An Open Middleware Architecture for Network-Integrated Multimedia

Marco Lohse, Michael Repplinger, and Philipp Slusallek

Computer Graphics Lab, Department of Computer Science,
Saarland University, Im Stadtwald, 66123 Saarbrücken, Germany
{mlohse, repplix, slusallek}@graphics.cs.uni-sb.de

Abstract. Today, we are surrounded by a constantly growing number of networked multimedia devices. These devices offer high-quality input and output capabilities often together with enough computing power and programmability to perform a variety of multimedia operations. However, integrating and controlling these distributed devices from an application is difficult because of the variety of underlying technologies.
In this paper we present a network-integrated multimedia middleware especially designed for this heterogeneous environment. Our architecture allows for a flexible usage of different networking technologies and offers the extensibility to transparently use various existing infrastructures. Distributed devices can be discovered, inspected, and then integrated into a common media processing graph. We demonstrate our approach with a distributed camera control application and a multimedia home entertainment center.

1 Introduction

Today's common multimedia middleware such as DirectShow from Microsoft [1] or the Java Media Framework from Sun [2] adopt a PC-centric approach. Applications can only access directly connected devices and the network is used as a source of data only. With the increasing number of networked multimedia devices like video recorders, set-top-boxes, TVs, hi-fi systems, multimedia PCs, or small screen devices like PDAs, support for distributed multimedia applications is becoming increasingly important. The goal of our work is to design and develop a multimedia middleware that considers the network as an integral part and enables the intelligent use of devices distributed across a network.

Several approaches for multimedia middleware supporting distributed computing have been proposed. These frameworks typically directly extend existing middleware for distributed object environments (DOE), namely OMG's CORBA or Microsoft's DCOM. The Multimedia System Services (MSS) [3] offer an architecture for building distributed multimedia application using CORBA. MSS is the basis for the ISO Presentation Environment for Multimedia Objects (PREMO) [4]. Like the Reference Model of Open Distributed Processing (RM-ODP) [5], PREMO is restricted to a high-level conceptual description rather than an approach for implementation. Other frameworks focus on specific problems

in distributed multimedia environments and propose sophisticated extensions for QoS management and adaption, e.g. the Multimedia Component Architecture (MCA) [6] or CINEMA [7]. The Toolkit for Open Adaptive Streaming Technology (TOAST) examines the use of reflection and open programming in distributed multimedia [8]. The approach described in [9] extends the standard CORBA Event Service to support multimedia and includes data types for multimedia data flows. In [10] an extension for the CORBA Event service is described which offers different levels of reliability, congestion control mechanisms and jitter suppression strategies. An architecture which allows mobile computers to use resources in a distributed environment is described in [11]. Like the above mentioned frameworks it is based directly on CORBA.

In contrast, our approach offers an integrating open architecture that does not rely an a particular technology or middleware. Instead, it allows the flexible usage of different networking and middleware technologies. The network-integrated multimedia middleware (NMM) [12] presented in this paper offers following advantages.

- **Heterogeneity.** In heterogeneous environments one cannot assume the availability of a certain DOE technology on all platforms. Instead, with mediating proxy objects and parameterizable communication strategies, different technologies can be combined. By recursively using this approach, even different technologies without a directly connecting communication channel can be used together, if some other technology can be used as mediator.
- **Optimized communication strategies.** The tight coupling of a multimedia middleware to an existing middleware for DOE might incur significant overhead, which is often permanent even for locally operating applications. The possible usage of optimized communication strategies allows to chose an appropriate technique depending on the current context or the locality of components. Furthermore, with this explicit binding mechanism, suitable strategies and QoS parameters can be chosen independently for the transmission of multimedia data and the controlling of components.
- **Extensibility.** Different extensions to existing DOE middleware as well as new multimedia middleware technologies will emerge in the future. Our middleware can take advantage of this progression by using these DOE technologies as new communication channels or by integrating new multimedia middleware functionality with suitable proxy objects.
- **Unified messaging system.** By providing a unified event system components can dynamically be queried for their supported functionality. Furthermore, application can register to be notified by components. Method invocation on interfaces exported by components have the same semantics as event passing.
- **Resource consumption.** On platforms with restricted resources like PDAs or set-top boxes, traditional DOE middleware often require too much computational power and memory capacity. By using light-weight transport strategies these platforms can still be integrated transparently.

2 Locally Operating Middleware

This section will describe the architecture and services offered for building multimedia applications operating on a single host. The next sections will then show how these aspects are extended for a network-integrated environment.

