and which features should be represented by different segment types.

Applications have a “start of day” problem because until they know the capabilities of some factories they still can not do anything useful. This problem is currently resolved by providing service location facilities in the Medusa C++ libraries which can register capabilities and query a capability database using mechanisms outside Medusa. Ideally this would not be necessary and Medusa will not be truly self contained until access to this database is at least available via a module.

Currently there are polymorphic file source and file sink modules which allow applications to record and play back any stream of messages. This is adequate, but it does not give full access to the facilities of a file system solely from the Medusa world. A better model would be to present files and directories as modules and factories respectively.

Although there are currently no facilities for multi-cast in our ATM network, the Medusa software architecture must be able to use multi-cast when it does become available. The simplest way of representing a multi-cast network connection would be to introduce a new kind of de-multiplexer module which was also a “connection buffer”. This might in turn be hidden behind a layer of proxy modules so that application writers would not need to concern themselves with the details.

7. Conclusions

There are two central ideas in the Medusa software architecture:

- Medusa connections are very simple, so that it is easy to build large, understandable groups of modules. By default reliable unbuffered connections are used, even across the network. Where required, buffered or unreliable network connections are specified using special “connection buffer” modules.
- Security is based on capabilities, with proxy modules used as revocable tokens which give partial and temporary access to sensitive facilities.

These ideas were arrived at after much experimentation. Our aim has been to provide an environment within which new applications can be prototyped very rapidly and new hardware can be integrated into real applications with a minimum of fuss. It is satisfying that this approach has worked well for both creating and running applications over networks in our local area. We must see how far we can go before introducing the complexities needed for a fully distributed system that scales to networks outside our domain.

References

[1] J. Adam, H. Houth & D. Tennenhouse, “Experience with the VuNet: A Network Architecture for a Distributed Multimedia System,” *18th conf. on Local Computer Networks*, September 1993.

[2] G. Blair, A. Campbell, G. Coulson, F. Garcia, D. Hutchinson, A. Scott & D. Shepard, “A Network Interface Unit to Support Continuous Media,” *IEEE Journal on Selected Areas in Communications*, February 1993.

[3] G. Finn, “An Integration of Network Communication with Workstation Architecture”, *ACM SIGCOMM Computer Communication Review*, October 1991.

[4] M. Hayter & D. McAuley, “The Desk Area Network,” *ACM Operating Syst. Rev.*, October 1991.

[5] M. Hayter, “A Workstation Architecture to Support Multimedia,” PhD Thesis, Cambridge University Computer Laboratory, Technical Report 319, January 1994.

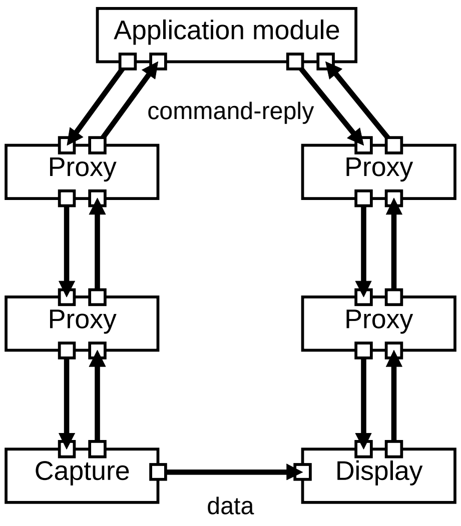
[6] A. Hopper, “Pandora — an experimental system for multimedia applications,” *ACM Operating Syst. Rev.*, April 1990.

[7] A. Hopper, “Digital Video on Computer Workstations,” *Proc. Eurographics*, 1992.

[8] A. Jones, & A. Hopper., “Handling Audio and Video Streams in a Distributed Environment,” *14th ACM symp. Operating Systems Principles*, December 1993.

