Connection management in a Lakes environment

Barry Aldred, Howard Lambert, David Mitchell

IBM UK Labs, MP 167, Hursley Park,
Winchester, Hampshire SO21 2JN, UK

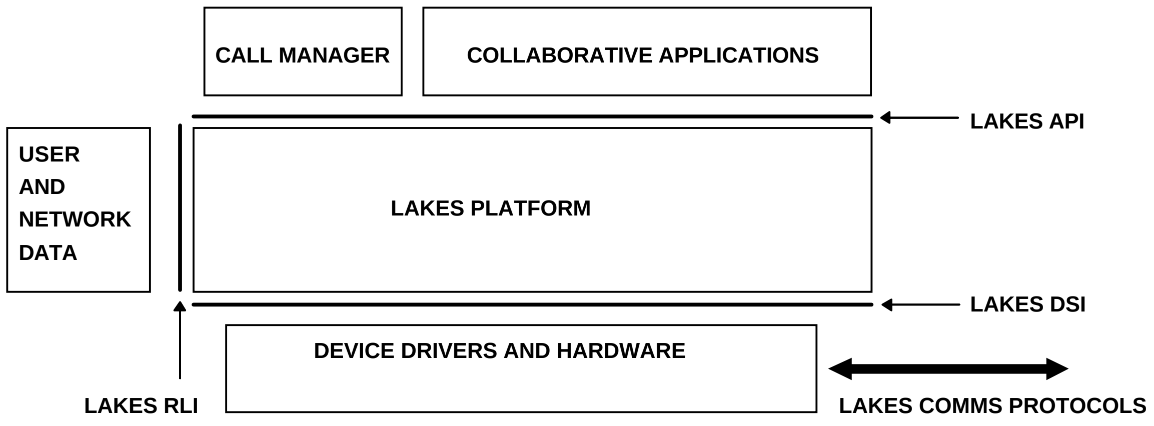
ABSTRACT

Lakes is an architecture for collaborative working developed to support a wide range of collaborative applications of the "same time/different place" variety across different platforms and communications media. When applications wish to share data in such an environment, connections must be established and channels created to provide the necessary communications links. As new nodes join or leave the conference, new connections must be made or broken. If an application is using several distinct channels for different types of data, the management and control of these channels can be a complex task. This paper presents an overview of the Lakes architecture and focuses on two particular features which have been designed to reduce the burden of connection and channel management on the application programmer:

- the ability of one particular Lakes application, the call manager, to provide flexible connection management on behalf of other applications.
- the facility to request that channels are automatically created and destroyed as applications are shared into or unshared from calls

1. AN OVERVIEW OF THE LAKES ARCHITECTURE

Experience gained in developing a series of desktop conferencing systems[1] over the last five years has led us to develop Lakes, an architecture[2] which supports a wide range of collaborative applications. Lakes defines four important interfaces, as shown in the figure below:



These interfaces are:

- the application programming interface (API) allows applications to request Lakes services.
- the device support interface (DSI) allows Lakes to support an extensible range of software and hardware sub-systems.
- the resources interface (RLI) through which Lakes requests details of nodes, users and network data.
- the Lakes communication protocols transmitted over the physical network, as a consequence of application calls through the API.

This paper does not discuss the DSI or RLI, concentrating on some specific features of the Lakes application programming interface, referred to from now on as the API. This API is designed to allow applications to:

- initiate peer applications and share resources, on a variety of hardware and software platforms, located on nodes across a diverse and complex communications network.
- define multiple dedicated logical data channels between shared applications, suitable for a broad range of multimedia traffic[4,5], independent of the structure of the underlying physical network.
- serialize, synchronize, merge or copy the data streaming between shared applications.
- support a range of attached devices and to allow the interception and redirection of the device data.

A Lakes platform includes other components to assist application development:

- an extensible set of logical devices, interfacing to external applications and devices.
- a set of end-user utilities, written to the API, whose function can also be invoked from applications through a command level interface (CLI).

1.1 Network, nodes and applications

At the highest level, the programming model consists of a communicating set of nodes. A **node** is the addressable entity in Lakes representing a user, and comprises an instance of Lakes, and a set of resources, such as application programs, data etc. Usually a node is a dedicated programmable workstation, capable of communicating with its peers; in a multi-user system a node is associated with each user.