2.1 Flow Graph-Based Architecture

Within the network-integrated multimedia middleware (NMM), which is implemented in C++ under Linux, hardware devices (e.g. a DVD-ROM drive) as well as software components (e.g. a decoding module) are modeled by so called *nodes* (see Figure 1). A node represents the smallest entity of processing. In order to be able to extensively reuse specific nodes in different application scenarios, nodes should represent fine-grained processing units. The innermost loop of a node produces data, performs a certain operation on the data, or finally consumes data.

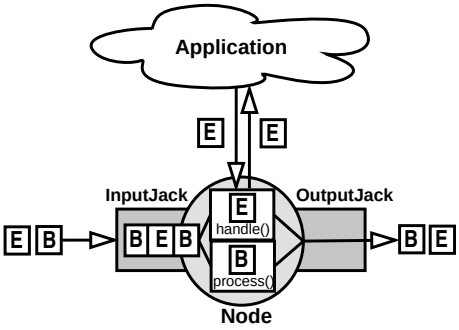

Fig. 1. NMM node with in-stream events and buffers, and out-of-band events sent from or to an application ('E' event, 'B' buffer). Buffers are processed inside the node and events are handled with registered methods.

A node has potentially several input and output ports, each represented by a *jack*. Associated with each jack are its supported *formats*, which precisely describe the supported type of multimedia data with type, subtype, and format specific parameters as key-value pairs. Multimedia formats can be described recursively. This allows for arbitrary complex formats. Formats characterize nodes with their capabilities in the registry and XML descriptions of formats can be exported during runtime and (see Section 2.4).

To perform a specific type of multimedia processing, nodes are connected to a directed graph, the *flow graph*. This is done by connecting output jacks to input jacks. The structure of this graph then specifies the operation to be performed.

Synchronization is performed by *Controller*-objects attached to nodes, which are in turn monitored by a *Synchronizer*-object. Several other services are provided by NMM that cannot be covered in detail in this paper.

2.2 Format Negotiation

Although a flow graph can be created by hand, our multimedia middleware provides support for setting up complex flow graphs. Instead of having to provide a valid and fully configured flow graph, only a high-level description of the wanted functionality has to be specified that is independent of possible connections and supported formats: the *user graph*. A format negotiation procedure then transforms the user graph into a valid and fully configured flow graph following a well-defined quality measurement as optimization criterion. The quality of a media presentation is defined as the quality perceived by a user. Solutions which provide the same quality, are ranked according to their costs. More details can be found in [13].

2.3 Messaging System

All communication within the NMM architecture is performed through an unified message system. This aspect is especially important because it allows to extend locally operating middleware like the one described in this section to incorporate distributed nodes without relying on a particular DOE middleware.

The message system consists of two types of *message* objects. Multimedia data is generated at source nodes and placed into messages of type *buffer*. Buffers are forwarded along connected jacks between nodes and can be enqueued before being processed (see *process()* in Figure 1). Jacks use a *transport strategy* that is responsible for forwarding messages from an input to an output jack. In a local application the transport strategy is a simple pointer forwarding. In Section 3.4 the role of transport strategies in distributed environments is described.

Messages of type *composite event* can be used to control node behavior. This can be done in two ways. Nodes can create *in-stream* composite events that are forwarded between nodes and possibly enqueued the same way as buffers. In addition, an application can communicate with its instantiated nodes by sending *out-of-band* composite events. This mechanism is mainly used for setting node specific parameters. A composite event consists of an ordered set of single events (or commands). Each event is identified by a key and contains additional parameters. To be able to handle a specific event, a node has to register a method. Events are then dispatched automatically (see *handle()* in Figure 1).

The unified event system allows to dynamically add and remove the ability to handle certain events. In addition, listener objects can transparently register to be informed when a certain event reaches a node. For example, an application can register to be notified when an "end-of-stream" event reaches a certain node. Furthermore, the event system provides reflection: nodes can be asked for all events they can handle together with the parameter types. For example, this is useful for applications providing a GUI for creating flow graphs and setting

node specific parameters. Here, the GUI elements for event parameters can be generated automatically from the parameter type.

During its lifetime, a node can be in different states. After being instantiated (*constructed* state), a node might require specific parameters to be set before its supported formats can be announced (*initialized* state). After the jacks of a node are connected, a node reaches the next state (*activated* state) where it is ready to start processing. A call to start() will then finally set the state of the node to *started*. In order to take the properties of these different states into account, the event system allows to specify two additional parameters to be set when registering an event handling method. First, the states in which an event can be handled; and secondly, the states in which the particular event must have been received and successfully handled before the next state can be reached.