[9] I. Leslie , D. McAuley & D. Tennenhouse, “ATM Everywhere?,” *IEEE Network*, March 1993.

created in the same process, on the same hardware as the modules they protect. Messages between modules in the same process are very cheap.



Destroying any link in this chain of proxies would have the effect of shutting down all data connections dependent on it, giving the ability to revoke the capabilities of modules handed out to less trusted applications.

5. The system in use

The current implementation of the Medusa system runs on UNIX, Windows NT and ATMOS using C++. Modules are implemented as C++ classes. The behaviour common to all modules is handled by a base class and there are several subclasses which specialise this behaviour and cover nearly all the cases that are necessary when writing a new Medusa module. Given a function written in C or C++ it need only take a few minutes to provide that function as part of a publicly available module.

There is a library containing the code for many standard modules which can be incorporated directly in new Medusa programs for efficiency. These modules provide buffering, video format conversion, simple file I/O, image display and other functions. Modules are also available to handle multiplexing and de-multiplexing of multiple streams, time-stamping and re-synchronising.

Using an extended version of the tcl/tk X toolkit, a number of graphical browsing and debugging tools have been constructed. Tcl is a scripting language for embedding in applications and tk uses this embedded language to control an X toolkit, making the construction of X windows applications very straightforward. A Medusa server has been constructed which is also a tk interpreter, extended with primitives to create, connect and control Medusa modules. The graphical tools written using this interpreter have greatly facilitated the re-use of modules and vindicated our decisions about the architecture.

All the “direct peripheral” ATM hardware mentioned in this paper exists now, linked into a working system by the Medusa software described in this paper. The “ATM Audio Brick” has CD quality quadraphonic input and output with a DSP for filtering. The “ATM Video Brick” can capture video simultaneously at six different resolutions from either its own camera or from an external programme source. These components have been built into a pilot “multi-view video-phone” which has four cameras and quadraphonic sound. The cameras are used for overall views of a room, head-and-shoulder shots and as rostrum cameras. Picture size and so on are selected by the recipient, not the sender, so the recipient can look at that part of a scene which is of most interest. Using the “ATM Storage Brick”, a multimedia RAID file store, it is possible to record and play back “multi-view video mail” in which all of the streams are presented simultaneously, again so that the recipient can decide what to concentrate on.

A good example of the ease with which unusual test programs can be prototyped is the “hand tracking quadraphonic pan control”. This program takes the output of an ATM camera and tracks a bright moving area in the picture. To provide feedback this video stream is displayed on a workstation screen and overlaid with cross hairs to show the deduced hand position. The output of the hand tracker also controls the position of the sound image of a quadraphonic hi-fi system driven by an ATM audio output device. The result is a program that allows you to position a sound image by waving your hands around.

We believe that the ATM devices we have available, from infra-red remote control receivers to auto-stereoscopic displays, coupled with the flexibility of the Medusa software architecture give us a tremendous potential for experimentation.

6. Future work

At present there is one generic audio segment and one generic video segment. It is likely that these will change and it will also be necessary to design segments which handle various forms of compressed audio and video. The problem here is to decide which features should be represented by parameters within a single segment type the corresponding value, but if the command fails none of the attributes are changed.

- The **watch attributes** command is used to monitor changes to the attributes of another module without polling. Its arguments are text segments containing attribute names, and it returns the same information as a get attributes command. However, whenever any of the specified attributes subsequently changes, an event message is sent on the reply connection with the new value of the attributes that have changed. If several watched attributes change at once, the watcher is notified of them all at once in the same event message.
- The **unwatch attributes** command is used to stop monitoring changes to attributes. Its arguments are text segments containing the names of the attributes to stop watching.

All commands generate a matching reply message on the corresponding reply connection. This indicates success or failure in its first segment and further segments in the message will contain results or error information as appropriate.