Nodes are identified by name; ideally all node names should be unique but duplicates can be tolerated as long as their associated nodes are never required to inter-communicate. The node naming scheme is not prescribed by the architecture but a hierarchical system, such as that defined by the Internet protocol, has many benefits. A collection of inter-communicating Lakes nodes is called a **Lakes network**. It is fundamental to the architecture that any node, independently of any others, can dynamically join or leave the network.

Nodes can contain logical devices. A **logical device** is a software extension to Lakes that allows an application to manipulate or manage software or equipment, and to do so in a way which is consistent with the other entities in the Lakes model. There are many possible Lakes logical devices including, for example: presentation windows, printers, disk drives and the system clipboard.

Multiple applications can be executed at a Lakes node, subject to the constraints imposed there by the operating and windowing systems. Applications are either Lakes-aware or Lakes-unaware; an **aware application** invokes the services of the Lakes API; an **unaware application** does not. Both aware and unaware applications will generally be executing simultaneously at a node. Although Lakes provides logical devices to assist in sharing windows belonging to unaware applications, it does not currently support the direct collaborative use of such applications, unlike MMConf[4] or Shared X[6].

The API consists of a set of function calls to LAKES together with a related set of events. The first function call an aware application makes establishes an event handler which will be sent Lakes events, most of which, being the result of function calls issued by remote applications, are asynchronous. The programming style is thus very similar to that required when writing applications for a GUI such as OS/2 Presentation Manager, Windows or X11. Work is currently underway defining an object-oriented API and class library for Lakes.

1.2 The call manager

In order for Lakes to be fully active at a node, one particular aware application must be running at that node. This application plays a unique role and is known as the **call manager**. Many call managers may be available for execution at a particular node but only one instance of an aware application can be performing this role at any time. The distinguishing feature of a call manager is that it responds to certain events generated by Lakes; these are typically concerned with name resolution or resource management for the node. Call manager responsibility can be transferred from one application to another; also the call manager role can be combined with user application function, if that is appropriate.

The Lakes support software may request that the resources of one node be made available for Lakes communication between two other nodes; this is termed **passive operation** and permission must be granted by the call manager at the passive node. An example of this may be two nodes, ALPHA and BETA on a LAN, with a third node GAMMA connected to BETA by an asynchronous communications link. If applications at ALPHA and GAMMA wish to communicate, the traffic will need to be routed via BETA. The consent of the call manager at BETA is required for its node to be used in this way.

1.3 Sharing sets

Aware applications can conveniently share data and resources with other aware applications at the same or different nodes by joining an **application sharing set**. Application sharing sets are named collections of application instances.

A Lakes application initiates a **share** request naming an application sharing set, a destination node and a target application. 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 the source application will be informed. The participation of the call managers in this process allows both local control of the sharing process and other actions to be initiated if necessary. The call managers play a vital role in resolving the names used by applications to identify other nodes 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 naming the same sharing set, with the result that all three applications are then sharing with each other.

Applications may also make local share requests on behalf of other applications thereby allowing membership control of the sharing set to be delegated. Facilities exist for either the issuer, or the target of the share request, to name the application sharing set. These names are not required to be unique; thus multiple sharing sets with the same name can exist.

Individual applications can cease sharing at any time and thereby withdraw from a sharing set, the remaining applications in the set being then notified of the application withdrawal.

1.4 Communications, channels and ports

Applications in a sharing set can establish data communication links, known as channels, with each other. **Channels** are logically dedicated, uni-directional pipes, with application specified transmission characteristics. A channel is always defined by the sending application and goes from the sending application to a receiving application. The ends of channels are known as **ports**; thus each channel has one sending port and one receiving port. A **sending port** sends data **packets** down the channel; a **receiving port** receives data packets 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-aware applications and the physical communication network in existence between the nodes.

An application may establish multiple channels to another application as a convenient way to separate data traffic of different types. Lakes may map some or all of the logical channels on to a single physical link, but this will be invisible to the application.

