In Proceedings of:
Multimedia Computing and Networking MMCN'97,
San José, CA, February 1997.

A CORBA-based platform for distributed multimedia applications

Christian Blum[1] and Refik Molva[2]

Institut Eurécom, BP 193, F-06904 Sophia-Antipolis

ABSTRACT

As distributed multimedia applications are starting to be offered as services in enterprise and residential cable networks, there is a growing interest in platforms that provide a standard framework for the development and deployment of these applications. Key issues in platform design are service diversity, service portability and interoperability of user terminal equipment. We propose a platform architecture for the provision of multimedia communication services which logically separates application processing from media processing. Applications are installed in *application pools* from where they control a set of communicating *multimedia terminals*. Application-specific intelligence is downloaded into the terminals in the form of Tcl/Tk or Java scripts that generate graphical user interfaces, control media processing components, and communicate with the application in the pool. The platform architecture is based on CORBA and is defined as an extensible set of IDL interfaces for control and stream interfaces for multimedia communication. The platform supports application development with high-level programming interfaces.

Keywords: networked multimedia applications, teleconferences, CORBA, TINA, IMA MSS

1. INTRODUCTION

As distributed multimedia applications are starting to be offered as services in enterprise and residential cable networks, there is a growing interest in platforms that provide a standard framework for the development and deployment of these applications. Platforms have to be seen in contrast to stand-alone applications with prototype character like they are found for instance in research environments. Such applications implement media processing on top of low-level device interfaces and are therefore highly dependent on hardware and operating system. They are built for a special purpose and require skilled personnel for setup and usage. While this is the normal approach to build research prototypes, it is clear that applications that are to be deployed as services must be integrated into a general runtime environment. This runtime environment must support the short life-cycles that are typical for multimedia applications and provide facilities for rapid development, deployment and removal of services. A first step into the direction of such an environment is the introduction of application programming interfaces that allow code and component reuse on a high level. Runtime platforms offer a certain amount of dynamically usable functionality to the applications that run on top of them, like for instance connection management services. Platforms are further characterized by an application independent session model that defines the relationship between user and service.

Runtime platforms for distributed multimedia applications like video conferencing are already commercially available [1], but they do not interoperate among each other. Interoperability has to be achieved on both control and medium object level. A multitude of standards exists for the definition of audio, video and graphics formats, allowing interoperability on medium object level, but agreement on a common control architecture for distributed multimedia applications has not been reached yet. There are ongoing standardization efforts to solve the framework problem for certain classes of applications, e.g. MHEG [2] for multimedia kiosk applications and the T.120 suite [3] for videoconferencing, but one problem with these standards is that their eventual success will hinder the adoption of more general and unifying frameworks like the multimedia system services (MSS) architecture of IMA [4] or the TINA architecture [5].

In order to come up with a common control architecture, agreement has to be reached on a distributed processing environment. The most promising candidate here is OMG's Object Management Architecture (OMA) with the Common Object Request Broker Architecture (CORBA) at its heart [6]. CORBA is an excellent platform for distributed applications in general, and as such is also an excellent choice for distributed multimedia applications. The problem with CORBA is that some of its important extensions are still in the process of being standardized, and can consequently not be used in other standards. Never-

[1] Correspondence C.B. E-mail: blum@eurecom.fr; WWW:http://www.eurecom.fr/~blum/; Telephone: (+33) 4.93.00.26.38
[2] Correspondence R.M. E-mail molva@eurecom.fr; WWW:http://www.eurecom.fr/Corporate/Staff/molva.html; Telephone: (+33) 4.93.00.26.12

theless, OMG has recognized the role that it can play in the telecommunications market, and is considering for instance a medium data stream extension for CORBA [7].

In this paper, we propose a platform architecture based on CORBA that fosters development and deployment of distributed multimedia applications. In this architecture, we separate application processing and media processing not only logically, but also geographically. Applications reside in the network in 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 supports a wide variety of services. Toolkit-like reusable components can be added to the application pool that provide comfortable interfaces for application development.

This paper provides insight into all issues that have been addressed so far in our architecture. It starts off with a general discussion of the benefits of distributed object computing principles and CORBA for networked multimedia applications (Section 2). It then presents an overview of our architecture (Section 3), followed by sections discussing its principal components, i.e., the multimedia terminal and the application pool (Section 4 and 5). A subsection is dedicated to the connection and configuration manager in the application pool, an application pool component that allows applications to establish complex source-to-sink connections among participating terminals. A final section before the conclusion describes the principal features of the prototype that is being developed at our institute (Section 7).

2. DISTRIBUTED OBJECT COMPUTING FOR MULTIMEDIA PLATFORMS

A history of multimedia platforms would start with the Touring Machine that was developed at Bellcore at the beginning of this decade [8]. While antiquated with respect to today's standards, it was at the time the distributed multimedia applications platform with the most powerful application programming interface (API) [9]. The target applications for the Touring Machine were teleconferences with collaboration support. The Touring Machine architecture is object-oriented in that it decomposes the functionality of the platform into a set of interacting managers and agents of which some are dynamically created when needed. One of the problems of the Touring Machine architecture is that it does not support the development of platform extensions by third parties - the internal interfaces of the platform are hidden, and communication among the various platform objects is provided by a proprietary message passing protocol. Also, every new feature or object that is added to the platform requires the modification of the object implementing the monolithic API if it is to be visible to applications.

