Proceedings of the 1st Workshop on Multimedia Software Development, Berlin, March 1996

A Software Platform for Distributed Multimedia Applications

Christian Blum and Refik Molva
Institut Eurécom
2229, route des Crêtes
F-06904 Sophia-Antipolis
{blum,molva}@eurecom.fr

Abstract

The paper discusses the functionality that a platform for distributed multimedia applications should provide. Five platforms - Touring Machine, Beteus, Lakes, Medusa and IMA - are presented and evaluated. The paper then introduces a new platform that is geared towards service provision in larger networks. In this platform, application and media processing are logically and per default geographically separated. Applications reside in application pools inside the network and control sets of multimedia terminals in the periphery. The platform is built on top of CORBA, and is thus defined as a set of IDL interfaces. The platform supports application development with high level programming interfaces, and can be further enhanced with tool-kits for special application classes. A tool-kit for teleconferencing applications is presented at the end of the paper.

1. Introduction

Now that distributed multimedia applications are starting to be offered as services in enterprise and residential cable networks, there is a growing interest in platforms that support development and deployment of these applications. Platforms have to be seen as opposed to standalone applications that are found for instance in research environments. Such applications implement media processing from scratch and are highly dependent on hardware and operating system. They are built for a special purpose and require skilled personal for setup and usage because they are not integrated into a general runtime environment. While this is the normal approach to build research prototypes, it is clear that applications that are offered as services have to be integrated into the network in which they are to be offered, i.e., they have to be built on top of runtime platforms.

Multimedia data processing is as modular as it is complex. Media streams lend themselves to pipeline processing by sharply defined building blocks like coders, mixers or network transmission units. An application will profit from such building blocks in that more time can be spent on design and implementation of higher-order features. This paves the way to true multimedia applications whose originality is not the single medium, but the way different media or media building blocks are assembled to form increased value. The granularity with which building blocks are defined determines the range of applications a platform supports, and it determines also the complexity with which application development is faced. The benefit of fine building block granularity is application diversity, but application programming is likely to be complex.

Platforms for distributed multimedia applications give access to reusable media processing building blocks via high-level programming interfaces. They impose an application model, a way an application is organized and implemented. A good application model guides application design without confining it. Programming interfaces and application model support rapid prototyping and incremental improvement of applications and move the application itself rather than its components into the center of attention.

Platforms allow multiple applications to run in parallel. They integrate applications in a standard, application-independent way. Applications are launched, joined or deleted via procedures that are built into the platform rather than into the application itself. Platforms will also give applications access to an extensible set of network services like network management, directory and billing.

This paper shall provide a general overview of multimedia platform issues. It starts off with a discussion of the major features a platform should provide (Section 2). It then presents five platforms, the Touring Machine of Bellcore, the Beteus platform developed at Eurécom, Lakes by IBM, Medusa by Olivetti and the multimedia system services architecture by IMA (Section 3). The

paper continues with the detailed description of a multimedia service platform that is developed at our institute (Section 4-6). Our platform separates application and media processing not only logically, but also geographically. Applications reside in the network within so called application pools from where they control a set of participating multimedia terminals. The architecture is geared to service provision because it provides a terminal with a clearly defined and extensible interface that can be remotely controlled by a wide variety of applications. Toolkits can be added to the application pool that provide high-level interfaces for application development. An example for such a toolkit, a toolkit for the development of teleconferencing applications, is described at the end of the paper (Section 8).

2. Platform Evaluation Criteria

Before we move to the description of the example platforms we want to introduce the criteria according to which we classify these platforms. The list of criteria that is presented here is certainly far from being complete, but it covers the most important aspects of the platforms that we present.

Target applications - platforms are designed and built with certain applications in mind, and no platform can claim that it supports every single distributed multimedia application.

Target network - platforms are built for certain classes of networks, e.g., local area networks, ISDN networks, or residential cable networks. There are also platforms that run on heterogeneous networks.

Application model - every platform imposes its personal application model on the design of the application. As an example, some platforms will require the application to be centralized, whereas others require it to be completely distributed. Distributed applications will use the communication features provided by the platform as basis for application-specific protocols.

Abstractions - every platform will try to define abstractions and use them as basic concepts of programming interfaces or protocol procedures. Abstractions are fundamental for the design process of a platform, and they consequently mirror the internal architecture of the platform. Abstractions are defined with a limited set of applications in mind they must support. Well chosen abstractions will survive more than one application generation, whereas poorly chosen abstractions will quickly tear both the programming interface and the platform into obsoleteness.

Configurability - platforms generally offer their media related functionality in form of building blocks. These building blocks can have varying granularity. As an example, one platform may offer video sender and video receiver as the smallest video related building blocks, whereas another one will decompose these blocks further, for instance into coder, transmitter, receiver and decoder. The finer the granularity of the building blocks is, the higher is the freedom with which an application can configure its media processing part. The drawback of fine granularity is increased application complexity. A platform should therefore ideally offer both, fine and coarse building block granularity.

Extensibility - a platform that is not explicitly designed for extensibility will soon be overrun by technological progress. A platform does not only have to be able to accommodate new media processing devices, but it must also have an extensible service interface towards its applications.

Scalability - does the performance of the platform decrease when the number of potential users, actual users, offered applications and running applications increases?

Synchronization - every platform must address the problem of inter-stream, intra-stream and event-driven synchronization [2]. Intra-stream synchronization will not be visible at the platform interface, but there will be abstractions for inter-stream and event-driven synchronization in the programming interface of the platform.

