display adapters. Since 1990 he has led the advanced technology work on collaborative systems and is currently the manager of the Lakes architecture group.

Gordon W. Bonsall *13 Fyfield Way, Littleton, Winchester, Hampshire S022 6PB, United Kingdom.* Mr. Bonsall joined IBM Hursley in 1962 and worked initially on the architecture for OS/360 loading facilities. In 1966 he was appointed the manager for PL/I language control and later for PL/I language development. Subsequently he worked on various architectures for low-end and future operating systems. In 1982, after working on general display architecture, he became the manager for graphic object content architecture and was appointed an IBM Senior Technical Staff Member. Following an assignment to Entry Systems technical staff to develop graphic and image architecture, he returned to Hursley as a member of the CUA architecture review board until retirement in 1990. Following retirement he was a full-time consultant to the Lakes group until 1994.

Howard S. Lambert *IBM UK Laboratories Ltd., Hursley Park, Winchester, Hampshire S021 2JN, United Kingdom (electronic mail:snowhite@vnet.ibm.com).* Mr. Lambert joined IBM in 1968 as a systems engineer before transferring to the UK Scientific Centre as a systems programmer. In 1979 he moved to Winchester and took a development position on the transaction systems monitor CICS. Subsequently he moved to information systems, becoming a technical strategist in 1991. For the past three years he has been an architect and designer for real-time collaborative systems.

H. Dave Mitchell *IBM UK Laboratories Ltd., Hursley Park, Winchester, Hampshire S021 2JN, United Kingdom (electronic mail:zeno@vnet.ibm.com).* Mr. Mitchell has worked for IBM for over 25 years in a wide variety of roles, including development, technical support, and teaching. He joined the Hursley laboratory in 1987 where he has worked on several advanced projects, including the LPEX live parsing editor and Object REXX. Since 1990 he has been involved in the design and implementation of real-time collaborative systems. He is a member of the ACM.

Reprint Order No. G321-5580.