Early work in the area of object-oriented stream processing, with emphasis on programming aspects, has been performed by Simon Gibbs at the University of Geneva [10]. Gibbs models stream processing with a network of source, sink and filter components called *active objects*. An object has an operational interface for control and one or more media stream interfaces called *ports*. An object is active because it performs actions in the absence of control messages, i.e., it is data driven. Such a model is also at the base of the Open Distributed Processing (ODP) reference model [11], and can be considered as the dominating paradigm for the modeling of multimedia stream processing today.

An interesting platform deploying the object model is Medusa that was developed at Olivetti Research Labs [12]. It was built for an ATM network that supports the direct interconnection 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. Active objects in Medusa are called *modules*, with applications being a special kind of module. The platform is designed to be extensible; reusable functionality is implemented within platform rather than application modules. Every application developed for Medusa will thus enrich the platform by increasing the number of modules that are available. Just like the Touring Machine, Medusa uses a proprietary mechanism for control communication among platform components. The Medusa platform is a middle-ware layer for networked multimedia applications, and does not define a framework for sessions or service provision. A similar platform is CINEMA developed at the University of Stuttgart [13].

The major standardization effort in the area of multimedia middle-ware is the multimedia system services (MSS) architecture of the Interactive Multimedia Association (IMA) [4]. As a standard, MSS is forced to either define its own control and media communication frameworks, or to deploy existing standards for this purpose. MSS is built on top of CORBA to solve the control and programming problem, and will define its own media stream protocols. MSS only defines interfaces within a basic inheritance tree; the richness of the architecture is then to be provided by third-party architecture extensions. One problem with MSS is its complex connection establishment procedure. Another problem is that it was started at a time when CORBA itself was not developed beyond the basic request broker architecture. CORBA features like the common object services that are now available could not be foreseen in the architecture, and have to be forced into it now. An example for this is the event service. Although MSS has been proposed to DAVIC [14], and is part of the ISO PREMO standard [15], its future appears to be unclear [16].

The major standardization effort in the area of multimedia service provision is the TINA initiative [5]. TINA provides a complete framework for all aspects of service provision in a future computation-oriented tele-communications network. TINA adopts the ODP model at the base of the architecture, and, with reservations, CORBA as the best available distributed processing environment on which it places its objects. Media processing objects are within the scope of TINA, which means that there is an overlap between the activities of TINA and for instance MSS. Given its emphasis on service provision it is questionable if TINA can come up with abstractions for multimedia programming that are as powerful as those foreseen for MSS. A future standard for a service provision framework would do best in leaving space not only for new components, but also for new programming paradigms.

3. APPLICATION POOLS AND MULTIMEDIA TERMINALS

With our architecture we try to provide a complete framework for the development and deployment of multimedia services [17]. This framework is a superset of MSS since it defines a relation between application and user in addition to a multimedia middle-ware. It is a subset of TINA because it does not go beyond a lightweight session model - it does not try to integrate itself into the network. It is thus an overlay architecture, and as such it is a normal user of the transport services of the underlying network.

Service provision in a network requires the existence of standard terminal equipment at the user's premises. The terminal architecture must be extensible in order to accommodate new software devices possibly representing new hardware. It must also take into account that the services to which the terminal can connect will be numerous and will have short life-cycles. This means that is not at all practical to install application specific software on a terminal or a network file system. In our architecture we chose is to distribute application intelligence between servers and terminals. An application residing on an application pool will download scripts into the terminals that serve as intelligent sensors and 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 coordination, 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. A terminal can activate a certain application only if it has the devices that the application requires. The administrational effort associated with the terminal is thus the installation of hardware and of component software with comparatively long life-cycles. 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 architecture is based on CORBA, and is thus specified via an extensible collection of IDL (Interface Definition Language) definitions.

The left side of Fig. 1 shows, from an engineering perspective, the major components of application pool and multimedia terminal along with control and media flows. The right side depicts the architecture from the application point of view. At the

