AN APPLICATION PROGRAMMING INTERFACE FOR COLLABORATIVE WORKING

B K Aldred, G W Bonsall, H S Lambert, H D Mitchell

IBM UK Laboratories Ltd, England

Personal computers are now widespread throughout the business community and many are able to inter-communicate, either through fixed connections e.g. local area networks, or through dynamically established links e.g. ISDN or async lines over the public switched telephone network. Increasingly, these connected personal computers can be used to enhance collaborative working between remote individuals; a typical example being the use of desk top conferencing software [Ensor et al (1); Clark (5)]. Successful collaborative work generally requires more than a simple data link between the participants; voice capabilities are normally essential and video links are frequently required. Thus remote collaborative working can often be regarded as an extension to the traditional telephone call - it being enhanced with the data and programs available at the desktop via the personal computer - and, on occasions, enriched with video services.

A broad spectrum of collaborative applications can be written, ranging from utilities taking advantage of the data and applications on a workstations, e.g. sharing of application windows, through to new collaborative applications designed to meet the needs of specific classes of remote user e.g. shared editors [Knister and Prakash (3)]; real time conferencing [Akuja et al (6)]; help desk; remote presentations; and many more. The common requirements behind these examples are:

1. Support of a wide variety of personal computer platforms - both hardware and software.

2. Operation over the existing communication networks.

3. Group communications and multi-media data services.

Experience gained with the implementation of the IBM Person-to-Person desk top conferencing product (7), has led to these proposals for an enabling platform which supports an extensive range of collaborative applications. Four interfaces are proposed:

1. An **application programming interface** (API) to allow applications to request group communications and multi-media data streaming services.

2. A **device driver interface** to allow the support of an extensible range of software and hardware communications sub-systems.

3. A **resources interface** through which details can be requested of externally held and managed information giving details of the installed communications network, such as node addresses, link capabilities and directory data.

4. A **data stream**, which is the information and associated protocols actually transmitted over the physical network, as a consequence of application calls through the API.

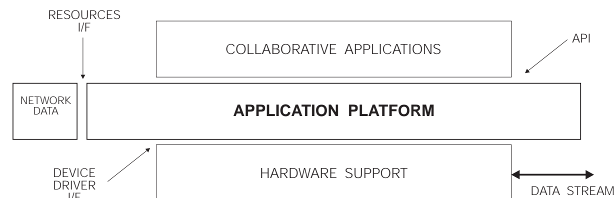

Figure 1. Proposed interfaces

The API needs to offer the following capabilities:

1. Applications to be able to start the execution of other local and remote applications, and share their combined resources.

2. Applications to be able to define logical communications channels between themselves, suitable for a broad range of multi-media traffic, independently of the underlying physical communications network. This

requirement has also been discusssed by Ahuja and Ensor (2).

3. Communications traffic to be able to be:

   a. Serialised - so that the same data sequence is received by all applications.

   b. Synchronised in time with other communications traffic.

   c. Merged - so that multiple sources of data can be combined.

   d. Copied to multiple destinations.

4. A range of attached workstation devices to be supported to allow the interception and redirection of data, both from devices and from other applications.

An architecture for the API is proposed that satisfies these requirements and allows the development of a network based, real-time collaborative platform.

Major elements of the architecture

Network, nodes and applications

At the highest level, the architectural model proposed is that of a communicating set of nodes. A **node** is a computer, typically a workstation capable of communicating with its peers. Nodes are linked together, such a collection of inter-communicating nodes constitutes a **network**. It is fundamental to the architecture that a node can dynamically join or leave the network. It is also assumed that the network topology can range from the simple to the complex; for example:

1. There may be multiple direct and/or indirect links between any two nodes.

2. Links may be switched (e.g. async, ISDN) or may be fixed.

3. Links may have very different characteristics (e.g. in terms of jitter, latency, reliability and bandwidth).

4. Certain links may offer bandwidth reservation capabilities.

5. Broadcast mechanisms may exist from a node to a subset of the other nodes.

Nodes can have associated logical devices. A **logical device** is a piece of equipment, a computer or a program that is communicating with a node. Logical devices are normally controlled by the node to which they are attach and frequently supply or receive data to or from that node. There is an extensive range of possible devices including: printers, cameras, disk drives, modems, codecs, multi-point control units, application windows and programs.