Resource management - a platform must interface to the operating system and possibly to the network in order to reserve resources for media processing and transmission. Important resources are transmission bandwidth and processing time. Resource management is visible at the platform interface level as soon as the application itself is the endpoint of a medium stream. In this case an application must request a certain quality of service for the media stream that it transmits or receives via the platform.

Security - security is an upcoming issue for multimedia communication in general. Platforms for distributed multimedia applications will have to integrate security services like authentication and access control.

3. Example Platforms

The platforms that are presented here are evaluated according to the criteria developed in the last section. For

further information about these platforms there are pointers given at the end of the paper.

3.1. Touring Machine

No overview of multimedia platforms can overlook the Touring Machine developed at Bellcore [3]. While antiquated with respect to today's standards, it was the first platform for distributed multimedia applications having a real application programming interface (API) [4]. The target applications of the Touring Machine are teleconferences with collaboration support. The Touring Machine was developed for a hybrid network with analog switches and analog endpoint equipment for audio and video. Applications are distributed with a typed platform *client* running for every participant. The programming interface offers an interclient messaging service for the development of application-specific protocols. Applications run within the framework of a *session*, with a session being able to support more than one application, in which case it contains as session members clients representing different applications. Further major abstractions are *connector*, *endpoint* and *port*. A connector is a medium bridge that can be of type audio, video or data and that has source and sink endpoints from participating clients attached to it. Connectors can represent point-to-point as well as multipoint connections. Clients access connector endpoints via ports. The port abstraction allows clients to switch locally between different sources or sinks for the same connector endpoint.

Audio and video presentation at sinks is uniform due to the underlying analog network. The only degree of freedom in medium transmission is therefore the connection structure. Since audio and video is live, and since there is no remarkable delay introduced on the transmission path, there is no need for synchronization between audio and video. Resource management is mainly concerned with analog connection establishment and routing within the network.

The API of the Touring Machine is monolithic and not designed for extension. It has functions for: client registration, session management, connection control, name server query and message passing.

3.2. Beteus

The Beteus platform was developed in the course of a tele-teaching project on the European ATM pilot network with project partners in France, Switzerland and Germany [5]. Beteus applications shall provide a wide range of support for the education of a group of students scattered over a small number of *sites*. Possible applications are distributed lecture, tele-seminar, tele-tutoring and tele-meeting. The decision to implement a platform instead of standalone applications was due to the uncertainty at the beginning of the project about which applications would finally be needed or would best promote the underlying broadband network.

Connection control and resource management are distributed inbetween Beteus sites, but centralized on site level because Beteus applications require in general the combination of equipment beyond that offered by a single multimedia workstation. The notion of a site exhibited by the platform is not visible to applications. A Beteus Application is composed of multiple processes called *session vertices* that are arbitrarily distributed over the sites. An application is implemented as a single executable that will take one or more *roles* within a session. The behavior of a session vertex within a session and the user interface that it generates depends on its momentary roles. There is one session vertex holding the *master role*; the session master is the sole session vertex being allowed to change session membership or connection structure. Note that this functionality is not necessarily visible at the user interface that the session master generates.

An application defines the connection structure it generates in form of multipoint audio, video and application sharing *bridges*. The endpoints of a bridge are given as role names, and bridges are bundled into *bridge sets*. The session master configures the platform on session startup with the roles, bridges and bridge sets that the application defines. Connection management during the session is then limited to role transfer and bridge set swapping. The platform resolves role names to session vertex addresses and rebuilds automatically connection structures when role assignment changes.

Application development on top of the Beteus API [6] is mostly user interface development. Communication between session vertices and the platform is based on Tcl-DP [7], the network programming package for Tcl [8]. This allows applications to be completely implemented in Tcl scripts, which is ideal for rapid prototyping. The Beteus API has functions for: registration, media endpoint control, directory service, session control, role/bridge/bridge set handling, message passing and application sharing. It offers a more sharply defined application model than the Touring Machine, and much more control over media transmission and presentation. The Beteus API is, similar to the one of the Touring Machine, monolithic and not designed for extensibility. The platform does not scale to a large number of sites.

3.3. Lakes

IBM Lakes is, as a commercial product, the most complete platform among the ones presented here. Lakes is

designed for collaborative applications that may optionally employ multimedia communication. Target network is the enterprise network, whereby Lakes supports analog as well as digital links among *nodes*. The platform is distributed with one instance of it running for every user. Applications are realized as independent processes that may temporarily come together in a *sharing set*. Applications within the same sharing set may establish *channels* among each other, whereby channels represent connections among application ports with explicit quality of service attributes.

Channels can be combined to *channel sets* of type *standard*, *merged*, *serialized* or *synchronized*. Merged channels are channels that terminate in the same port, supporting multicast within a sharing set. Serialized channels serialize data from different sources, so that every sink sees the same ordered sequence of data packets. Synchronized channels finally deliver packets with the same timing relationship with which they received them. Channels are terminated by application ports or *logical device* ports. Applications realize audio or video connections with channels between audio and video logical devices.

Lakes has a rich set of interfaces: a programming interface, a command level interface, a resources interface, a device support interface, and a logical device interface. The command level interface is a high-level interface to the call manager furnished by IBM and allows to develop custom call managers. The resources interface allows to customize resource management, and the device support interface to port the platform. The logical device interface supports platform extension with user-written logical devices.

Lakes supports theoretically, and probably unintentionally, the chaining of logical devices, which is a key feature for configurability.

The Lakes API consists of more than 50 function calls and about the same number of events. The strength of Lakes is certainly the richness of its interfaces, but Lakes remains a proprietary solution that will have a hard time to be accepted by a large base of developers. Also, its function based API is outmoded given the actual tendency to object-oriented interfaces.