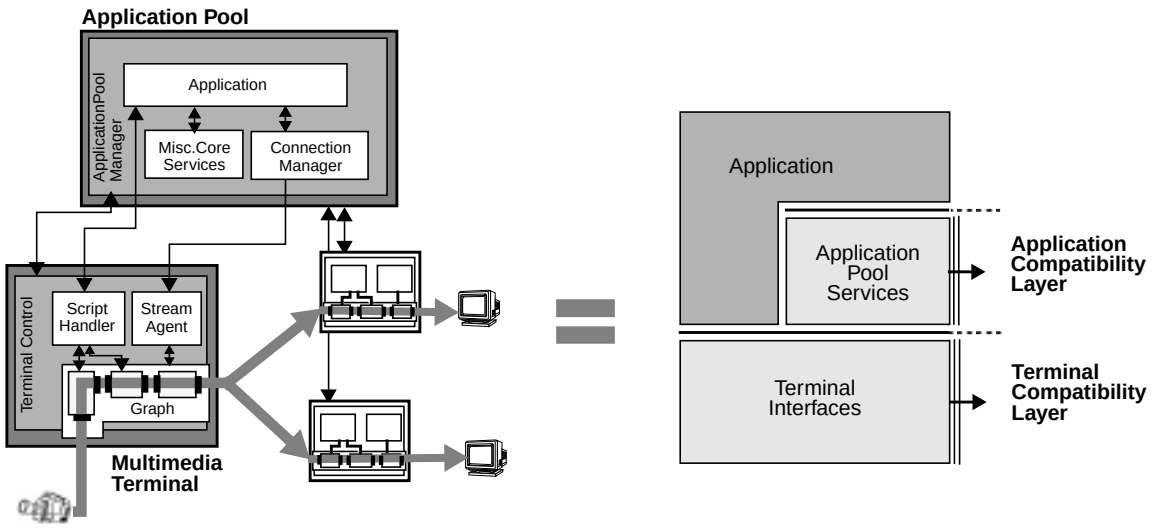

Figure 1. The APMT architecture.

bottom is the extensible *terminal compatibility layer* that is provided by the totality of visible terminal object interfaces. The *application compatibility layer* is built on top of the terminal interfaces and provides high-level support for applications that do not want to deal directly with certain terminal objects. It is provided by the totality of application pool object interfaces that are visible to the application. The application interfaces to both terminal and application pool services. It can only run in pools that offer the application services it needs.

For convenience, we will refer to our architecture as APMT (Application Pool - Multimedia Terminal). The following subsections give an overview of APMT.

3.1. Overview of the multimedia terminal

The brain of the multimedia terminal is the *terminal control* (see Fig. 1). The terminal control manages the application lifecycle on the terminal side: it starts and joins applications in the application pool on behalf of the user, or on behalf of applications that are already running on the terminal, 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 *script handler* and the *stream agent*. A script 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 [18]. If the downloaded script is to perform more advanced tasks than the user interface, strongly typed languages like Java [19] or ScriptX [20] 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 script 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. The script handler can claim such object references 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.

3.2. Overview of the application pool

The counterpart to the terminal control in the application pool is the application pool manager (APM). The APM launches applications on behalf of terminals, and invites terminals on behalf of applications. The APM grants applications access to the application pool objects and monitors them.

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 and configuration manager (CCM)*. The CCM provides support for the establishment of complex connection structures among groups of terminals and configures the device graphs within these terminals. An application will usually prefer to deal with one connection manager rather than many stream agents. Multiple connection managers can be imagined that provide support for different categories of applications.

The component *Miscellaneous Core Services* in Fig. 1 alludes to modules other than the CCM, like for instance a module that handles multiple graphical user interfaces within a session. Note that an application may itself start another application, so as to run multiple applications on top of a single session.

3.3. Additional components

The most important additional component to be mentioned is the directory service that is needed for example to register terminal addresses, user identifiers, momentary user locations, as well as announced and ongoing sessions. The directory service could be transparently accessed via the CORBA trading service [21]. Another important component would be a service gateway between terminal and application pool that transparently routes service requests to application pools, so as to provide for load

balancing on application pool level. Both the directory service and the service gateway will be described in a future publication about APMT.

Media servers are supported by the terminal abstraction and do not require any extension of the architecture.

4. MULTIMEDIA TERMINAL

This section discusses the major building blocks of the terminal. It starts with a description of the multimedia middle-ware layer that is implemented by the terminal, and describes based on this the terminal components that are grouped around this middle-ware.

4.1. Terminal middle-ware

The elementary unit of processing functionality in APMT is the *device*. A device has an operational IDL interface that is visible to the application, and a management IDL interface that is hidden from the outside. A device can generate events for which interested APMT objects register. Events may contain processing results, or they may inform other APMT objects about changes in the state of the device. The latter is important given that the control of devices can be shared among multiple client objects. Devices may also become active in the sense that they obtain object references and invoke operations of the respective interfaces.

A device contains a set of ports that are classified into input ports and output ports. Media data enter the device via input ports, and leave it via output ports. There is no inherent constraint on the relationship between the data that enter a device and the data that leave it, i.e., outgoing data may be of a completely different type than incoming data. Ports appear as simple identifiers in the definition of a device, and not as active objects with an externally visible IDL interface. Port identifiers are used to interconnect compatible ports of different devices. An output port can be connected to an input port if both ports support the same medium format. Media formats are given as chained identifiers that form a medium format hierarchy that starts at the root *any* and has subtrees for major media types like image, audio, video or text. Again, formats do not appear as objects with an IDL interface as for instance in MSS. In fact, APMT devices are closer to the format abstraction than MSS devices, which allows to set format parameters in the operational interface of the device rather than the one of a format object. The result of this is that APMT defines different devices for MPEG and JPEG compression, whereas MPEG and JPEG appear as port format objects in MSS. The benefit of renouncing on format objects is reduced complexity when it comes to matching port formats in a device network that is spanning multiple terminals. It is assumed that a device supporting a certain format does automatically support a certain range of parameter values associated with this format. Format parameter settings of source devices are conveyed as part of the medium stream data, with receiving devices adapting themselves to these parameters on-the-fly. A device definition restricts the formats available on a port to a subtree in the medium format hierarchy. The formats that are then supported on the port must be equivalent with respect to the operational interface of the device. This allows to introduce a wide variety of formats into the architecture without adding new devices. It is even imaginable that subformats are defined that correspond to different parameter value settings in a parent format, e.g., different types of MPEG1 video.