Together, we consider the reflection property together with the possibility to register additional listeners and to restrict event handling to certain states to be essential parts of a multimedia middleware. This is especially important in distributed environments where new devices with unknown properties may become available at any time.

However, compared to a method call, sending an event to a node requires additional programming effort since the event must be created and filled with parameters explicitly. Additionally, since events are identified by strings, type-safety is not guaranteed at compile time. Therefore, our architecture adds interface classes on top of the event system. This step will be explained in detail in Section 3.

2.4 Registry Service

The basis for device discovery is a registry service. On each host, a unique *registry server* object administrates all local resources represented by NMM nodes. Here, information about all available NMM nodes are stored in internal data structures, which can be exported in XML format. Locally running application send queries in XML format to the registry server. After searching its database, the registry server returns a reply and eventually performs reservations of resources (including network and CPU). Objects are then instantiated by a *node factory* object. If the search was not successful, the request can be forwarded to other registries in the network using different peer-to-peer protocols. Details for this step are not covered in the scope of this paper.

The registry server not only administrates nodes. It can also include other specialized registries which are then requested recursively. These registries are useful for the management of devices with special properties (like Firewire devices, which may be connected to more than one host at a time) or for integrating devices from other multimedia frameworks (like the Java Media Framework).

3 Distributed Middleware Architecture

Within the architecture described in the previous section, the nodes connected to a flow graph are restricted to be in the same address space on a single host. In

order to be able to control and connect distributed nodes, a network-integrated multimedia middleware needs to provide several services [5].

3.1 Requirements

In the scope of this paper we focus on fulfilling the following requirements.

- **Network- and technology-transparent control.** An intermediary infrastructure has to be established that allows control of all distributed devices and components independent of their physical location and underlying technology.
- **Open components and reflection.** Due to the dynamics and the heterogeneity of distributed environments, components have to provide a self-describing notation of their capabilities and supported interface during runtime. Furthermore, application need to be able to register with components independently of their location and technology.
- **Network-wide cooperation.** Connections for streaming continuous multimedia data between distributed nodes as well as between different technologies must be possible to allow seamless cooperation.
- **Migration-transparency.** Nodes should be allowed to be relocated. This relocation process should be transparent to the application.

The network-integrated registry service described in Section 2.4 provides information about all available nodes and can instantiate nodes on the specific host upon request.

3.2 Proxy Objects

By providing a *proxy architecture*, our multimedia middleware is able to fulfill the above mentioned requirements. The term proxy object is generally used to describe an object that acts as a surrogate for another object to enable access to it [14].

The essential point in using proxy objects is that they allow to separate method invocation and execution. This allows for redirecting an invocation to a remote object. Migrating this remote object only requires updating the way in which communication is performed and not the proxy object itself. Furthermore, proxy objects can be used as translators between different technologies. Section 4 shows the integration of NMM and the Java Media Framework as an example. In addition, proxy objects can be used as interface classes on top of the event system. Section 3.3 will describe this step in detail. Method invocations on these proxy interfaces have the same properties as sending an event. Remember, events allow nodes to be queried for their supported functionality, additional listeners can be registered, and event handling can be restricted or required for certain states of a node. Therefore, local nodes and jacks as well as distributed nodes and jacks are represented as proxy objects within applications.

In order to represent the communication between two objects as a first class data type, we use *communication channels*, which are binding objects according

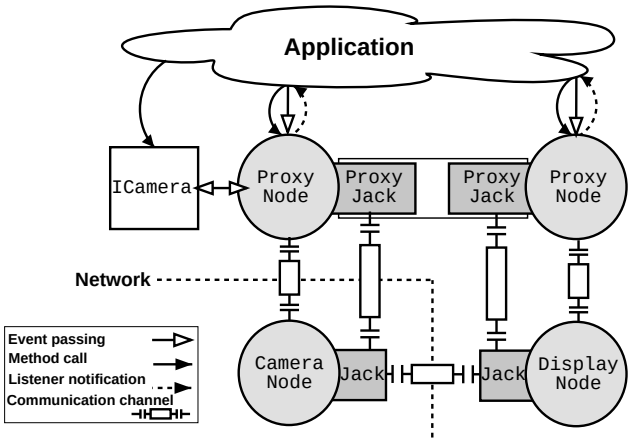

Fig. 2. An application controls a remote camera and a local display node through their proxies and communication channels. An interface object for the camera interface is also provided. Nodes are connected with a communication channel for streaming media.

to the RM-ODP [5]. The design of communication channels as a first class data type allows for a flexible plug-and-play design were the application can choose or change the *transport strategy* of a communication channel depending on its requirements. Communication channels are used in two ways. First, proxy objects control possibly remote objects by sending out-of-band events that were generated by an application or an interface. Secondly, communication channels can be used to transfer multimedia buffer objects or in-stream events between connected nodes.