3.4. Medusa

Medusa [10] is the first platform presented here that is explicitly designed for extensibility. It is built for an ATM network that supports the direct connection of media processing hardware. Medusa workstations consist of a standard workstation plus multimedia devices that are grouped around a small ATM switch which is itself connected to an ATM backbone. The so-called desk area network (DAN) architecture allows to add an arbitrary number of direct ATM peripherals to the workstation switch without running into performance problems.

The main abstraction of Medusa are *modules*. Every active object, including the application itself, is a module. Modules usually represent some clearly defined function like video compression or audio source and are intended to be chained together to form pipelines from source to sink. Modules have *ports*, and the ports of different modules are linked via oneway *connections.* Connections are untyped, but carry data *segments* that can be of type audio, video, command, reply, event and so on. Connections are thus used for both control and media data.

A Medusa application will ask *factory* modules at different locations in the network to create modules and will interconnect them. The Medusa developers have written an extension to Tcl/Tk that allows to put together module pipelines within Tcl scripts, which in turn supports applications that are completely implemented in Tcl. There is also a special application, a graphical programming tool, that allows to assemble and configure pipelines in real-time.

Simple Medusa modules can be implemented with a few lines of C++ code. Every application developed for Medusa will enrich the platform by increasing the number of modules available to factories. Second generation applications will then have access to a rich library of modules that they can assemble in many different ways. A problem with the Tcl based application programming interface of Medusa is that programming becomes tedious when a large number of modules is used [11].

The Medusa platform addresses security by providing access restrictions to factories and instantiated modules. Unlike the platforms presented until now, Medusa does not offer any session abstractions, but note that such abstractions can be easily layered on top of it. The platform and platform abstractions will support multicast only in a future version.

3.5. IMA

The multimedia systems services architecture defined by the Interactive Multimedia Association [12][13] is a layer of abstraction constructed above the multimedia relevant hardware and software resources of a distributed system. As such it constitutes a framework of middleware rather than a platform. It is thought to be used as the media control part of platforms that include in addition functionality that IMA does not address, namely security, toolkits, scripting, user interfaces and application sharing.

The IMA architecture is defined on top of the Common Object Request Broker Architecture (CORBA) [14] and is given as a set of IDL (Interface Definition Language) interfaces. It is natural for a platform for distrib-

uted multimedia applications to be built on top of another platform that supports distributed applications in general. The IMA architecture profits from present and future developments in CORBA like the event service and can directly employ them. The architecture employs both interface inheritance and inclusion: inclusion to reduce complexity, and inheritance to provide for extensibility. IMA will only define interfaces within a basic inheritance tree; the richness of the architecture is then to be provided by third-party architecture extensions.

IMA decomposes media stream processing into *virtual devices*. A virtual device contains a device-specific interface, a generic *stream* interface and one or more *format* interfaces per device *port*. The stream interface allows to control a medium stream (pause, resume, fast forward and so on) and is an interface found also in other objects. The format interface allows to set the medium format generated by a port. Two device ports are connected via a *virtual connection*, an object that has again a stream interface. A virtual connection can be unicast or multicast.

An application can combine the objects that it generated into a *group*. A group applies operations on its stream interface to all the stream interfaces that it controls, simplifying application action for stream control on device networks to single operations.

In line with CORBA, objects in IMA are created by *factories*. An application specifies constraints for the objects that it generates, with the location of the object being one prominent constraint. The factory matches constraints against object properties and creates an appropriate object if possible.

Other features of the IMA architecture are: support for synchronization, support for both stored and live media, and a refined event service. A key feature certainly is the variable degree of application involvement in device network assembly and configuration. An application specifies the object attributes it is interested in as a set of constraints, and leaves the remaining attributes to the discretion of the infrastructure. Looking at the current draft of the standard, it is hard to see how this will be done for a complex multipoint connection including many endpoints and many devices per endpoint, resulting in a proliferation of negotiating virtual connection objects. The IMA architecture generally neglects the cost of a remote procedure call, and will certainly stress most of today's commercially available object request brokers. One might cite other problems with the architecture, but it is generally agreed upon that future platforms for distributed multimedia will be at least similar in spirit to IMA, if not compatible with it.

4. Application Pool and Multimedia Terminal

The platform architecture that is presented in the rest of the paper addresses the problem of how to provide multipoint multimedia services to a large community of users, with these users being for instance private households or the employees of an international enterprise. The design decisions taken for our platform were based on the following three assumptions about future multimedia services:

- there are many services offered in parallel
- most of the services have short life-cycles
- there are many service providers

The amount of services offered on the Internet today gives a hint about the service diversity that tomorrow's multimedia networks will have to support. This does not only concern the retrieval and other man-machine services that will be offered in first generation residential networks, but also the multipoint communication services that will come up as soon as the subscriber lines become symmetric. The number of services will be enormous, and services will have short life-cycles. They will be rapidly developed and deployed, and once deployed they will be constantly improved up to the point where they become obsolete. The lifetime of a service will be governed, among other things, by commercial interest, by fashion, and by technological progress. Service diversity can only be attained in a competitive environment where service provision is open to everybody. The first service providers in residential cable networks will be the network providers themselves, but it can be foreseen that they open their networks to third parties in order to offer a richer variety of services to their customers. A good example for a functioning community network is the French Minitel. The success of Minitel is largely due to its service diversity, which in turn is a result of France Telecom's politics of opening the network for service providers. Today, around 24600 services can be accessed via a Minitel terminal [15].