Device ports are interconnected via *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 under control of *connector boxes*. A connector box can connect the input and output ports of attached devices in variable ways by activating or deactivating its connectors. It must therefore be considered as a controllable software switch for media streams.

Devices and connector boxes are assembled to form arbitrarily shaped *stream graphs*. Stream graphs provide a way to execute compound operations on the devices and connector boxes they contain. The two principal operations the graph interface defines are `start()` and `park()`. The `start()` operation activates a graph, which results in an activation of all devices and connector boxes. The `park()` operation deactivates a graph, which corresponds to the release of all resources that the graph holds. It is assumed that it is much faster to restart a parked graph than to create an identical graph all over again, for instance from persistent object storage. Parked graphs keep state - parking a graph in our prototype means putting the process to sleep that implements the graph. A graph is also a device factory since it allows to add and to remove subgraphs.

Graphs themselves are created by the stream agent (see Fig. 1). The two principal operations of the stream agent interface are `create_graph()` and `remove_graph()`. The `create_graph()` operation is the most complex one defined by APMT. It takes as arguments a list of instantiation requests that describes the graph in terms of devices and connector boxes. Every instantiation request contains initialization data, which is for instance a list of connectors in the case of a connector box. The benefit of this approach is speed - a complex device network can be created with a single call instead of one call per device and connector box. Remote procedure calls do not come for free; they perform orders of magnitude worse than normal function

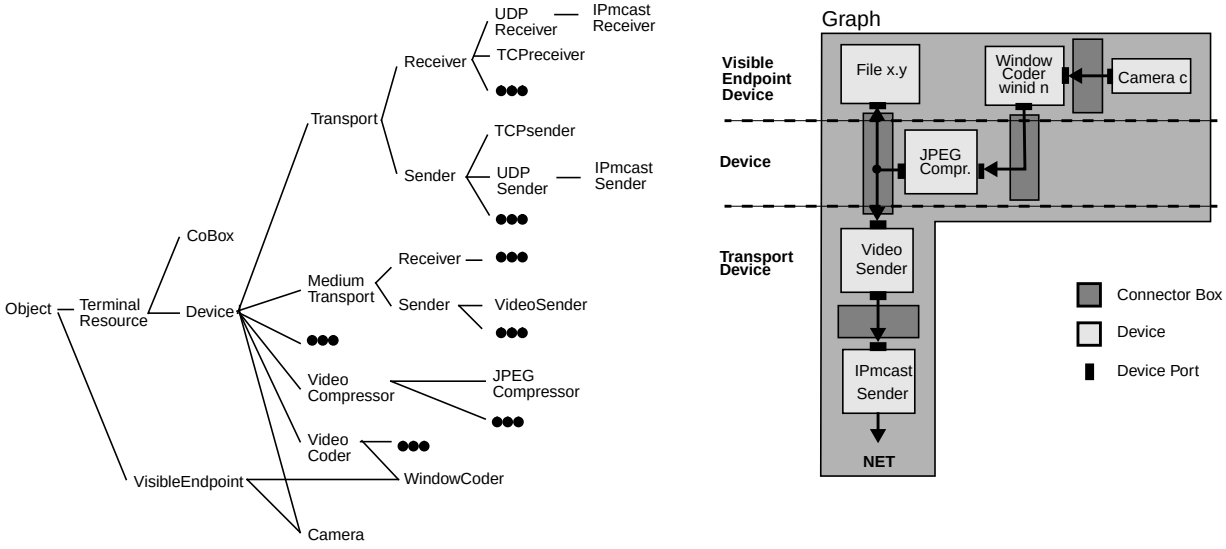

Figure 2. Device interface hierarchy and example device graph.

calls, and a platform for distributed applications like APMT must take this into account. As another point, the way graphs are created requires that the client has prior knowledge as to what objects are supported by the terminal. Information about this is made available to the client as part of a compatibility negotiation procedure on application startup.

Fig. 2 depicts on the left side the interface inheritance diagram for terminal devices. Both the connector box and the device are derived from a common base interface called *terminal resource* that defines the operations `activate()` and `deactivate()`. It is assumed that devices take hardware and computation resources, whereas connector box may take inter-process communication resources. Terminal resources, just like graphs, are deactivated after creation, and their activation may fail due to resource problems. When deactivated they keep state, but they release resources and stop processing.