4.5. Factories and proxies

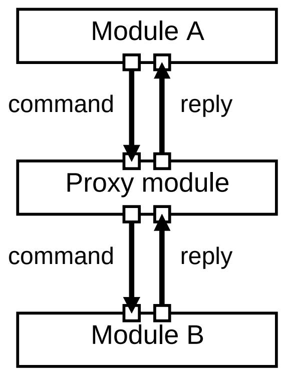
Factory modules are important in the Medusa system because they provide the mechanism for creating new modules. They accept the Create Module command and will attempt to create a module matching the parameters for that command. Factories also have a supervisory role and are often used to expose some physical property of a piece of hardware which will affect all its users. For instance a video camera factory will produce video camera modules each delivering a stream of video images and independently controllable as to picture size, frame-rate and so on. However altering the focusing control on the factory will change the picture for everyone. Factories are too powerful to be exposed to the outside world without some protection and this is where proxy modules come in.

The concept of proxies is key to the security of the Medusa system. Capabilities are the foundation upon which security is built, but proxies are the mechanism by which access is restricted. Anyone can obtain access to a module at any time if they know the capability of the module. In the case of a long-lived module such as a factory, this may not be desirable. Instead, a proxy module is created which simply passes all commands it receives to the factory and forwards all replies from the factory. This allows the user of the proxy full access to the factory while the proxy exists, but allows a session manager to terminate access simply by deleting the proxy.

Two crucial elements in the Medusa system make proxies possible:

- A proxy module can forward connection requests to the module it is protecting.
- A proxy module can monitor changes in the protected module's attributes and reflect them in changes to its own attributes.

Taken together these facilities mean that a module can be written which stands in the command-reply connections between two modules:



As far as module A is concerned, the proxy module is indistinguishable from module B. This fact is used to protect module B, because the proxy can be used as a revocable token giving temporary access to B. Destroy the proxy and A can no longer use B, since it doesn't know B's capability.

A proxy module can be used to protect module B even further, by restricting the operations that module A is allowed to perform. For example the proxy module might make it look as though the attributes of module B were read-only.

In a realistic system there will be several layers of proxies between an application and the bottom level modules which actually implement the multimedia functionality of the capture and display devices. However, because of the forwarding of connection requests the proxy modules can “hand off” requests for data connections down the stack of proxies to the modules which do the real work. Thus data connections are made efficiently, going directly between the real modules without traversing the proxy stacks, even though the commands controlling the capture and display modules are filtered through these layers of protecting proxies. We get efficient data transport while maintaining the protection of the restrictive proxies.

It might be thought that commands traversing these stacks of proxies would be exorbitantly expensive. This is need not be the case, since for efficiency proxies can be connected to is in the same process on the same machine or it is on a far distant machine across the network.

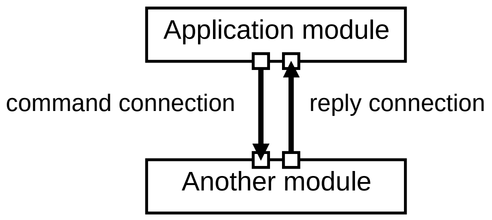
A call-back is used to notify a module when another module attempts to make a connection to it. The call-back tells the module the port name and the direction of the requested connection, but not the capability of the other module. The module must then decide whether to accept the connection request, to reject it or to forward it to another module. Accepting or rejecting are fairly self-explanatory result in the connection request either succeeding or failing at that point.

When a module forwards a connection request it passes it back to the Medusa library in its own process and nominates a substitute port and capability for this end of the connection. An attempt will then be made to connect to that substitute module instead. In its turn, it could accept, reject or forward the connection. There is no way for a module to determine whether a connection was made directly to it or via a convoluted chain of forwarding. It is the forwarding mechanism which makes proxy modules possible, as described in a later section.

It is important to note the difference between the connection requests described in this section and commands described in the next section. A connection request is a library call which a module uses to ask the Medusa library in its process to set up a connection, possibly entering into negotiations with other Medusa processes in order to do this. A command is a special kind of message which is sent by one module to another over a pre-existing connection in order to ask the module at the far end of the connection to perform some action on behalf of the first module.