Service provision in a network requires the existence of standard terminal equipment at the user's premises. While this is also the case with multimedia service provision, it is clear that the terminal equipment there will need to be more intelligent than for instance the Minitel terminal. A crucial problem is the distribution of application intelligence. One might be tempted to think that multi-point applications like tele-conferences will reside and run on end-systems, and that the only service required from the network is connectivity. The problem with this approach is that it limits diversity simply because people would need to install on their endsystem every application they possibly want to use. Another possibility would be to have application servers in the network from which user endsys-

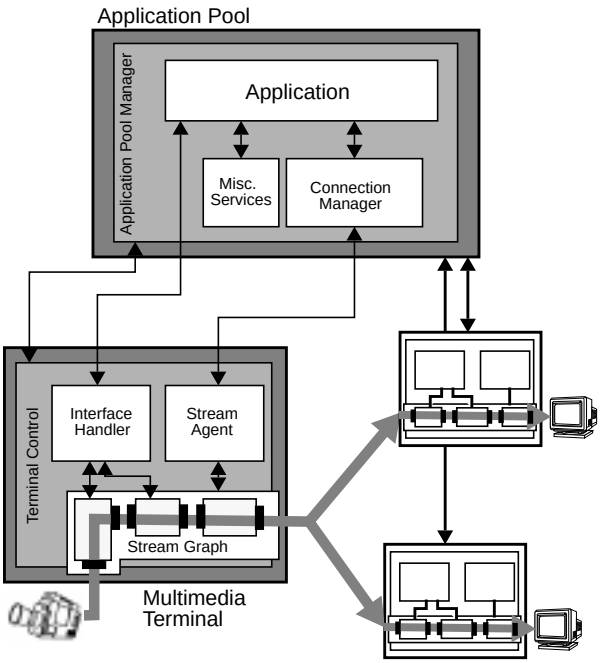

Figure 1. The APMT architecture.

tems can download the executables they need to run a certain application. This approach has the problem that the application, once developed, has to be ported to every possible endsystem architecture, and that the application server would need to store all of these executables.

The approach we chose is to distribute application intelligence between the network and the endsystems. Applications reside on *application pools* inside the network and are accessed by means of *multimedia terminals* [1]. An application will download scripts into the terminals that serve as intelligent sensors for the application and that deal with every issue that is local to the terminal. Connections among the terminals that participate in a multipoint application are established by a central connection manager within the application pool that acts on behalf of the application. The application pool must be considered as a center of control, and will rarely be the source or sink of media data. Media acquisition, transmission, processing and presentation is performed by standard hardware and software devices within the terminals. The application controls the devices and receives the events that they generate. A terminal can activate a certain application only if it has the devices that the application requires. The architecture of both the application pool and the multimedia terminal is device independent, i.e., new devices can be introduced without any modification of the major building blocks of the architecture.

The complete architecture is based on CORBA, and is thus defined as a set of IDL interfaces. There are roughly two kinds of interfaces: interfaces that define the interaction between terminal and application pool, and interfaces that are internal to application pool or terminal. The internal interfaces of the application pool must be known by application developers, the ones in the terminal by device developers, and the interfaces between terminal and application pool by both application and device developers. Internal interfaces may need to reflect hardware and operating system particularities and are therefore system dependent. It is evident that external interfaces must be unique.

Fig. 1 shows the major components of the application pool and multimedia terminal along with control and media flows. For convenience, we will refer to our architecture as the APMT (Application Pool - Multimedia Terminal) architecture. The following subsections give an overview of APMT.

4.1. Overview of the multimedia terminal

The brain of the multimedia terminal is the *terminal control*. The terminal control manages the application life-cycle on the terminal side: it starts and joins applications in the application pool on behalf of the user, and processes invitations to applications. It grants applications access to the major terminal interfaces and supervises application actions within the terminal. Every major object created by the application has a hidden interface to the terminal control which allows it to be queried, monitored, and deleted.

The operations defined for the terminal control interface constitute together with equivalent interface operations in the application pool an application control protocol. Since this protocol is application independent, it can be expected to remain stable over an extended period of time. Protocol extensions, and the eventual existence of different protocols for different terminal types, can be handled via interface inheritance.

The two principal servers an application accesses are the *interface handler* and the *stream agent*. An interface handler executes a script downloaded from the application. This script generates the graphical user interface of the application and controls the locally generated device networks. As a result of user action it will call operations in application interfaces, and will itself respond to application calls. An adequate scripting language for simple tasks is Tcl/Tk. If the downloaded script is to perform more advanced tasks than the user interface, strongly typed languages like Java [16] or ScriptX [17] must be used. Java and ScriptX perform better than Tcl/Tk because they are precompiled. The major requirement on the scripting language to be used is the existence of a respective CORBA language mapping. The multimedia

terminal will have separate interface handlers for every scripting language that it supports.

A stream agent assembles, controls and modifies *stream graphs*. A stream graph is an arbitrarily structured network of media processing devices similar to the module pipelines of Medusa or the virtual device graphs of IMA. Stream graphs are generated in single operations that return a list of device object references. An application may forward some of these references to the interface handler for local control, as is indicated in Fig. 1. A straightforward example for this would be the reference to an audio device that allows the terminal user to control audio volume via the graphical interface generated by the downloaded script.

A terminal will have other components with interfaces hidden from the outside, among them for instance a resource manager. Interfaces like the resource management interface have to be taken into account by the device developer.

4.2. Terminal interfaces and stream formats

The terminal interfaces are the low-level programming interfaces in the APMT architecture: the terminal control, the interface handlers, the stream agent, the graph, and the graph components. In addition to these control interfaces, there is a format to be specified for every kind of medium that is communicated between terminals. Such stream formats are also foreseen in the IMA architecture. In APMT, we specify stream formats and header formats. Headers serve inter-device communication and can be arbitrarily added to a stream. This allows to use the medium stream itself for oneway in-band control, relieving the application or other architecture components from information transfer inbetween source and sink graphs.