Channels have a number of **quality of service** characteristics, initially negotiated with Lakes during the creation process, which allow data transmission characteristics to be tailored to the requirements of the expected traffic. The quality of service parameters are defined according to the signal **type**, which distinguishes analog from digital data. They need not be specified explicitly but can be notified to Lakes in terms of the **data classes** that are to be transmitted down the channel. This mechanism allows video, voice and other data channels to be sensibly established. Channel characteristics can be re-negotiated after channel creation. Channel quality of service may also be left **undefined**; this allows channels to be created whose operational characteristics depends upon the resources available when data is being sent down the channel.

Channels may be collected into named sets; such named sets are known as **channel sets** and may be of one of four types: standard, merged, synchronous and serialized. **Standard channel sets** provide a convenient way of referring to a collection of channels, but the individual behavior of the constituent channels is not changed. Through a **merged channel set** however, data packets are taken from the constituent channels and delivered to each receiving application through a single port. There is no guarantee that each application receives all the data packets in the same sequence, only that each application receives all the packets. Through a **serialized channel set** data packets are taken from the constituent channels and then delivered to each application such that each receiving port receives its data packets in the same sequence as the other receiving ports. Through a **synchronized channel set** data packets are synchronized, so that the packets on the separate channels are tied together in time (for example, voice with video), but are delivered through the individual ports belonging to their respective channels. Channel set names are local to an application sharing set.

2. THE ROLE OF THE CALL MANAGER IN APPLICATION SHARING

2.1 What is a call?

Because of its role in handling share requests, the call manager is the visible manifestation of a set of policies. What we mean by a "call" is the result of a set of policy decisions:

- Are all the parties in a call equal? Can any one of them add a newcomer? Do the other parties get a vote?
- Can people outside the call join in or do they have to be invited in by an existing member?
- Can anyone launch a shared application? Is it automatically launched at all nodes in the call?
- Can anyone end a multi-way call or merely leave it? Can anyone leave at any time?

- Can two calls be merged into one or one call be split into two?

Most of these questions do not have a single right answer. Rather each combination of answers defines a possible meaning of the word **call**. It should be clear that if we embed the answers in the system itself, we hardwire the notion of a call. In the Lakes architecture we have deliberately arranged that all such call-related policy decisions are surfaced to the call manager for resolution, which means that alternative policies can be imposed by reprogramming or replacing it[3].

It should be clear that there are many possible call managers. The simplest, modelled on the metaphor of an informal telephone call, impose few rules on users. By contrast, a call manager for a formal, chaired meeting is likely to implement rules of order, provide facilities for minute taking and possibly support operations such as voting.

2.2 The sharing process in detail

Before we examine how a call manager can implement the notion of a call, we need to examine the sharing process outlined in section 1.3 in more detail. An aware application initiates a **share request** by naming a sharing set and a source and target application, where the issuer need be neither the source nor the target of the share. The receiver of a share request may accept it, reject it or transfer it to another application. Such a request is passed:

- to the source, if this is not the issuer. This allows an application to prevent itself being shared when it does not wish to. For example, a two-player game application might reject attempts to enlarge the size of its sharing set beyond two unless it supported kibitzing.

- to the call manager at the source node. This ensures that unscrupulous applications cannot make covert calls to other nodes without the call manager being aware of it. Our standard call manager rejects an outgoing share request initiated by an application if it is to a node which is not already a party to an existing call.

- to the call manager at the target node. Typically this will either launch a fresh instance of the requested application or, if a suitable instance is already running will transfer the share to it. Our standard call manager allows applications to be marked as non-sharable, sharable by launching a fresh instance, sharable if currently unshared or sharable even if already shared.

- to the target application. In rare circumstances this may wish to reject sharing, particularly if it's current state and stored data is incompatible with a new sharing operation.

- assuming the share request is eventually accepted, share confirmed events are sent to the existing members of the sharing set informing them of the newcomer. They in turn send share confirmed events to the newcomer informing it of their existence. Of course, requests to enlarge a sharing set may be sent out by more than one node in the set at the same time. The order in which share confirmed events are sent out guarantees that this does not cause confusion.