4.4. Modules and commands

Command connections always appear in conjunction with associated reply connections in the reverse direction, since individual connections are uni-directional.



The task of controlling remote modules is simplified by having a small, fixed set of commands that can be sent to a module. Commands are messages that begin with a command segment and the following segments in the command message form the arguments of the command. All modules accept the same small fixed set of commands.

Every module also has a set of attributes that can be used to control the behaviour of the module and to reflect its state. Each attribute has a textual name and a value which is a segment. Controlling modules is like controlling a hi-fi by moving the knobs and buttons on its front-panel. It is something one might do as the music was playing, unlike re-connecting the components of the system, which one would do more rarely. Some attributes can be set only at the time a module is created, some are read-only and some can be changed at any time. In order to avoid inconsistencies, any number of attributes can be changed in one go, and the whole set will either succeed or fail atomically. For example a video source can insist that its width and height are always equal and it can reject changes that break this rule.

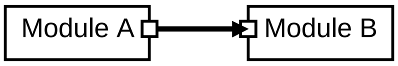
Here is a complete list of the commands accepted by every module:

- The **create module** command is used to ask a factory module to create a new module. Its arguments are a text segment specifying the kind of module to make, followed by segments specifying initial values for the attributes of the new module. If the command succeeds it returns the capability of the newly created module.
- The **delete module** command is used to ask a module to delete itself, or to ask a factory module to delete one of the modules it created. In this second case a capability must be supplied to specify which module is to be deleted.
- The **connect** command is used to ask the module to make a connection to another module. This command takes four arguments: the name of a new port on the module executing the command, the capability of another module, the name of a new port on that module and a direction. If the command succeeds, then when this new connection is subsequently broken an event message will be sent on the reply connection to inform the application.
- The **disconnect** command is used to ask a module to disconnect one of its ports. It takes the name of that port as its argument.
- The **get attributes** command is used to read the values of a module's attributes. Its arguments are text segments containing attribute names whose values are to be returned.
- The **set attributes** command is used to change the values of a module's attributes. Its arguments are pairs of text segments and value segments. If the command succeeds, each of the named attributes is changed to

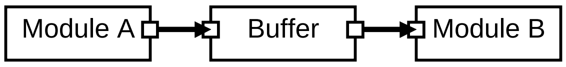
- Connections are reliable. Once a message is sent over a connection then it will either arrive safely at the other end or the connection will be broken. In some circumstances the underlying network connection will be irrecoverably destroyed (the so called "mad axe-man problem") and in those circumstances the Medusa connection over that network will of course be destroyed too.

Although reliability is something which in practice must be added to a network connection to make it useful, in the Medusa architecture all network connections are made reliable by default and if a network connection with different properties is required this is represented by a "connection buffer" module. By choosing an appropriate "connection buffer" an application writer specifies which quality of service or which of a number of protocols should be used on the underlying network connection. From the point of view of application writers the Medusa convention is easily understood: network connections. with the simplest representation are reliable and simple to reason about; network connections with more complex representation may be unreliable and are more difficult to reason about.

For example, a source module could be connected to a sink module in a variety of ways:



The first way is just to connect the source to the sink in the most straightforward fashion. This gives a reliable, synchronous connection between the two modules.



A second way of connecting them is to select a module from the various system-provided buffering modules and to connect an instance of that kind of module in-between the source and the sink. Depending on the precise type of module that is selected the connection between the source and the sink may now be buffered, it may discard data so as to reduce latency, it may be optimised for a specific kind of data and so on. The properties of this new type of connection are reflected in the attributes of the buffer module, which may be inspected and controlled in the same way as for any other module.