The inheritance diagram further shows branches for *transport* and *medium transport*. The transport branch contains the devices with which the terminal interfaces to the network. Since unicast transmission is assumed there is a classification into sender and receiver, and then further into the transport protocols UDP, TCP and IP multicast. A medium transport device knows how to transmit a certain medium over the network. which concerns mainly packetization and transmission timing. Both categories of devices are intended to be used by application pool services like the CCM, and are of no interest for the application itself.

The inheritance diagram also shows a couple of video devices, among them a video coder that generates a window containing the video signal it is digitizing. The video window coder interface is also derived from the *visible endpoint* interface. A device is a visible endpoint if it has an address, or some kind of visibility at the user interface. A visible endpoint has some correspondence to the source/sink abstraction in other frameworks, but it does not put any restrictions on the ports endpoint devices contain. A visible endpoint can therefore be source (one output port), sink (one input port) or a mixture of the two.

The right side of shows as an example a graph that transmits video. This graph contains three visible devices: a camera, a video window coder, and a file. The camera is a visible endpoint because it needs to be addressed among the cameras that can be accessed by the terminal. The video window coder has a visible window with an identifier which allows the video to be displayed in a specific window, possibly one under control of the script handler. The file device finally has a pathname under which its content can be accessed on a storage medium. The JPEG compression device on the contrary does not need any address beyond its object reference. The connector box to the left of the JPEG compression device may contain a point-to-point connector in addition to the multicast connector which allows to control the recording of the compressed video stream simply by switching between the two connectors.

4.2. An internal stream interface

Device development is confronted with a stream interface that may be terminal architecture specific since it is hidden to the outside. It is clear that it is preferable to have a stream interface, and in general a device API, that is independent from the terminal architecture. As an example, we sketch the stream interface as it has been implemented in our prototype.

Inside the terminal, medium data is moved within *containers*. A container is an object that can store headers along with medium data. It provides methods for deep and shallow copies of medium data and headers and relieves the device programmer from memory management issues. Many devices will have some sort of relationship with peer devices in other terminals, or even with devices in the same graph. It is therefore practical to use the medium stream as a oneway control communication channel in downstream direction. A device that needs to communicate information that is needed by a downstream device for correct data processing will add an *attribute header* to the container. Another form of header is the *medium header* that describes the physical format of the medium data. A container provides access to a single medium header, and allows to add multiple attribute headers.

Ports are implemented as C++ objects that are dynamically created by devices when needed. The connector box connects ports for instance by setting a list of pointers to input ports in an output port. There are three mechanisms for communication among ports: *push*, *pull*, and *implicit*. The connector box makes sure that there is no mismatch between connected input and output ports with respect to the supported mechanism. With the push mechanism, a device transmits a container by calling the respective procedure in the output port interface, and receives a container via a callback from an input port. With the pull mechanism, a device retrieves a container by calling the pull method in a `PullInPort` which in turn will call the same method in a `PullOutPort`. The `PullOutPort` will then retrieve a container out of a buffer maintained by the upstream device and return it via the `PullInPort` to the downstream device. With the implicit mechanism, data is actually not exchanged, for instance because it is in analog form. The implicit mechanism is used when the APMT object modeling does not match the reality of data exchange and control on a particular endsystem. As an example, the window coder and the camera in the example graph in Fig. 2 will be linked via an implicit mechanism. The camera object is a proxy that forwards operation invocations to the video coder object which is known to it.

4.3. A stream format on transport level

A sender transport device transforms a container into a PDU and transmits it over the network. A receiver transport device recreates a container from a received PDU and forwards it to downstream devices. An APMT medium data PDU contains a main header, a medium header, a variable number of attribute headers, and the medium data.

The PDU format is part of the terminal interface specification. This means that the format of medium headers attribute headers, and medium data must be specified. It is an open issue if the header format specification must be linked with the device specification, so as to be able to perform compatibility checks upon graph creation.

4.4. Script handler

The Tcl/Tk script handler that was implemented for the terminal is based on a proprietary IDL language mapping for Tcl and an equivalent extension of the Tcl interpreter [22]. Applications download a Tcl/Tk script into the script 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 via the CORBA dynamic invocation interface (DII). The IDL interface of the script 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 script handler, rather than directly and unchecked.

While Tcl/Tk is excellent for the generation of graphical user interfaces, it is not a good language for the programming of complicated tasks. The language of choice here is Java [19]. Java will make it possible to develop the APMT application model towards scenarios where script handlers in terminals communicate directly with each other.

4.5. Terminal control

The terminal control is a container for application related interfaces. It implements the `Terminal` interface to which application pools direct session invitations and announcements. There is further an interface of the terminal control to the local user that offers, among other operations, a registration operation which associates the terminal with the user. The registration operation returns a reference to an `ApplicationControl` interface that allows to start and join applications in application pools.