Figure 2 shows the overall structure of a distributed flow graph. Here, a remote camera (CameraNode) is connected to a local display (DisplayNode). These two nodes are accessed by the application via their proxy nodes. Similarly, the jacks of the nodes are controlled through corresponding proxy jacks.

All these proxy objects are connected to their implementation objects by communication channels. Since the camera is a remote object in this example, a network transport strategy is negotiated for the camera node and its jack upon requesting the node from a remote registry. Jacks represent the binding between two nodes. If the application connects the two proxy jacks, a network transport strategy is negotiated because the two nodes are not collocated. Details on this process are described in Section 3.4

In addition to the general node interface, the camera provides a specific interface. This interface is requested by the application and an interface object is dynamically created (e.g. ICamera). The application can control these proxies by passing an event directly or by invocating the corresponding method on the interface object. Method calls are forwarded as events to the corresponding proxy node. Furthermore, the application can register to be notified for received events.

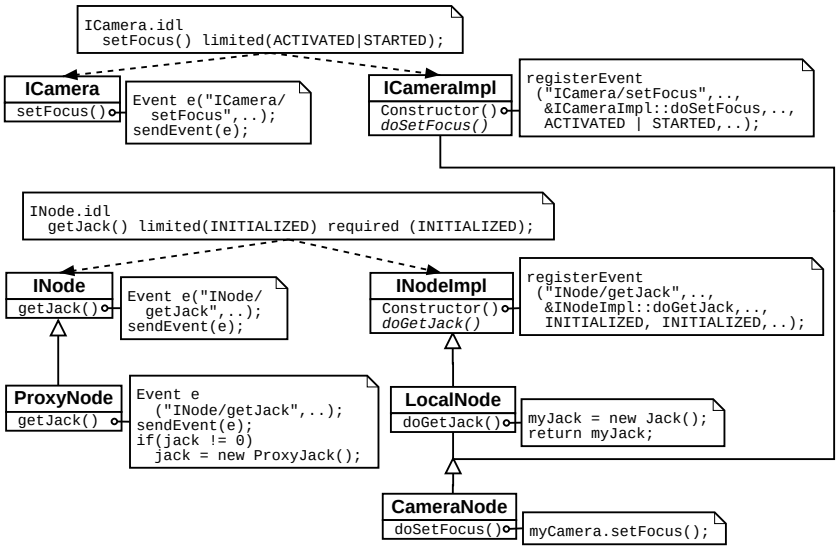

Fig. 3. Simplified IDL notation for interfaces and UML diagram for proxy classes and implementation classes.

3.3 Interface Definition

As mentioned above, the unified event system offers reflection and the possibility to register additional event listeners. Furthermore, event handling can be restricted to certain states of a node and events can be specified as required to reach another state. In order to provide type-safe interfaces for application development, our architecture offers automatically generated interfaces that provide the same semantics as sending an event.

Interfaces are described in an interface definition language (IDL) which is an extended subset of CORBA IDL. Methods can additionally be restricted to a certain state or marked as required to reach another state (limited or required). For each interface, a proxy class and an implementation class is generated. The proxy class (prefix I) contains public methods which internally generate the corresponding event. The implementation class (suffix Impl) contains a protected do-method for each public method in the proxy class. These methods are to be implemented by subclasses. Within the constructor, do-methods are registered for handling the corresponding event.

Figure 3 shows as a simplified example the corresponding classes for NMM nodes and for a camera interface in UML notation. The setFocus-method in ICamera.idl is limited to two states and the getJack-method in INode.idl is limited to the initialized-state and must also be called in this state. The CameraNode inherits from LocalNode and ICameraImpl and finally implements the setFocus-method. Notice, that ProxyNode re-implements the getJack-method: instead of a local jack, a ProxyJack-object is returned.

3.4 Object Binding and Communication Channels

Within our architecture, communication channels are used to transport objects. Objects are mainly of type message but it general any user-defined type can be transferred.