Most modules are quite simple-minded about demand. Pure source modules fulfil demand as soon as they can. Pure sink modules issue demand when they are ready for some more data. Pipeline modules, such as format converters and compressors, issue demand when they are themselves demanded of. Then they process the resulting data, pass that on down the pipeline and wait for further demand. Of course a pipeline composed of only this type of module runs completely synchronously, with demand going all the way from the ultimate sink to the ultimate source, followed by a data message retracing this path back to the sink again.

Synchronous working is simple and efficient when modules reside on a single processor, but a pipeline which must deliver real-time performance while running between many machines on a network may need to use "connection buffer" modules to optimise its performance. However it is worth mentioning in passing that, to our surprise, we have found the default reliable connections to be quite satisfactory for 44kHz audio links across our ATM network.

An alternative approach to the implementation of connections would have been to say that they were another kind of active object, different from modules, but with similar facilities for controlling and monitoring their performance. We decided that this would bring more problems than benefits since new operations for dealing with connections would have to be defined and implemented. Using our approach the pre-existing operations used to control ordinary modules could be reused and nothing new needed to be introduced into the architecture. For completeness the alternative approach is also being explored at Olivetti Research as part of the Mobile Distributed Architecture project mentioned earlier.

4.3. Making connections

A module requires two pieces of information in order to make a connection to another module. It needs to know the name of a port on the other module and it needs to know the capability which uniquely names that module. A capability contains network addressing information which unambiguously identifies a process on some machine on the network. This process is the one which implements the functionality of the module corresponding to the capability. So as to make capabilities unforgeable, in addition to the network addressing information they also contain a large random number generated when the module is created.

When a module knows the port name and capability it wishes to connect to, it makes a library call to the Medusa library in its own process. The module must also nominate a local port on itself which will be used for one endpoint of the connection and specify whether it is for input or output. This same mechanism is used to request connection set-up regardless of whether the module to be and plentiful. On an ATM network this is certainly the case, though on other networks it may be less true.

For pragmatic reasons we decided that the system should avoid implementing functionality which could be more easily provided by an application level toolkit. For example, applications have a "start of day" problem because they are not initially connected to other modules and they don't know the capabilities of any other modules. This problem is resolved by providing service location facilities in a toolkit layer. Another project at Olivetti Research called the Mobile Distributed Architecture is exploring service location issues, especially in an environment containing small portable computers. We hope to integrate our work with this at some future time.

4. The architecture

In this section we describe the software architecture in more detail. Starting with the representation of data as segments we then go on to describe connections, commands and proxy modules.

4.1. Segments and messages

A segment is the atomic unit of data in the Medusa software architecture. The format of segments is principally a network format, specifying the layout of data to be transferred between machines. In our current implementation of Medusa this network format is transcribed directly to and from buffers of contiguous memory which are used as the internal representation of segments. This choice of internal format was made for reasons of efficiency and convenience in our implementation language, C++, but in other languages the internal representation could be quite different. The only thing that must stay the same between implementations is the network format. Polymorphic modules, which are happy to deal with messages containing any type of segment, can treat all the segments they see as “raw data” without bothering about their types at all.

There are the following types of segments in Medusa:

- **Text** segments contain a null-terminated string of ASCII characters.
- **Integer** segments contain an array of signed integers.
- **Capability** segments contain the capability for a module.
- **Mark** segments contain no data. They are used as delimiters in messages.
- **Time** segments contain a time value.
- **Audio** segments contain a byte string which encodes the audio data and also parameters describing the sample rate, encoding scheme and so on.
- **Video** segments contain a byte string which encodes the video data and also parameters describing the width, height, pixel format and so on.
- **Command** segments contain an integer identifying one of the small number of commands in the system. These are used to identify command messages. Further details are given in section 4.4.
- **Reply** segments contain a status code for the result of a command. These are used to identify reply messages.
- **Event** segments contain a status code. They are used to identify event messages.

In the case of command, reply and event messages the command, reply or event segment must be the first segment in the message.