Both operations return an `Application` interface via which an application can be controlled in which the local user participates. The `Application` interface allows to pause, resume, hide, show, quit and kill an application. The terminal control provides a reference to a `TerminalControl` interface to every application in which it participates. The application, probably the script that is running on the terminal, can use this interface to get access to an `ApplicationControl` interface. This allows applications to start themselves other applications. A striking example for this is the yellow-page application that browses the directory service for available applications or ongoing sessions and allows the user to transparently start or join a displayed application. The `TerminalControl` interface contains in addition a name service that allows objects to register themselves on instantiation with their object reference and a name. The script in the script handler may register callbacks for the instantiation of objects with a certain name. The object name service is necessary because most of the graphs, devices and connector boxes in the terminal will be instantiated by auxiliary servers like the CCM in the application pool, which means that their object references are not automatically available to the application. One solution for this is to have the CCM call the application whenever it instantiates a graph, but this is tedious. Instead of that, an application may assign names to the devices it requests, and once these devices are instantiated in a terminal they are registered with the terminal control which in turn forwards the respective object references to the interested local application script.

5. APPLICATION POOL

This section describes the application pool manager and the CCM, with emphasis on the latter.

5.1. Application pool manager

The application pool manager (APM) provides a control framework for the application pool similar to the one that is provided by the terminal control for the terminal. The interface towards the terminals is the `Pool` interface. This interface allows terminals to request a reference to an `Application` interface by providing an application name in the case of application startup, or an application identifier for joining an ongoing session. The `Application` interface offers operations to determine terminal compatibility, and to start, join, kill or quit the application. The APM implements further an interface towards the applications that allows an application to invite users. Applications themselves implement a management interface towards the APM.

5.2. Connection and configuration manager

The CCM is the most important auxiliary component that can be envisaged for the application pool [23]. It has to be considered as a special application offering connection establishment and configuration services to real user applications running on top of it. The CCM is not a canonical component of the application pool that has to be used by all applications that require connection services; it is rather foreseen that there are different CCMs for different classes of applications. The interface of a CCM, which is in fact an API, is adapted to the class of applications it supports. The CCM presented here is tailored to conferencing applications with a possibly large and dynamically changing number of participating terminals. To distinguish it from the general CCM we will refer to it as conference CCM (CCCM).

The CCCM supports the establishment of simplex, duplex, multicast, all-to-all and all-to-one connection structures called *bridges*. Bridges are instantiated on *subsets* of terminals that participate in the application session. The endpoints of such bridges are formed by device graphs within the interconnected terminals. An application registers so-called *graph models* with the CCCM. Graph models indicate the device ports on which they interface to the network, but they do not contain any transport or medium transport devices. As an example, a graph model for the video sender in Fig. 2 does not contain the `VideoSender` and the `IPmcastSender` devices. When creating a bridge, an application will identify two of the registered graph models that serve as bridge endpoints: a sender graph and a receiver graph. A sender graph contains exactly one network port. The situation with receiver graphs is more complex because a receiving terminal in a bridge may have to handle streams from multiple sources. There are basically two options for the receiver graph, which are replication or concentration. In the case of replication one receiver graph is instantiated for every incoming stream. In the case of concentration, the application indicates a concentrating device within the graph model that will be the sink of all incoming streams. An example for a concentrating device is the audio mixer.

When asked to activate a bridge the CCCM adds appropriate transport devices to the respective graph models and instantiates the resulting complete graphs on the terminals that are interconnected via the bridge. Note that the CCCM does not know any other devices than transport devices and connector boxes, and does not have a notion of audio, video or other media types. This is necessary to keep the platform open for the introduction of new media devices in the terminals.

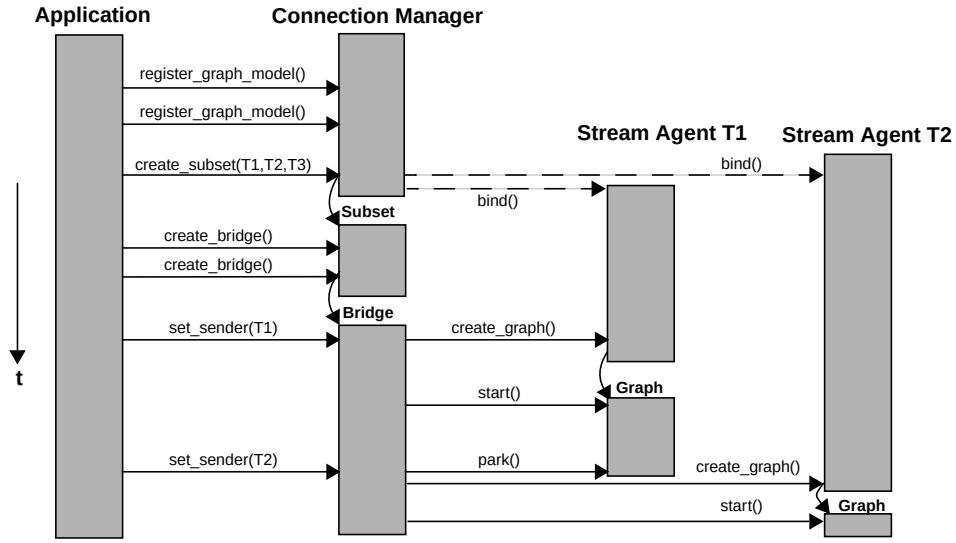

Figure 3. CCM usage scenario.