As mentioned above, our architecture allows the flexible usage of different transport strategies within a communication channel. Communication between objects in the same address space is implemented as pointer forwarding. Communication over a network can be performed using different transport strategies. In any case, serialization of objects on the sender side and deserialization on the receiver side is needed.

For these steps our architecture uses so called *ValueStreams*. The ValueStream interface provides input and output operators (i.e. the $\ll$ and $\gg$ operators in C++) for classes of type *Value*, which include classes for basic types and for sequences of Value-objects. With these types, arbitrary complex objects can be built, which are automatically serialized and deserialized without additional programming effort. For instance, the buffer class and the event classes are subclasses of Value. A buffer is a sequence, which includes an array of bytes for the multimedia data and additional value types for information like the size and timestamp. For user-defined classes the ValueStream operators have to be implemented or a mapping between user-defined types and the generic value types has to be provided.

By choosing different subclasses of ValueStream for serialization and deserialization the data format for transmission is selected. One option we are currently providing is to generate an XML representation, another is a representation with "magic numbers" for different data types. Both representations can be directly transferred over a socket using protocols like TCP, UDP, or RTP. Another option is a transport strategy together with a ValueStream subclass that uses CORBA any-types for transmission. We currently use The Ace Orb (TAO) [15] which also provides real-time extensions. Other possible middleware technologies we are planning to evaluate are the Multimedia Communication Protocol (MCOP) [16] and the Desktop Communication Protocol (DCOP) [17] used in the KDE project.

Our architecture supports explicit binding where the strategy of a communication channel and the ValueStream class to be used can be selected or automatically negotiated. In order to simplify this process we use *binding factories* that try to negotiate a connection with wanted properties like QoS requirements. If a direct connection to a distributed object fails the factory searches the network for available proxy objects and communication channels. Based on this result the factory tries to negotiate a connection through more than one proxy objects. If the negotiation process is successfully completed the factory requests the necessary resources and configures them depending on the given requirements. This process is repeated if a node is migrated during runtime.

Communication channels provide interfaces which can be used to realize the mapping of QoS requirements to certain network settings. They also allow the registration of event handlers that are called if certain settings are no longer

maintained. Upon these facilities, advanced services like dynamic QoS adaption can be built.

3.5 Discussion

Although CORBA and other DOE systems also offer a proxy architecture, our framework only uses these facilities as one possible solution among others for realizing access to remote objects. Our approach provides an open architecture that offers the transparent and flexible usage of different networking and middleware technologies.

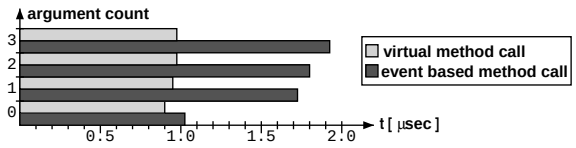

Fig. 4. Execution time per call: dynamic cast plus method invocation compared to event handling (measured for 10,000,000 invocations, 1 GHz Linux PC).

With the unified event system, nodes can be queried for their supported functionality, additional listeners can be registered, and event handling can be restricted or required for certain states. Despite this extra functionality, our current non-optimized implementation shows good performance compared to a virtual method call with a preceding dynamic cast (see Figure 4). We added the dynamic cast to simulate the typical usage of interfaces.

4 Applications

Within the NMM framework, we have implemented a distributed control application for Firewire cameras in C++ that allows to access all cameras registered in the network. The upper half of Figure 5 shows the user interface of this application running on a Linux PC. The features of the currently activated camera are requested dynamically and the user interface is configured appropriately. A distributed camera source node is connected to two sink nodes for rendering the video image. The lower part of Figure 6 shows the image captured by the currently activated camera as rendered by these two different video output nodes. On the left side, the video is rendered with a NMM video display node for the X window system. On the right side, the video is rendered with the Java Media Framework (JMF). To integrate this middleware, we provide the node and jack implementation classes and communication channel based on sockets in Java. Messages that were sent from the connected jack or the corresponding proxy

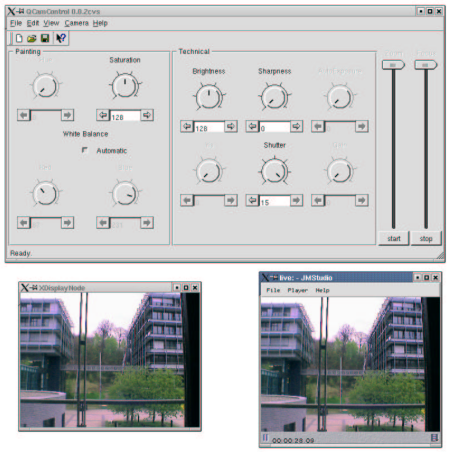

Fig. 5. Distributed camera control application with X display and JMF display.