The unsharing process is much simpler:

- an unshare request is sent to the target of the unshare, if this is not the issuer. This allows an application to refuse to be unshared if it is in the middle of an uninterruptible operation, such as the transfer of data as a series of blocks to other members of its sharing set.

- assuming the application agrees to be unshared, unshare events are sent to the members of the sharing set informing them of the departing instance.

2.3 The role of the standard call manager

Given the way sharing works, our standard call manager implements the concept of a call by forming sharing sets with other call managers. Each such sharing set defines the set of nodes within that call. Further, it provides commands by which the user can share applications into a call or unshare them from one. Sharing an application into a call involves constructing an application sharing set that is "congruent" with the call manager sharing set, that is involving the same set of nodes. The call manager tries to keep such application sharing sets congruent with the corresponding call manager sharing set. Thus when a new node is invited to join a call, its call manager joins a call manager sharing set. The inviting call manager will then issue share requests on behalf of all the applications currently shared into the call to extend their sharing sets to the new node too. Similarly when a node leaves a call, its call manager then leaves such a sharing set. It will then issue unshare requests for all the applications in the call to cause them to leave their sharing sets too.

To illustrate this, consider the following sequence:

1. the user at node ALPHA makes a call to node BETA. A share request is sent to the call manager at BETA, asking it to join a sharing set. The name, chosen by ALPHA, identifies the call - let us assume it is "ALPHA 1". The call manager at BETA recognises the request as an incoming call, since it itself is the target, and either responds automatically or asks the user to decide. Assuming the call is accepted, the call managers at ALPHA and BETA are now in a sharing set.

2. the user at node BETA uses his call manager to launch a shared text editor application and then share it into the call. A share request is sent to ALPHA, where the call manager recognises it as a request to share the appropriate application. It launches an instance of the shared text editor and transfers the share request to it. Assuming the application accepts the request, the two shared text editors are now in a sharing set which matches that of the call managers.

3. the user at node ALPHA extends the call to include node GAMMA. A share request is sent to GAMMA, using the name "ALPHA 1". If the call manager at GAMMA accepts, the sharing set will now have three members. The inviting call manager, at ALPHA, will then issue a share request to node GAMMA on behalf of the shared text editor which is currently shared into call "ALPHA 1". Assuming such an application is launched at GAMMA and accepts the share request, the shared text editor sharing set will then also have three members.

4. the user at node ALPHA decides to leave the call. His call manager issues an unshare on behalf of the shared text editor, to remove it from the call and then unshares itself. As a result, the call managers at nodes BETA and GAMMA remain in the call, as does the shared text editor at those two nodes.

It's perfectly possible for a given call manager to participate in several calls simultaneously, since each involves a sharing set with a different name. Similarly, it's possible to write applications that can be shared across several calls.

2.4 The role of applications in sharing and unsharing

It should be clear that this means that applications do not need to issue share requests or unshare requests themselves. The call manager's user interface provides the means by which the user can launch applications and share and unshare them from calls. This relieves the application programmer of the burden of providing user interface mechanisms for listing the available calls and nodes, selecting from them, sharing and unsharing and so forth, effort which would have to be duplicated for each aware application.

Many applications will want to take note of the share confirmed and unshare events however. At the very least they will need to be aware of the transition from unshared to shared activity, because this is likely to have user interface implications, for example menu choices relating to certain shared functions may need to be grayed or ungrayed. They may also wish to maintain a list of which nodes are currently in the call. The programming effort required to process these events is typically small.

