MANAGEMENT AND CONTROL OF DISTRIBUTED MULTIMEDIA DEVICES AND STREAMS THROUGH OBJECT-ORIENTED MIDDLEWARE

Reinhold Eberhardt, Christian Rueß, Jochen Metzler

DaimlerChrysler AG, Research and Technology 3

PO Box 2360, 89013 Ulm, Germany

Phone: ++49 731 505 2103

Fax: ++49 731 505 4110

{Reinhold.Eberhardt, Christian.Ruess, Jochen.Metzler}@DaimlerChrysler.com

Abstract

In this paper an architectural concept for the management of a distributed multimedia environment based on object-oriented middleware is presented. The system enables remote control of distributed multimedia devices such as MPEG encoders, decoders, video servers and file systems with multimedia content. Through this approach, the multimedia streams between the devices can be bound to an appropriate communication infrastructure, such as native ATM, RSVP or another network which may or may not be QoS aware. The system is easily extendable and provides a generic framework for the support of a variety of multimedia devices and communication infrastructures. Experiences using a prototype of the concept are given.

Keywords: Object-oriented distributed networking, multimedia communications, application and network QoS, ATM and IP QoS

1. INTRODUCTION

The provisioning of multimedia content is gaining increased importance in many areas of our daily life. Product presentation and visualization as well as learning environments are only a few examples. Furthermore, multimedia products such as hardware encoders and decoders are now available for everyone at reasonable prices. The speed of current computing systems is sufficient so as to permit real-time decoding and encoding of high-quality

videos. On the other hand, computer networks provide the speed required to support a distribution of the multimedia devices and their sharing.

In order that a distributed multimedia environment may be supported, an architecture for the control and the management of such systems is required. Not only is there a need to manage the multimedia devices but there is also a requisite to control the QoS-aware communication system between the devices.

This paper sets forth a system architecture geared for the management and control of distributed multimedia devices and streams through object-oriented middleware. The system proposed is largely derived from the CORBAtelecoms specification [2], yet has the added benefit of reducing or extending it to fit our requirements as needed. The system is based on a CORBA platform with Java language mapping [1], [3], [4]. However, native code is integrated in the Java platform to support native ATM and device-specific libraries. For performance reasons, multimedia flows are directly transferred within the native code without propagation to the higher Java or CORBA level, thereby creating an interface to control the multimedia traffic from Java instead of a Java ATM API [7].

Section 2 shows how the user could make use of such a platform. In Section 3 a short overview of the architecture is presented. Section 4 goes into detail on several of the possible concepts for implementation of the system. The last two sections summarize related and future work.

2. THE SYSTEM AS SEEN BY THE USER

The users of a distributed multimedia system would like to be able to deploy the available resources, e.g. a remote encoder and the intermediate network. If the resources can be accessed remotely via a well-defined CORBA interface, several possible scenarios are generated:

Client-driven control: Should a user at a client PC with an MPEG-2 decoder desire to see MPEG-2 encoded material from a camera or VCR located at a remote location, the client could be utilized to set up the network connection and then configure and control the remote and local devices. In this scenario, the management of the system is driven solely by the client.

Server-driven control: The management and control of the devices and streams could also be driven by a server. For example, when using a camera to observe a security area and a motion sensor to detect any movement in this area, the motion sensor could trigger the transmission

of pictures taken by the camera to a specific client, e.g. the security inspector responsible.

Manager-initiated control: In the third scenario, management and control is performed by a management application outside the client and server. Consider a telelearning scenario with an MPEG encoder in a central studio and several decoders in classrooms interconnected by a heterogeneous network infrastructure: when the session starts, a manager configures and controls all of the devices and sets up the intermediate network connections.

The objective of the system is to provide the flexibility needed to interconnect arbitrary source and sink devices across an arbitrary network infrastructure. The users can express their requirements and choose from a list of devices and communication systems those they want to use. The mapping of user requirements with available services is carried out via a trading service.

3. THE ARCHITECTURE

The architecture of our system reduces the CORBAtelecoms specification [2], on one hand, and, on the other hand, also extends it to meet our specific requirements. In our approach, the devices only support one single, unidirectional data flow at a time. This may change when the system is extended. A device used in a connection is classified as either a source or a sink object. A device can support both directions, e.g. a fileserver; however, during the setup of a connection, the direction of the flow is selected. In the same manner, streams are simplified to only support a single, unidirectional data flow.

In the initial state of the system, the servers running on the hosts provide factory objects for the different devices and StreamEndPoints (SEP). The factory objects are registered within the CORBA naming service and provide a CORBA interface.