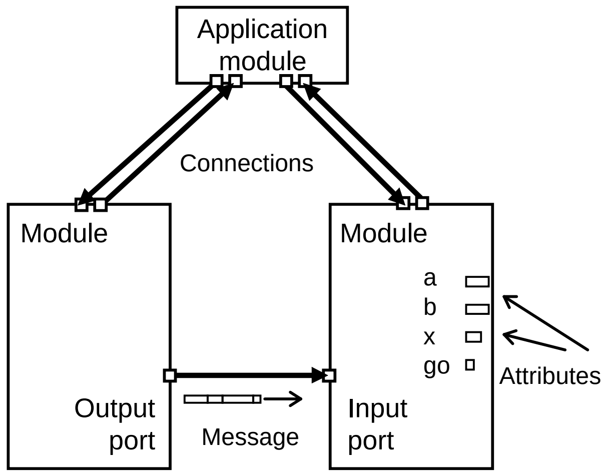
4.2. Connections

Connections are very simple and any sophistication such as buffering, unreliability or asynchronicity is represented by a module. Connections join a port on one module to another port, usually on a different module. Once a connection has been made to a particular port, no other connection may be made to that port until the first connection has been removed. It is important to note that connections are made between two ports as a single operation: there is never any time at which only one port is attached to a connection. A connection *is* the attachment of one port to another. The most important feature of a connection is that it is location transparent. The communicating modules do not need to know whether they are on the same machine or separated by a network. Connections have the following properties:

- Connections are not typed. All connections are equivalent and all connections can carry any kind of data that can be represented in a message.
- Connections are synchronous and demand driven. Data must be demanded by the sink before the source can send a message to the sink. The sink does not demand again until this outstanding demand has been fulfilled. When the source is ready it sends a message to the sink and waits for the next demand. Implementations of this scheme are simple and can be made to run over any network. This is an important consideration where rapid prototyping is an issue.
- Connections are one-way: data flows from the source module to the sink module. Ports are either input or output and there must be one of each for a connection to be made.

data itself. A peer-to-peer architecture allows more flexibility when deciding how to structure the flow of data in individual cases.

Here is a glossary of the principal components of the Medusa software architecture:



- **Modules** are the active objects in the Medusa software architecture. They are the sources of data, sinks of data and processing elements within the system. All active objects within the Medusa world are modules, even applications.
- **Attributes** are variables local to each module, through which they expose some of their internal state to other modules. Other modules may inspect these attributes, modify them, and may ask to be notified when they change.
- **Capabilities** are unique global names for modules. Capabilities contain network addressing information and large random numbers. The addressing information allows connections to be made to their corresponding modules, and the large random numbers make them unforgeable.
- **Segments** are the atomic units of data representation. Segments are typed and may contain strings, integers, pieces of audio or video, time stamps, commands and so on. The values of attributes are segments.
- **Messages** are the units of data transfer between modules. A message is a sequence of one or more segments.
- **Connections** allow one module to communicate with another. Connections are not typed, but the messages flowing through them are composed of typed segments. Connections are unidirectional and data transfer across connections is demand driven.
- **Ports** are the endpoints of connections. Modules have collections of named ports which they may create and delete dynamically. Ports are either input or output, but other than this restriction on direction there is no constraint on the kind of data flowing through a port.
- **Commands** are messages whose first segment is a command segment indicating the operation to be performed. The set of commands is small and fixed and these commands are understood by all modules.
- **Factories** are modules that can create other modules when asked to by a command.
- **Buffers** are modules interposed in the data connection between two modules which provide a more sophisticated connection semantics than the simple demand-driven semantics that a direct connection would impose. Most buffers are polymorphic, that is they are happy to buffer messages containing any type of segment.
- **Proxies** are modules interposed as a “fire-wall” in the command-reply connections between an untrusted module and a sensitive module. The proxy is indistinguishable from the sensitive module except in the case of certain forbidden operations, which will fail. If the proxy is destroyed, the untrusted module has no further control over the sensitive module.