Fig. 3 illustrates the usage of the CCCM. On startup, the application registers a set of graph models with the CCCM. It then creates a terminal subset that represents a subsession with respect to the application session. The call `create_subset()` contains already an initial list of terminals, in this case T1, T2 and T3. The interface `Subset` itself allows to add or remove terminals from the subsession. Whenever a new terminal is added to any of the subsets, the CCCM will immediately bind to its stream agent. In a next step, the application creates two bridges on the subset. Fig. 2 shows one if these bridges, which is actually a multicast bridge. The multicast bridge becomes active when the application denominates a sender. The call to `set_sender(T1)` results in an instantiation of a sender graph on terminal T1 and receiver graphs on T2 and T3 (not shown). Once the graphs are created the CCCM will call `start()` first on all receiver graphs, and finally on the sender graph. At a later stage, the application decides to assign the sender role to T2. The CCCM will then park the sender graph on T1 and create a receiver graph on T1 and a sender graph on T2 and start them. The transfer of the sender role from T1 to T2 will take some time because the respective graphs need to be created, but once these graphs exist it is assumed that the application can rapidly switch back and forth between T1 and T2.

6. APMT Prototype

We have a first prototype running that implements most of the building blocks described in the previous two sections. The CORBA implementation we are using is Orbix^{TM} from Iona Technologies [24]. The prototype was developed on SunOS 4.1.3 and runs on Sun Sparc10 stations grouped around an ATM LAN switch in our laboratory. The Orbix 1.3 version for SunOs 4.1.3 we were using did not support multithreading, which turned out to be a serious limitation especially for the implementation of the CCCM. The speed of connection establishment via the CCCM can be increased dramatically if graphs are created, started and parked in parallel. Since it was not possible to implement the CCCM as a multithreaded client it was decided to use the deferred synchronous call mechanism of the CORBA dynamic invocation interface (DII) for this purpose. The problem here is that marshalling and unmarshalling has to be coded by hand, which is very tedious for the complex IDL types used especially in the `create_graph()` call of the stream agent.

A demo application has been implemented that sheds light on all major features of the platform. The demo application simulates a videoconference among three terminals. It first loads Tcl/Tk graphical user interfaces into the script handlers. It then creates a video multicast bridge on the three terminals and rotates the sender among them. The sender rotation appears in the application code as the loop depicted in Fig. 4. Once rotation is terminated the application creates low quality video multicast bridges to create all-to-all visibility. A momentary speaker is transmitting high quality video, i.e., big window size and high frame rate, whereas the other terminals transmit low quality video. The graphical user interface allows a user to request the speaker role, in which case the connection structure is completely rearranged.

The prototype demonstrates the feasibility of APMT, and its merits. The platform was implemented by two programmers within five months. This would not have been possible without CORBA/Orbix providing a ready-made object-oriented control

```
/* rotate senders */
for (i=0; i<ROUND_NUMBER; i++)
  for (int j=0; j<termnum; j++)
    {
      try {
        mcast_bridge->set_sender(terminal[j]);
      }
      catch (...) {
        cerr << "\ncould not change sender...\n";
      }
      sleep(10);
    }
```

Figure 4. Example code for sender rotation

middle-ware. The demo application could be implemented with a few lines of code although it already employs a rather complex and dynamically changing connection structure. Lessons learned were mostly resource related. For instance, care has to be taken in the mapping of interface implementations onto processes since this has a hidden effect on the number of TCP connections established by Orbix. A problem is also the big amount of C++ stub and skeleton code generated by the IDL compiler for interface collections like the one of APMT, which leads to excessive compilation times and large executables.

Work on the prototype is continuing. It has meanwhile been ported to Solaris 2.5 and Orbix 2.0. The deferred synchronous calls in the CCCM are being replaced by threads, and audio devices are being added to the terminal. Work has started to integrate a video server into the platform, and it is further planned to develop a Java script handler based on OrbixWeb [24].

7. DISCUSSION

A platform has been presented that supports development and deployment of networked multimedia applications. The applications that are installed in the application pool are made available as services to multimedia terminals in the periphery. A main characteristic of the platform is that it is based on CORBA, a standard framework for distributed object computing. The principal issues addressed by the platform are: service diversity, terminal compatibility, application portability and platform extensibility. The platform can be deployed today as an overlay service provision framework for IP networks that do not support the notion of service beyond packet transfer services. A platform like APMT can add a communication service layer to the Internet in a way similar to the MBone [25], but with a much wider scope.

The multimedia middleware of APMT is tailored to the multimedia terminal abstraction which allows it to be much simpler than IMA MSS [4] and more powerful when it comes to multipoint applications. Other work in the area of multimedia middleware leaves configuration and connection management (or stream binding in ODP terms) to the application. This becomes a complex task when many devices and endpoints are involved, which is why we introduce the intermediate CCM which relieves the application from most computational aspects of connection and configuration management. Finally we provide a complete application framework into which dynamically downloadable multimedia applications can be embedded with reasonable effort. This is similar in spirit to TINA, but again, simpler and certainly more realistic for the time being.