4.3. Overview of the application pool

The counterpart to the terminal control in the application pool is the application pool manager (ApMgr). The ApMgr launches applications on behalf of terminals, and invites terminals on behalf of applications. The ApMgr grants applications access to the application pool objects and monitors them. Like in the case of the terminal, there will be hidden resource managers within the application pool from which applications will request, among other things, storage place and computation resources.

Applications can access the terminal interfaces directly or via intermediate modules that reduce the complexity of multi-user scenarios. One such module is the *connection manager*. The connection manager (CxMgr) provides support for the establishment of complex connection structures among groups of terminals. An application will usually prefer to deal with one connection manager rather than n stream agents. Multiple connection managers can be imagined that provide application support at different levels. The CxMgr interface is internal to the application pool, and therefore not canonical, but it must be a visible part of the architecture because it is fundamental to application portability.

Similar to the CxMgr, a module can be imagined that handles multiple user interfaces, maybe even in coordination with the CxMgr. An application that represents a service will certainly access additional services like video server control, billing or network management. Such services fall into the category miscellaneous services, which is indicated in Fig. 1. The application pool is, like the terminal, open for any kind of extension.

4.4. Additional architecture components

The APMT architecture provides a basic framework for multimedia service provision. Depending on the actual deployment of the architecture, there will be additional architecture elements within the network. One such component would be for instance a service gateway that transparently routes service requests to application pools. This allows for load balancing on application pool level. Another important component is a directory service. A directory service would be accessed by both terminals and applications.

4.5. Deployment

The APMT architecture addresses multimedia service provision in a wide sense. An application pool can be located somewhere in a network on a cluster of dedicated machines. It can equally be found together with terminal software on a personal computer in a private household. An application pool is visible in a network via the name or object reference of its ApMgr interface. Whoever wants to export an application may do so by running application pool software and promoting the ApMgr object reference.

An application is not necessarily a single process. The only restriction on the way an application organizes itself into processes is that one of these processes implements a control interface towards the ApMgr.

A terminal is represented by the object reference of its terminal control. A multimedia terminal is not necessarily a personal computer. It can equally be a whole range of equipment forming a classroom for distributed lectures.

5. Terminal Building Blocks

This section and the following one discuss the building

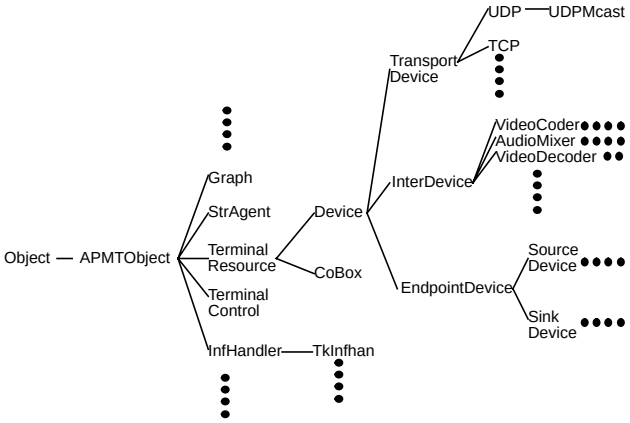

Figure 2. Terminal interface inheritance diagram.

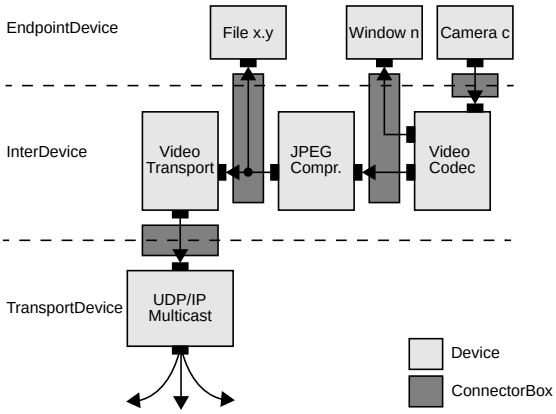

Figure 3. Example device graph.

blocks of the platform prototype we are currently implementing at Eurécom.

5.1. Terminal control

The terminal control is a process that implements four interfaces. First of all there is an internal interface that is used by terminal processes to make themselves known to the terminal control and to convey activity events. There is another internal interface meant to be accessed by a control panel process that generates a service control menu towards the user. Part of the same functionality is offered in a third interface that can be accessed via locally active application scripts. This allows for instance a user interface script of a service yellow page application to directly launch a chosen service. The user of the terminal is notified of such application script actions via events or callbacks sent to the control panel. Since this interface is visible to application scripts it belongs to the category of external interfaces. The fourth interface is the one to the application pool that was already introduced.

5.2. Graphs, devices and connector boxes

The media processing related abstractions of APMT are influenced by IMA, but differ from them in important points. Fig. 2 shows the branches of the APMT interface inheritance diagram that define the terminal. The elementary unit of processing functionality is the *device*. Devices have *ports*, and device ports are interconnected via untyped *connectors*. Connectors can be unicast or multicast, i.e. they can connect one output port with various input ports. They do not exist in an isolated fashion, but are contained by *connector boxes*. A connector box (CoBox in Fig. 2) can connect the input and output ports of attached devices in variable ways by activating or deactivating connectors or sets of connectors. Stream graphs are assembled out of connector boxes and devices. Both the connector box and the device interface are derived from the *terminal resource* interface which reflects their property of requiring resources for instantiation. A device may require many resources, among them hardware and computation time; a connector box may require inter-process communication resources. The terminal resource interface defines two important operations: activate() and deactivate(). Deactivating a connector box is equivalent to the temporary disconnection of all attached devices. A deactivated device releases all of its resources and halts operation. The terminal resource interface is finally derived from the general APMTObject interface.