We decided that connections between modules should be reliable, so that the same mechanism could be used to transmit both commands and multimedia data. Although commands and data always travel over different connections in practice, we felt that this approach was justified for the pragmatic reason that it was more straightforward to implement one mechanism than two. When network connections which expose the workings of unreliable networks are required, they can be specified using “connection buffer” modules, described in a later section. For “off-line” working reliability is vital for both data and commands. Consider, for example, an image compositor in a video editing suite. Its real-time performance is secondary: the main thing is that it combines images reliably. Even in live applications, such as video-phones, reliable connections are desirable when certain kinds of compression such as MPEG are in use.

Wherever possible different media are handled in a unified way. For example switching, synchronisation and storage do not in general require precise knowledge of the media type, so polymorphic modules are provided which perform these common functions and the modules can be re-used in a wide variety of rôles.

It was always intended that Medusa would run in an ATM environment and this led to a number of assumptions about the underlying network. In particular we assume that a global addressing scheme exists, so anything on the network can connect to anything else. We also assume that network connections are simple, cheap component in a new application. Indeed, from the point of view of an application designer it is not necessary to draw any fixed line between hardware and software components. A component can be prototyped in software and then replaced by hardware, the only difference being speed of operation. Alternatively, a hardware component could be replaced by software to reduce the cost of a system or to take advantage of a fast new general purpose processor.

2. Design goals

Considering the design of Pandora [6,8], an earlier networked multimedia system built at ORL, we highlighted certain goals whose attainment would be necessary for the success of a more open system. The Pandora system consists of several UNIX workstations each with an attached peripheral which handles all ATM network traffic and all audio and video for the workstation. The video images are inserted into the workstation's display by this peripheral using analogue switching techniques. We gained valuable experience from building and using the Pandora system. However, the Pandora software architecture is not suitable as a foundation for a more open system.

In the Pandora architecture it is easy for an application to set up multimedia connections from one place to another, but hard for an application to manipulate the multimedia data itself. Applications must also know too much about whether devices are attached directly to the network or to a local bus. The Pandora software architecture is good for interactive applications where latency of multimedia data is the main concern, but very poor for off-line working where reliability is more important. Because the Pandora hardware is basically a single box attached to the network, it is difficult to add to the system incrementally and heterogeneously. Perhaps the most fundamental problem with the Pandora architecture is that there is no access control, and in fact no security mechanism at all. With all this in mind we decided on the following goals for the Medusa software architecture:

- Medusa must be able to integrate data from the heterogeneous hardware being developed at Olivetti Research. This means that the system has to be able to run under UNIX, Windows NT and ATMOS (our in-house micro-kernel for embedded systems). This in turn leads to a requirement for a simple and lightweight implementation.
- Medusa must provide security mechanisms so that users will have confidence in the multimedia applications. In the same way that telephones are seen as assets, not threats to privacy, we must ensure that our networked microphones and cameras are assets.
- Medusa must support the processing of multimedia data in software. This allows applications to manipulate multimedia data as well as controlling its transfer. This also means that it is easy to move between hardware and software implementations of a particular functionality.
- Medusa must facilitate the re-use of hardware and software components so that it is easy to build supporting structures for new applications. If these structures are hard to build, new ideas can be discarded untried because it is too much effort to test them out.
- Medusa must be able to support applications involving agents as well as applications such as video-phones which provide direct communication between people. Reliability of the data connections appears to be the most important thing where agents are involved, in contrast to direct human interactions where low latency is the key. However, both classes of application have a great deal of functionality in common and re-use of components between them must be easy.
- Medusa must be able to handle multiple streams effectively. We anticipate that dozens of streams will go in and out of individual offices, and eventually the whole Medusa system at Olivetti Research will be called on to handle thousands of streams simultaneously.