The platform lacks two important features that cannot be neglected: synchronization and resource management. Synchronization will be addressed in future work. Resource management is reflected in the APMT device and graph interfaces, but a real resource management architecture for the multimedia terminal has not been defined. We point out that interesting research in resource management for object-based multimedia middleware is going on at Lancaster University [26]. Equally, all platform interfaces will be rewritten to make a maximum use of existing CORBA services of which a good number seems to be interesting for APMT. The architecture of the platform will at this point diverge from the one of the prototype, because many of the used services will not have been implemented yet by CORBA vendors.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments that helped to improve the paper.

REFERENCES

1. InSoft Inc.,"InSoft OpenDVE: Digital Video Everywhere", http://www.insoft.com/ProductOverview/OpenDVE/OpenDVE.html, 1996.
2. Th. Meyer-Boudnik and W. Effelsberg,"MHEG Explained", *IEEE Multimedia Journal*, vol. 2, no. 1, pp. 26-38, Spring 1995.
3. Interactive Multimedia Teleconferencing Consortium (IMTC),"T.120 Standards for Audiographic Teleconferencing", ftp://ftp.csn.net/ConferTech, January 1996.
4. Interactive Multimedia Association,"Multimedia System Services", *IMA Recommended Practice Draft*, Sep. 1994.
5. D. Brown and S. Montesi,"Requirements upon TINA-C Architecture", Telecommunications Information Networking Architecture Consortium document TB_MH.002_2.0_94, February 1995.
6. Object Management Group,"The Common Object Request Broker: Architecture and Specification", John Wiley & Sons, Inc., 1992.
7. K. Raymond,"Streams and QoS: A White Paper", Telecommunications SIG 96-02-01, Object Management Group, February 1996.
8. M. Arango et al.,"The Touring Machine System", *Communications of the ACM*, vol. 36, no. 1, pp. 68-77, January 1993.
9. V. Mak, M. Arango and T. Hickey,"The Application Programming Interface to the Touring Machine", Bellcore Technical Report, February 1993.
10. S. Gibbs,"Composite Multimedia and Active Objects", in Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91), pp. 97-112, New York:ACM Press, 1991.
11. ISO/IEC JTC1/SC21 Working Group 7,"Reference Model for Open Distributed Processing", ISO draft standard, Turino, November 1993.
12. S. Wray, T. Glauert and A. Hopper,"The Medusa Applications Environment", *IEEE Multimedia*, vol.1, no. 4, Winter 1994.
13. I. Barth,"Configuring Distributed Multimedia Applications Using CINEMA", in *Proceedings of the IEEE International Workshop on Multimedia Software Development*, Berlin, March 1996.
14. Digital Audio Visual Interoperability Council,"What is DAVIC", DAVIC home page, http://www.cnm.bell-atl.com/whatdav.html, 1995.
15. ISO/IEC JTC1/SC24 Committee Draft 14478-1,"Information Processing Systems - Computer Graphics and Image Processing - Presentation Environments for Multimedia Objects(PREMO)", ISO draft standard, January 1996.
16. Private E-mail from Steven J. Mitchell, Director of Systems & Information Mgt at IMA, March 29, 1996.
17. C. Blum and R. Molva,"A Software Platform for Distributed Multimedia Applications", in *Proceedings of the IEEE International Workshop on Multimedia Software Development*, Berlin, March 1996.
18. J. K. Ousterhout, "TCL and TK Toolkit", Addison-Wesley Publishing,1994.
19. J. Gosling and H. McGilton,"The Java Language Environment", Sun Microsystems White Paper, May 1995.
20. Kaleida Labs,"ScriptX Technical Overview", Kaleida Labs Technical Report, http://www.kaleida.com, 1995.
21. Object Management Group,"Trading Object Service", OMG Document orbos/96-05-06, Mai 1996.
22. 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.
23. M. Schmid,"Design and Implementation of a Connection Management Platform for Networked Multimedia Applications", diploma thesis at the University of Stuttgart, http://www.eurecom.fr/~blum/pub.html, Mai 1996.
24. Iona Technologies,"The Orbix Homepage at Iona Technologies", http://www-usa.iona.com/www/Orbix/index.html, 1996.
25. M.R. Macedonia and D.P. Brutzman,"MBone Provides Audio and Video Across the Internet", *IEEE Computer*, vol. 27, no. 4, April 1994.
26. G. Coulson and D.G. Waddington,"A CORBA Compliant Real-Time Multimedia Platform for Broadband Networks", in *Proceedings of the International Workshop on Trends in Distributed Systems TreDS'96*, Aachen, Germany, October 1996.

WWW

Find the APMT homepage at : http://www.eurecom.fr/~blum/proto.html

Find other APMT publications at: http://www.eurecom.fr/~blum/pub.html