There are three different kinds of devices. InterDevices have at least one input and output port and do perform some intermediate processing on a medium stream. EndpointDevices are media stream sources and sinks. They differ from InterDevices in that they have only input or only output ports, and that they have a *logical device name*. Endpoint devices can only be addressed if they have names. Video windows will be named with their window identifier, whereas external hardware devices will be named by standard functionality descriptions: PersonalCamera, RoomCamera, ObjectCamera. It is the visibility of EndpointDevices at the user interface that requires them to be addressable. TransportDevices perform media transmission over the network to which the terminal is attached. They are visible because it is assumed that a central connection manager will need to manage transmission compatibility among various terminals. If terminals negotiated network transmission among themselves, network connections could be represented by untyped connectors, i.e., without differentiating them from connections among terminal devices.

Devices and connector boxes are put together to form streams graphs. Fig. 3 shows as an example a video

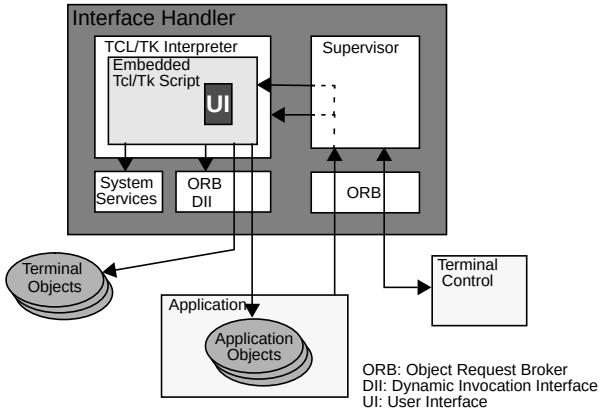

Figure 4. User interface handler

sender graph. The source of the video stream is a camera c that connects to the input port of a video codec. The video codec has an output port for digitized video and one for the unprocessed camera image that is shown in window n. The digitized video is compressed in a JPEG compression device and simultaneously transmitted and stored in a file x.y. This is achieved via a multicast connector in the connector box left to the JPEG compression. This connector box could contain another connector that would only connect the video transport device. Switching between these two connectors allows to control the recording of the compressed video stream. The graph interface allows to start, park and restart graph action. Graphs can also be modified by adding or removing branches. The stream agent interface offers the create_graph() and remove_graph() operations. Graphs are described as a sequence of device and connector box descriptions. The graph generated with a single create_graph() call can be arbitrarily complex.

A major difference between IMA and APMT is the way processing functionality is mapped onto devices. IMA offloads much of the device functionality into format objects. The IMA virtual device supports different formats on its ports and has thus an unlimited, but undefined range of functionality. An APMT device on the other hand has clearly defined functionality, and is thus predictable for a programmer. As an example, video stream compression in IMA is modelled as a format, whereas in APMT it is modelled as a separate device. IMA object instantiation is based on trading - an application requests an object by specifying the properties it is interested in. One and the same device may have very different properties on different systems. APMT on the other hand models properties in interface hierarchies as is indicated in Fig. 2 with extensibility dots to the right of InterDevices and EndpointDevices. The result of such a proceeding is that hardware/ software device developers have to stick to standard device specifications, i.e., they have to implement the complete interface of a standard device, which in turn makes the device predictable to the application programmer. The benefit of this is application compatibility among terminals, a drawback maybe that new processing features have to be packed into devices by a light-weight standardization process before they can be employed by applications.

5.3. User interface handler

Fig. 4 depicts the Tcl/Tk user interface handler that was implemented for the terminal. It is based on a pseudo language mapping for Tcl and an equivalent extension of the Tcl interpreter [18]. Applications download a Tcl/Tk script into the user interface handler where it is evaluated. The script generates a Tk user interface and may be fed by the application with object references, or discover object references via the CORBA name service. Button clicks by the user will then result in operation invocations either to local terminal objects or to distant application specific objects. The interface of the user interface handler exports a part of the Tcl/Tk C library and allows for instance to download icons and photos, set or get variable values and to call script procedures. Security is a major concern: the script accesses system services like file I/O via a set of Tcl procedures provided by the interface handler, rather than directly and unchecked.

6. Connection Manager

The CxMgr is the most important auxiliary component that can be envisaged for the application pool. It has to be considered as a special application offering services to real user applications running on top of it. Its usefulness grows with the number of terminals participating in an application - teleconference applications and any other multi-user application will depend on its connection management services. The CxMgr found in an application pool within a public network will deal with the network infrastructure in addition to managing the terminals. Even simple applications like a video-phone or a video on-demand service will have to be built on top of a connection manager. As was already discussed, there can be different connection managers covering different application classes.

The CxMgr being implemented for our platform provides support for the establishment of simplex, duplex, multicast, all-to-all and all-to-one connection structures called *bridges*. Applications register *graph models* with the CxMgr. A graph model describes an arbitrary graph the CxMgr has to instantiate within the terminals at the vertices of a bridge. Graph models do not contain transport devices. The CxMgr adds transport devices automatically to the graphs it instantiates so as to reach compatibility

among the connected terminals. Transport devices are the only devices that are known to the CxMgr, i.e., the CxMgr does not have the notion of audio and video, and the components of a graph model are transparent to it. This is necessary to keep the platform open for the introduction of new media processing devices.