3. Overview

The goals of a lightweight implementation and rapid prototyping immediately led to the idea of the active objects which we call modules. The Medusa system provides a large number of simple modules which may be combined to produce the structures of control and data flow required by individual applications. Modules will usually be implemented as threads or co-routines within server processes on the different machines on a network. For example the modules representing and controlling an ATM camera are implemented on the ATMOS embedded processor attached to that camera.

Medusa has a peer-to-peer architecture rather than the strict hierarchy of a client-server architecture with call-backs. This is because one application may wish to take a “hands on” approach and inspect or modify data as it flows from a source to a sink, while another application may wish to take a “hands off” approach and simply connect a source to a sink without ever touching the raw

The Medusa Applications Environment

Stuart Wray†, Tim Glauert† & Andy Hopper†‡

†Olivetti Research Limited,
24a Trumpington Street,
Cambridge CB2 1QA, England

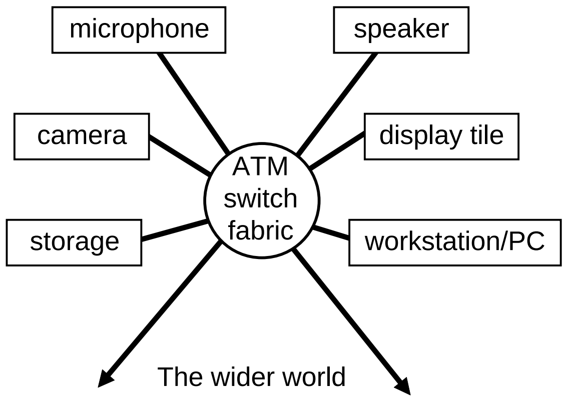
‡University of Cambridge Computer Laboratory,
New Museums Site, Pembroke Street,
Cambridge CB2 3QG, England

Abstract

Medusa is a peer to peer architecture for controlling networked multimedia devices. This paper describes the software model presented to the applications programmer. Active objects called modules are used to represent cameras, displays, format converters and so on. Data can flow from module to module on connections between them. We introduce two key ideas: firstly, connections between modules are simple, reliable and unbuffered. More complex connections are represented by special intermediate modules. Secondly, security is provided by naming modules with unforgeable capabilities then using hierarchies of proxy modules to restrict access. Keywords: multimedia, distributed systems, programming paradigms, ATM networks

1. Introduction

The Medusa project at Olivetti Research aims to provide a networked multimedia environment in which very many streams, perhaps thousands, are active simultaneously.



We envisage that most of the traffic in our system will not involve direct human interactions but will instead involve agents whose purpose is to examine video and audio on behalf of their human clients. For example there will be agents for face recognition, for interpreting gestures and spoken commands, and so on. These agents will often have real-time requirements which differ from those of direct human interaction, and it is important that all these different requirements are supported equally well by our hardware and software.

The hardware upon which the Medusa project is based is a collection of "ATM direct peripherals", including cameras, microphones, loudspeakers and multimedia file servers, as well as ATM networked workstations. Each of the direct peripherals is a small computer built around an ARM processor which is connected to an input or output device and to the ATM network [7,9]. Each of these computers runs an in-house micro-kernel called ATMOS which was designed specifically for such networked embedded real-time systems. In more traditional multimedia systems the input-output devices would be attached to a workstation bus and the workstation would be connected to a network. In our approach, the input and output devices are separate components, each independently connected to the network.

A similar but slightly less radical approach has been taken by a group which has built clusters of tightly coupled input and output devices around standard workstations [2]. Other groups [1,4,5] have been even more radical than us, dividing the workstation itself into separate components, with CPUs, caches and memories all directly connected to the ATM network. We have stopped before this point, preferring instead to concentrate on providing a wide variety of input and output devices and the software infrastructure to rapidly prototype applications using them.

With all the ATM direct peripherals under the control of the Medusa software it is easy to replace an old component with a new component or to re-use an old