Multiple applications can be executed at a node, subject to the constraints imposed by the operating system and windowing system environment at that node. Applications may be classified as either aware or unaware; an **aware application** is defined as one that is written to use the services of the API. The use of unaware applications in a collaborative environment has been discussed by Crowley et al (4).

It is a requirement of the architecture that one particular aware application must be running at an active node. This application plays a unique role at that node and will be known as the **call manager**. This performs many of the functions of the conference manager in (4). Many call managers may be available for execution at a particular node but only one can execute at a time. The distinguishing feature of the call manager is that it handles resource management for the node and resolves any requests that are not directed specifically at an instance of an application. Call manager responsibility can be passed from one call manager to another.

Aware applications can share data and resources with other aware applications at the same or different nodes. To do this the application initiates a **share request**, specifying, by name, the application and its node location. The node name is resolved by reference to a network data base through queries issued externally via the resources interface. The application name is processed at the destination node by the currently active call manager. It can accept or reject the share request; if it accepts the share it can decide to handle the sharing itself, to launch an application to be shared, or to share with an existing instance of an application. This sharing mechanism can be cascaded, such that if two applications are already sharing, one of them can initiate a share with a third application. The consequence of this is that all three applications are then sharing with each other. Applications may make share requests on behalf of other applications. A

collection of applications sharing is called a **sharing set**. Applications can cease sharing at any time i.e. withdraw from the sharing set.

Once the addressability between applications has been resolved all subsequent interactions between members of a sharing set are direct, using application and node handles.

Communications, channels and ports

After a sharing set of applications has been established, there is typically a need for the applications to exchange data. This is implemented through a channel mechanism. **Channels** are logically dedicated, uni-directional links between applications, intended to pass a particular type of data e.g. voice, video, mouse movements, keystrokes etc. A channel is always defined by the sending application and it goes from the sending application to one or more receiving applications. The ends of channels are termed **ports**. A **sending port** sends data down the channel; a **receiving port** receives data from the channel. There is no direct mapping between the logical channel structure seen by the aware applications and the physical communication network in existence between the nodes. Multiplexing or de-multiplexing of the data is handled below the API.

An application may establish many channels, of different capabilities, to another application in its sharing set, as a convenient way to communicate different kinds of data. Some or all of these may be mapped on to one or more physical links but this will be invisible to the application.

Channels have a number of characteristics which are negotiated during the creation process to allow data transmission characteristics to be tailored to the expected traffic; these include **encryption** and **quality of service** parameters, such as capacity and latency. These mechanisms allow suitable video channels, voice channels and other specialised data channels to be established.

Channels may be collected together into named collections known as **channel sets**; any channel set may be classified as being of one of four types: standard, merged, synchronous and serialised. **Standard channel sets** are the default case where the channels in the set are conveniently referred to collectively by the channel set name, but no other consequences follow from channel set membership. The other three types have important consequences for the behaviour of the constituent channels. Through a **merged channel set** data is combined from multiple channels and delivered, to an application, through a single port at each destination. Through a **serialising channel set** data is combined from different channels, serialised**,** and delivered to each application such that each receiving port sees the same sequence of data. Through a **synchronising channel set** data is synchronised, so that data on separate channels is tied together in time (as is required for the synchronisation of voice and video), but delivered through the individual ports belonging to the channels.

Ports may be **connected** together to establish extended communication links, so that an application may route its inputs through to another application for processing. When ports are connected in this way no further application involvement is required after the routing has been established. This allows the **streaming** of data between applications and devices. Connected ports can also be **welded**, so that the connection is permanent and persists even when the connecting application has terminated.

Event, command and null ports are required; **event ports** generate an event when data is either available or is required; **command ports** read or write data to/from a buffer, and the application has the responsibility of filling or emptying this buffer; **null ports** are a special case reserved for ports that are unable to supply data to the application e.g. ports on analogue channels. Ports can be controlled through commands which are sent to the port event handler routine supplied by the application when the channel was created; typical commands are, for example, *rewind* or *pause* to a tape drive.

An alternative method of application inter-communication, avoiding the use of ports and channels, is provided for application control data.

One characteristic of ports is that they are associated with a **data class** and **compression hints**. The data class describes the kind of data, e.g. voice, video, file, interactive, that is sent by a sending port down the channel, or to be received via a receiving port. Compression hints allow data compression during transmission, without impacting applications.