The application creates bridges on terminal *subsets*. On a create_bridge() operation invocation, the CxMgr instantiates the graph models named for the bridge in the concerned terminals and interconnects them. On a destroy_bridge() invocation, the CxMgr will disconnect the various graphs it generated, but it will park rather than destroy them. The parked graphs can then be reused by other bridges in a later create_bridge() call with the primary advantage of accelerating connection setup. This way the application can rapidly switch between fundamentally different connection structures.

Complex IDL types are necessary to describe graph models, with the result that the interface of the CxMgr is too complex to be used directly by application programmers. The interface provided to application programmers must be a convenience library that allows to construct graph models with multiple function calls; another idea would again be a Tcl extension for graph model construction.

7. A Tele-Conferencing Toolkit

The functionality of the platform can be further augmented by specialized tool-kits and authoring environments. The teleconferencing toolkit we present here is a mind game is an adaptation of the Beteus API [6] to the APMT platform and demonstrates how the monolithic programming interfaces known from platforms like Beteus or the Touring Machine can be built on top of an extensible platform. The toolkit supports user interface handling and audio, video and application sharing among a set of terminals. It is adequate for the prototyping of simple cooperative teleconferencing applications as was seen in the Beteus project [5]. The toolkit is realized as a server in the application pool which itself uses the services of the connection manager. It completely hides the platform from the application: neither the ApMgr, the CxMgr nor the terminals are visible to it.

7.1. Abstractions and interfaces

The toolkit resumes the abstractions role, bridge and bridge set that were described in Section 3.2. Roles can either be static or transient. A terminal can hold one static role and multiple transient roles at a time. Static roles are linked with a certain user interface and will be rarely swapped. Transient roles can be dynamically created, assigned, transferred, taken away and deleted. Bridges are connection structures of type audio, video and application sharing whose endpoints are defined with role names, or more exactly, both bridge source and sink terminals are given as a list of role names. Depending on the cardinality of a role within the application session a bridge can represent a point-to-point, point-to-multipoint, multipoint-to-point and multipoint-to-multipoint connection. Since the toolkit is aware of audio and video devices it knows that it has to realize a sink for multiple audio streams with a mixer, whereas it realizes a multistream video sink with one video receiver graph per stream. Bridges are active CORBA objects: the interfaces AudioBridge, VideoBridge and SharingBridge are derived from a general Bridge interface. The interfaces AudioBridge and VideoBridge allow to adjust a certain amount of settings like frame rate, window size and position, and audio encoding and represent a simplified interface to what will be realized as a graph of media processing devices. Bridges can further be grouped into bridge sets. An application defines multiple bridge sets corresponding to fundamentally different connection structures. During the lifetime of the session, the application switches from one bridge set to another, changing the connection structure with a single call. The bridge set abstraction is maybe more an application design paradigm than a programming aid: since a bridge set corresponds to an application state it helps to structure a teleconferencing application on the time scale.

The SharingBridge controls the interconnection of application sharing agents on terminals. A SharingBridge is defined with a single and uniquely assigned source role and possibly multiple sink roles. The user at the source terminal has control over the application of which the user interface is replicated at the sink terminals of the bridge.

The application furnishes one Tcl/Tk user interface script for every static role it defines. This script must have a standard interface that can be called by the toolkit, with functions for role assignment and others for the assignment of object references for callbacks to the application. The application must also furnish an interface for the standard *participant* role that is automatically assigned to every terminal in the application. This allows joining users to identify themselves and to take their static role if the application does not assign it automatically.

When the application is started up it will first launch a toolkit server and configure it via its main interface. The main interface of the toolkit offers operations for role name registration, bridge and bridge set creation and script transfer, called in the order as they are mentioned

No.	Type	Source Roles	Sink Roles
1	audio	participant	participant
2	audio	professor,studentSpeaker	professor,studentSpeaker
3	audio	studentSpeaker	participant
4	video	professor	student
5	video	student	professor
6	video	professor,studentSpeaker	professor,studentSpeaker
7	video	studentSpeaker	participant
8	sharedApp.	studentSpeaker	professor
9	sharedApp.	studentSpeaker	participant

Table 1. Example bridge definitions.

here. Bridge and bridge sets are objects with interfaces and can therefore be manipulated at any time during the lifetime of the application. The application transfers one user interface script per static role to the toolkit. As new terminals join the application, they automatically get the participant user interface and later, when the application assigns a static role to them, the respective application specific user interface. The toolkit demands for every new user a callback object reference from the application that it then hands over to the user interface script.

Once the toolkit server is configured, the application will call the toolkit operation session_start() - from then on the application will only react to callbacks from terminals, accept new terminals, reassign roles and change the active bridge set. The toolkit will map role names to terminal names and create the respective bridges via the CxMgr.

7.2. Example scenario

An example shall serve to illustrate how application scenarios are translated into role, bridge and bridge set definitions. Imagine a virtual school with professors and students all geographically dispersed. The school is an application pool somewhere in the network to which both professors and students connect via multimedia terminals. Professors have application scenarios for all kinds of teaching purposes at hand, among them a scenario that supports translation work on stage-plays written in a foreign language. The scenario has five states or phases:

- **phase one**: introduction
- **phase two**: students work, professor visits
- **phase three**: students present their results
- **phase four**: recitation of the stage-play
- **phase five** = phase one: conclusion

In a first phase, the professor gives an introduction into the translation assignment that was previously distributed by E-mail. Students see and hear the professor, and they hear each other, which allows them to hear questions asked to the professor by fellow students. In a second phase, the students start to work on the translation of the stage-play. The professor goes from student to student and answers their questions. The editor of the current student is automatically shared with the professor. The professor may return to phase one if one of the questions is of general interest. Once students have finished the translation, phase three begins where students present their results. The professor and the presently presenting student are visible to all other students and to each other. The editor of the student is automatically shared with all others, and audio is like in phase one. In phase four, students take roles in the stage-play and recite them. Their image and voice is distributed to the professor and to the other students. The professor finishes the course with some remarks, with the application being again in phase one. During the whole session the professor has as the replacement of a classroom-view an icon-sized video image with low frame rate from every student.