3. SHARING SETS AND CHANNEL SETS

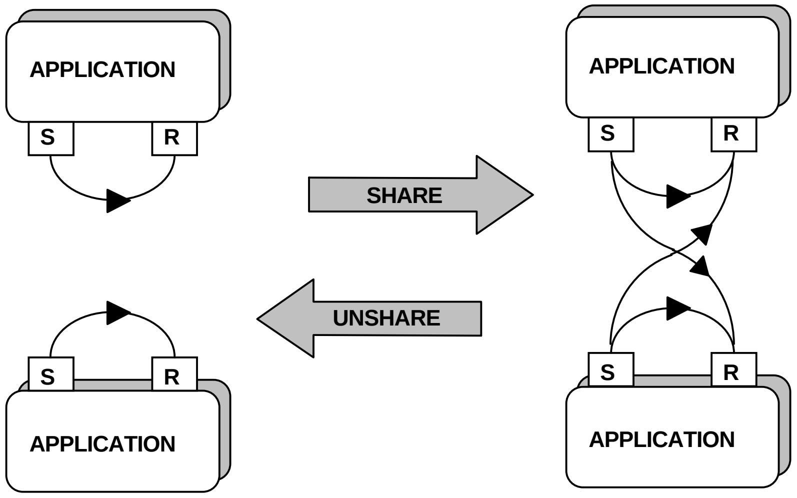
As was mentioned in section 1.4, channel sets are local to application sharing sets, so that applications must join a sharing set before they can construct channels to other applications. This also means that when an application leaves a sharing set any associated channels are automatically destroyed. Merged and serialized channels have an additional property however, which makes them extremely convenient to the programmer of collaborative applications.

3.1 Implicit channel creation

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 additional channels required will be created automatically.

The application programmer can use this mechanism to drastically simplify the writing of a collaborative application. If the application creates a merged channel to itself during initialisation and is written so that all user input is sent out via the sending port of this channel and processed when it arrives at the receiving port, then if the application joins the sharing set of another application which has defined a merged channel set with the same name, Lakes will automatically construct the two cross-channels. If a third such application joins the set, again Lakes will create the necessary four cross-channels. Thus any packet of data sent by the application from its sending port will be transmitted to each of the receiving ports. Each instance therefore receives all the packets. When an application leaves a sharing set, these cross-channels will be destroyed but the channel to self will be retained.

The figure below shows this process:



3.2 Serialization

The use of a serialized channel confers the additional advantage that each instance receives the same packets in the same sequence. Using serialized channels, a shared chalkboard application can allow any user to use any drawing or erasing tool at any time, in a free-for-all, since the results at each node will be consistent. Without serialized channels this may not be the case. For example at one node, a local drawing operation might be processed before a remote erasing operation, while at the remote node the two operations might be received in the reverse order.

There is a performance overhead however, since serializing is achieved (in our current implementations at any rate) by sending all packets to a serialisation process at one node which then rebroadcasts them. The node at which serialisation takes place can be chosen dynamically, based on the nodes in the sharing set and the physical links between them. Lakes also provides alternatives to serialisation, such as a set of token management calls which allow applications to serialise the use of certain resources instead of serialising data packets. Use of these facilities allows the programmer to trade increased programming complexity for better performance.

4. CURRENT STATUS

Our first Lakes prototype, running under Windows, implemented about 75% of Lakes. Over the last year or so, we have used this prototype to build a wide range of applications, including several games, shared text editors and chalkboards, as well as several different call managers, all with remarkably little effort. Our simplest application, a shared scribbling surface, consists of less than a dozen lines of Visual Basic. A more complete Windows implementation is now under construction and work is also proceeding on an OS/2 version.

5. ACKNOWLEDGEMENTS

The authors wish to acknowledge the substantial contributions made to Lakes by Gordon Bonsall and Peter Cripps.

6. REFERENCES

1. T. Baldwin, I. Brackenbury, D. Mitchell. H. Sachar, "A design for multi-media desk-to-desk conferencing", 4th IEE Conference on Telecommunications, pp. 160-166, Manchester, April 1993

2. B. Aldred, G. Bonsall, H. Lambert, D. Mitchell, "An application programming interface for collaborative working" 4th IEE Conference on Telecommunications, pp. 146-151, Manchester, April 1993

3. M. Arango et al, "Touring Machine: a software architecture to support multimedia communications", 4th IEEE COMSOC MultiMedia Workshop, pp. 186-189, Monterey, April 1992

4. T. Crowley et al, "MMConf: an infrastructure for building shared multimedia applications", 3rd ACM CSCW Conference, pp. 329-342, Los Angeles, October 1990

5. S. Ahuja et al, "A comparison of application sharing mechanisms in real-time desktop conferencing systems", ACM Conference on Office Information Systems", pp. 238-248, Cambridge, April 1990

6. D. Garfinkel et al, "The SharedX user's guide", Technical Report STL-TM-89-07, Hewlett-Packard Labs, March 1989