Negotiation of quality of service

Certain applications have fixed quality of service requirements for the channels needed to communicate with other applications. In these cases the channels may be established directly, using a *create_channel* request. Parameters on this request identify the receiving application and both the channel and the sending port characteristics. If the resources are available, and the receiving application accepts the request, then the channel will be created.

Some applications are more flexible in their quality of service requirements and need to determine what is available to a particular node and then use this information in setting the parameters of the *create_channel* request. This is accomplished through the *query_resource* command. The subsequent *create_channel* can request an equal or lower quality of service and expect the request to be satisfied, if there is not competition for the communications resource.

Other applications have flexible quality of service requirements, but need to compromise the specification over a number of channels. This can be achieved by means of the *reserve_resource* command specifying a **resource set identifier** and a quality of service. This has the effect of reserving that resource and associating it with the specified identifier. This identifier can then be specified in a subsequent *create_channel* command, in which case the resources are allocated from those reserves. The *query_resourc*e command can be used to determine remaining resources in a resource set.

Certain applications need to dynamically change their channel characteristics during execution; for example, available bandwidth must be re-allocated across channels. This can be done through the *change_channel* request, specifying a resource set identifier. The resources are given to, or taken from, those resources associated with that identifier. This technique allows, for example, a fixed resource to be secured for an application to application communication, and then re-allocated dynamically according to the traffic e.g. video bandwidth can be temporarily reduced to allow faster file transfer.

Logical devices

Logical devices are supported by the architecture; these include: disk drive, printer and window; further logical devices may be defined. Logical devices can be **opened** by an application; the process of opening creates a port. A logical device may be opened more than once to have multiple ports if appropriate; thus a disk drive logical device can have both a sending and a receiving port. Two classes of logical device are required: real and virtual. A **real logical device**, when opened, provides a port that reads or writes data to a physical or logical entity, such as a disk drive or system clipboard. A **virtual logical device** however, provided primarily for use with unaware applications, is generally a replacement for a standard device driver. Thus a virtual printer logical device can replace the standard printer device on LPT1, and redirect the data to a destination port.

Resource management

Collaborative working frequently requires that resources owned by a node, for example a printer device, can be shared with other nodes. Such resources are considered to be global resources and access is controlled through **global tokens**. Other resources are owned by an application, for example a shared pointer, and access to these is managed through **application tokens**.

Applications are expected to know the location of a globally available resource that they require, and therefore facilities for the broadcasting of availability information are not provided. Instead, the call manager at the node with the global resource is responsible for resource management. Global tokens may be held by an application instance on an exclusive or shared basis; global token ownership may not be transferred. Requests for a global token may be queued, with the queue being held above the API and managed by the node call manager. Access to global tokens is not restricted to an application sharing set.

Management of application resources may be performed by any application in the sharing set. Application tokens may be held on an exclusive or shared basis and requests for tokens queued, with the queue being held above the API, and managed by the current application token owner. Application token ownership may be transferred across an application sharing set.

Other networks

Private analogue networks: The architecture supports analogue communications in a very similar way to digital communications, in those situations where:

1. Analogue links exist between nodes.
2. Connectivity and routing at each node can be controlled.
3. A digital control channel exists between the nodes.

Analogue channels are logically dedicated, uni-directional communication links, established by the sending application, and they may terminate in more than one receiving application. They may be distinguished from digital channels by their data class. Only standard or merged channels may be established; serialising and synchronising channel sets are not permitted.

Logical devices can present analogue ports when opened; thus a **video player logical device** can be used as a source of analogue video and may be connected to an analogue channel. The direct connection of analogue and digital channels is not permitted; however certain logical devices e.g. a **codec logical device** provide both analogue and digital ports when opened and can be used to effect such a coupling.

Switched digital networks: Switched digital networks can be used for inter-node communication without exposing the switched nature of the connection.

Equipment, such as digital telephones, attached to a switched network, are accessed by applications through logical devices. Thus an **ISDN phone logical device** may be opened to present receiving and sending ports, with an associated event or command connect type; dialling, and other control functions, are implemented through port commands. Third party connection between digital telephone equipment is similarly affected through commands to an appropriate logical device; this may be physically implemented through commands to the local switch.

Public switched analogue networks: Analogue telephones and other equipment, attached to the public switched network, are similarly accessed. A **PSTN telephone logical device** can be opened to present a port, but with a null connect type i.e. it cannot supply or receive data from an aware application. Port commands are used to control the device. First party connection can be implemented through a modem injecting dialling tones into the local line; third party connection, and multi-way calls through commands to the local switch.