The roles that can be identified in this scenario are:

- **professor**: static professor role
- **student**: static student role
- **studentSpeaker**: visible students in phase two, three, four
- **participant**: professor and students

The transient role studentSpeaker is assigned to the visited student in phase two, to the presenting student of phase three, and to the reciting students in phase four.

The bridges that need to be defined are shown in Table 1. The first audio bridge is the all-to-all audio of phase one and three. Audio bridge 2 and video bridge 6 form a bidirectional audiovisual connection for phase two. Audio bridge 3 and video bridge 7 form the virtual stage of phase 4. The multipoint-to-point bridge 5 represents the icon-sized classroom view.

Four bridge sets are defined according to the four different phases of the application scenario:

- **bridge set one**: audio bridge 1, video bridges 4+5
- **bridge set two**: audio bridge 2, video bridge 5+6, application sharing bridge 8
- **bridge set three**: audio bridge 1, video bridges 5+7, application sharing bridge 9
- **bridge set four**: audio bridge 3, video bridge 5+7

During the session the application switches between bridge sets and assigns the transient role studentSpeaker. Both kinds of actions are triggered by callbacks from the professor's user interface.

8. Discussion

The objective of our prototype implementation is to evaluate the suitability of CORBA for our platform. The CORBA implementation we are using is Orbix™ from Iona Technologies. We expect to have a first running version of the prototype at the end of February 1996.

Work on design and platform will continue in various directions. As for now our architecture does not address media synchronization and resource management. An important study issue will be connection management for terminals that are attached to different networks. The problem here is that a connection manager that is not aware of devices and device parameters has to configure an inter-terminal device network in a way that generated media streams do not exceed the bandwidth capacities of network links, or the computation capacities of terminals. In short, there has to be a quality of service framework for APMT.

We will also have to study how we can incorporate intelligent network and open distributed processing concepts into our architecture.

Pointers

Touring Machine: ftp://thumper.bellcore.com/pub/tm/
Beteus: http://www.cica.fr/~beteus
Lakes: http://cspcvwe1.leeds.ac.uk/WWW/lakes.html
Medusa: http://www.cam-orl.co.uk/medusa.html
IMA: http://www.ima.org:80/forums/imf/mss/

References

[1] C. Blum, R. Molva and E. Ruetsche,"A Terminal-Based Approach to Multimedia Service Provision", in *Proceedings of the 1st International Workshop on Community Networking*, San Francisco, July 1994.

[2] C. J. Screenan,"Synchronisation Services for Digital Continuous Media", Ph.D. thesis at the University of Cambridge, October 1992.

[3] M. Arango et al.,"The Touring Machine System", *Communications of the ACM*, vol. 36, no. 1, pp. 68-77, January 1993.

[4] V. Mak, M. Arango and T. Hickey,"The Application Programming Interface to the Touring Machine", Bellcore Technical Report, February 1993.

[5] C. Blum, Ph. Dubois, R. Molva and O. Schaller,"A Development and Runtime Platform for Teleconferencing Applications", submitted to the *IEEE Journal on Selected Areas in Communications*, special issue: Network Support for Multipoint Communication.

[6] C. Blum and O. Schaller,"The Beteus Application Programming Interface", Eurécom Technical Report, December 1995.

[7] L. A. Rowe, B. Smith, and S. Yenftp:" Tcl Distributed Programming (Tcl-DP)", University of Berkely Computer Science Division, *ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/Tcl-DP/tcl-dp-v1.0ak*, March 1993.

[8] J. K. Ousterhout, "TCL and TK Toolkit", Addison-Wesley Publishing,1994.

[9] IBM Lakes Team,"IBM Lakes: An Architecture for Collaborative Networking", R. Morgan Publishing, Chislehurst, 1994.

[10] S. Wray, T. Glauert and A. Hopper,"The Medusa Applications Environment", *IEEE Multimedia*, vol.1, no. 4, Winter 1994.

[11] F. Stajano and R. Walker,"Taming the Complexity of Distributed multimedia Applications", in *Proceedings of the 1995 Tcl/Tk Workshop*, Toronto, July 1995.

[12] Interactive Multimedia Association,"Multimedia System Services", *IMA Recommended Practice Draft*, Sep. 1994.

[13] J. F. Koegel Buford,"Multimedia Systems", pp221-244, Addison-Wesley Publishing Company, New York, 1994.

[14] Object Management Group,"The Common Object Request Broker: Architecture and Specification", John Wiley & Sons, Inc., 1992.

[15] H. C. Lucas, H. Levecq, R. Kraut and L. Streeter,"France's Grass-Roots Data Net," *IEEE Spectrum*, November 1995.

[16] J. Gosling and H. McGilton,"The Java Language Environment", Sun Microsystems White Paper, May 1995.

[17] Kaleida Labs,"ScriptX Technical Overview", Kaleida Labs Technical Report, http://www.kaleida.com, 1995.

[18] G. Almasi et al.,"TclDii: A TCL interface to the Orbix Dynamic Invocation Interface", Technical Report at the West Virginia University, http://www.cerc.wvu.edu/dice/iss/TclDii/TclDii.html, 1995.