Fig. 6. The Multimedia-Box, a multimedia home entertainment PC.

object are received by this communication channel and then mapped to JMF buffers or control commands, respectively.

Based on NMM, the Multimedia-Box application offers an extensible application framework for a home entertainment center based on a Linux-PC (see Figure 6). The Multimedia-Box provides the following functionality: TV and video recorder with time-shifting, DVD replay, audio-CD replay, and MP3 encoding and replay. All these features are realized as flow graphs of NMM nodes. Furthermore, the Multimedia-Box can extend its functionality by using devices distributed across the network, e.g. a remote DVB-board can be used as a new data source or time-consuming tasks like video transcoding can be distributed to other devices or specialized hardware.

5 Conclusions and Future Work

In this paper we presented a network-integrated multimedia middleware (NMM) for heterogeneous distributed environments. Our middleware offers network-transparent control and network-wide cooperation of components with different underlying technologies. Based on a unified messaging system, distributed components can be queried for their supported capabilities and interfaces. An open and extensible design allows to integrate new technologies and existing middleware solutions. Two applications were demonstrated; one using distributed components of the NMM architecture together with components of the Java Media Framework.

Future work will include the study of different communication strategies in terms of performance and network bandwidth. Quality of Service monitoring and probing together with advanced QoS negotiation and adaptation will complete

our approach. The locally operating middleware and a number of plug-in nodes is already available as Open Source; other parts will follow.

6 Acknowledgements

This research has been supported by Motorola, Germany, and the Ministry of the Saarland.

References

1. Microsoft: DirectShow Architecture. http://msdn.microsoft.com/ (2002)
2. Sun: Java Media Framework API Guide. http://java.sun.com/products/java-media/jmf/ (2002)
3. Hewlett-Packard Company and IBM Corporation and SunSoft Inc.: Multimedia System Services (1993)
4. David Duke and Ilvan Herman: A Standard for Mulimtedia Middleware. In: ACM International Conference on Multimedia. (1998)
5. Gordon Blair and Jean-Bernard Stefani: Open Distributed Processing and Mulitimedia. Addison-Wesley (1998)
6. Waddington, D., Coulson, G.: A Distributed Multimedia Component Architecture. In: IEEE International Workshop on Enterprise Distributed Object Computing. (1997)
7. Rothermel, K., Dermler, G., Fiederer, W.: QoS Negotiation and Resource Reservation for Distributed Multimedia Applications. In: IEEE International Conference on Multimedia Computing and Systems. (1997)
8. Tom Fitzpatrick and Julian Gallop and Gordon Blair and Christopher Cooper and Geoff Coulson and David Duce and Ian Johnson: Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware. In: International Workshop on Interactive Distributed Multimedia Systems. (2001)
9. Chambers, D., Lyons, G., Duggan, J.: Stream Enhancements for the CORBA Event Service. In: ACM International Conference on Multimedia. (2001)
10. Orvalho, J., Boavida, F.: Augmented Reliable Multicast CORBA Event Service (ARMS): A QoS-Adaptive Middleware. In: International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services. (2000)
11. Seitz, J., Davies, N., Ebner, M., Friday, A.: A CORBA-based Proxy Architecture for Mobile Multimedia Applications. In: International Conference on Management of Multimedia Networks and Services. (1998)
12. Network-Multimedia Workgroup: Network-Integrated Multimedia Middleware. http://www.networkmultimedia.org (2002)
13. Lohse, M., Slusallek, P., Wambach, P.: Extended Format Definition and Quality-driven Format Negotiation in Multimedia Systems. In: Multimedia 2001 – Proceedings of the Eurographics Workshop. (2001)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
15. Douglas C. Schmidt et al.: The Ace Orb(TAO). http://www.cs.wustl.edu/∼schmidt/TAO.html (2002)
16. Stefan Westerfeld: Multimedia Communication Protocol (MCOP) documentation. http://www.arts-project.org/doc/mcop-doc/ (2002)
17. Preston Brown et al.: Desktop Communication Protocl (DCOP) documentation. http://developer.kde.org/documentation/library/dcop.html (2002)