Interfacing to unaware applications

The architecture described above provides facilities which allow unaware applications to be used for collaborative working. An aware application supplies the user interface dialogue and interacts with the particular unaware application via virtual logical devices. This same aware application then communicates with a related aware application at the remote node to pass the information to the remote user.

An example of this is the sharing of an application window across a network. A virtual window logical device is opened such that it intercepts the output that an unaware application is making to its presentation window on the sending node. Similarly, a real window logical device is opened at the receiving node, such that data can be displayed in a presentation window at that node. The port on the sending node virtual window logical device is then connected, via a channel, to the port on the receiving node real window logical device. A copy of the unaware application window is then displayed at the remote node. The parameters specified when the virtual window logical device is opened control whether window snapshots or continuous updates are captured from the unaware application.

Other facilities

User exits and **function hooks** are provided to ease programming and debugging. All ports can be associated with a user exit to monitor data traffic or process data. All functions, including calls and events, can be hooked and passed to an application supplied event handler. Additionally, a full set of **queries** are provided, so that applications need not keep track of status at their node, nor of the applications being shared. Application program debugging is assisted through allowing collaborative applications to be shared at a single node; thus avoiding physical networks being involved during initial program development.

Discussion

The architecture permits applications to dynamically interact with each other, and through the call manager mechanism, gives each node control of the application sharing process at that node. This control can be arbitrarily complex and sophisticated. Allowing applications to share on behalf of each other allows switching utilities to be developed, whose sole job is to connect and disconnect other applications.

The channel mechanism, based on multiple uni-directional, logically dedicated communication links, requires an application to precisely specify, its data communication requirements, without any reference to the underlying physical network. The mapping to that network is handled quite independently below the API. Moreover, the nature of the data flowing over these links can be queried independently of the data itself; thus facilitating one application inter-working with another and about which it had no prior knowledge. Requiring the data type to be independently specified as an attribute of the sending and receiving ports permits the network to offer data conversion facilities, again assisting inter-application communication.

The ability of one port to send data to multiple ports; and the corresponding ability for a single port to receive data from multiple sending ports leaves open the question of where data gets cloned, and where data flows get combined. This function can therefore be distributed within the network by intelligent decisions, taken dynamically, below the API. Very similar considerations apply to data serialisation, which is now clearly identified as a network function, with the implementation method and location dynamically determined.

Likewise, the compression hints supplied with the data, allow the network the option of compressing and decompressing data, subject to meeting the quality of service constraints specified by the sending application.

The treatment of analogue data identically with digital data is a recognition of the current state of much of today's voice and video technology. It is helpful to allow analogue applications to be written in the same manner as digital applications, so that the ultimate conversion from analogue to digital is a trivial change of parameters rather than a fundamental application re-design.

Connection and welding of ports allows data flows to be submerged below the API whenever possible. It is not normally possible to handle real-time isochronous traffic if application level software must move data between different communications links, or between communication links and devices. The process of surfacing data up through the communications stack to the application impacts the ability to sustain data rates and avoid excess jitter and latency. The connection mechanism embodied in the proposed architecture allows the application to describe the data flows required, rather than be directly involved in the data flow itself. This permits low level hardware and software coupling of both devices and communications adapters.

References

1. Ensor, J.R., Ahuja, S.R., Horn, D.N., Lucco, S.E., 1988. Proc. 2nd IEEE Conf. on Computer Workstations, March, 52-58

2. Ahuja, S.R., Ensor, J.R., 1992. IEEE Comm. Mag., May, 38-43

3. Knister, M.J., Prakash, A., 1990. ACM Conf. on Computer-Supported Cooperative Work CSCW'90, 343-345

4. Crowley, T., Milazzo, P., Baker, E., Forsdick, H., Tomlinson, R., 1990. ACM Conf. on Computer-Supported Cooperative Work CSCW'90, 329-342

5. Clark, W.J., 1992. IEEE Comm Mag, May, 44-50

6. Ahuja, S.R., Ensor, J.R., Lucco, S.E., 1990. ACM Conf. on Computer-Supported Cooperative Work CSCW'90, 238-248

7. IBM Person-to-Person/2 PRPQ, 1991. Program number 7J0332