For portability reasons, we use Java as a programming language for the implementation of the different services, as many of the APIs that come with encoder, decoder and ATM network interface card merely provide support for C/C++. For this reason, as well as for performance reasons, some parts of the system have to be programmed in C/C++ and integrated in Java via the Java Native Interface (JNI) [9]. This leads to a layered architecture, consisting of a high-level CORBA interface, accessing a lower level Java implementation that uses native code to achieve access to the lower level resources.

4. THE PROTOTYPE

4.1 CORBA Interfaces

In order that different devices may be included in the system, interface inheritance was used at the CORBA level. A base interface is utilized for the joint operations of all of the devices. The interfaces of those devices which should be included have to inherit from this basic interface “device”:

```
enum DeviceOperation {start, stop, pause};
enum VideoFormat {MPEG1, HalfD1, FullD1};
enum StreamDirection {Source, Sink};
interface Device {
    void operation(in DeviceOperation op);
    void bind(in StreamEndPoint sep);
};
interface Encoder : Device {
    void set_format(in VideoFormat format);
    void set_bitrate(in double kbps);    ...
};
interface Decoder : Device { ... };    ...
```

The same approach was deployed for the StreamEndPoints: a single base interface provides the common operations for all of the SEPs:

```
enum StreamDirection {Source, Sink};
interface StreamEndPoint {
    void set_other(in string other);
    void set_direction(in StreamDirection d);
    void connect();
};
interface SEP_ATMCBR : StreamEndPoint {
    void set_bitrate(in double kbps);
};
```

4.2 Java Implementations

As a rule, the CORBA interfaces are implemented in Java classes. Should it become necessary to access specific native libraries in order to access the hardware in turn, the Java native interface is used (see next section). Nevertheless, the CORBA interface maps to a Java implementation class in the first step. Certain methods of the Java class may be implemented as native methods.

The CORBA interfaces of the different classes represent the control interface of the system. The actual data transmission is performed beneath this surface. When using native SEPs and devices, data can be transmitted direct at native level. If the SEPs and devices are pure Java objects, the transmission can be carried out in Java. A mixture of different kinds of objects necessitates data propagation from native code to Java and vice versa. Any combination has to be supported.

Two major aspects have to be considered for direct data transmission:

- The implementation of the SEP must have a direct reference to the implementation of the local bound device.
- Devices and SEPs must provide methods for a direct hand-over of data to each other.

The first aspect is realized through a table which maps a CORBA object reference (an IOR) to a Java object which implements this interface. The same approach is used for the StreamEndPoint objects. When a new SEP is instantiated from a factory, a corresponding entry is made in the table. When a StreamEndPoint IOR is passed to a device object via the bind method, the device consults the table and acquires the implementation object and then registers its own implementation at the SEP, thereby ensuring that the implementation of the SEP knows the implementation object of the device and vice versa.

Two modes are taken into consideration for the direct data transmission between the implementation objects:

- Push: data can be pushed from an originating device into an SEP for transmission and from a destination SEP to a device.
- Pull: an SEP can retrieve data from a device at the sender, or the receiving device could poll the data from the sink SEP.

Exactly which mode is used depends on whether or not a specific bandwidth is prescribed by the SEPs and devices. For example, an encoder is configured for a specific bandwidth, and the connection to the sink SEP supports this bandwidth. In this case it is much more effective for the encoder to push the data to its associated SEP. If a FileSender is used as source and data is to be transmitted at the maximum data rate supported by the connection to the other SEP, then the SEP should pull the data from the FileSender.

To support push/pull models, both the SEPs and devices support a push and a pull method for the forwarding of data at the Java level. The methods may be implemented using native methods. In addition, these methods may be called direct at native level via a similar mechanism.

4.3 JNI Interfaces

Generally, the Java programming language is used to implement the CORBA objects with the given interfaces. Hardware-specific native code is integrated via the Java Native Interface. Special JNI wrapper classes are utilized to provide an interface to the lower level services in Java.

In the first step, the transmission of MPEG-2 material from an Optibase encoder to an Optibase decoder over ATM CBR is implemented in a prototype. Decoder, encoder and the ATM networks interface card are accessed through native code. Consequently, we decided not to propagate any multimedia data to the Java level. Instead, we hand the data from the encoder over direct to the ATM library, and, at the decoder side, forward the data direct from the ATM library to the decoder board. The JNI therefore provides a Java interface for the control of the data flow on the native layer.

Figure 1 Integration of Devices with JNI

One major task of the JNI other than the provision of access to a hardware device is to manage how the multiple virtual devices at the CORBA level access the joint hardware and how the link between a multimedia device and a *StreamEndPoint* is achieved. Figure 1 roughly illustrates the components within the JNI layer in one of the host systems.

Each time a new *StreamEndPoint* is to be created, a new *SEP* record is inserted in the configuration manager. These records are globally available within the server process for all kinds of *SEPs*. The records include all the relevant information for a *StreamEndPoint*, such as the type of the *SEP* and a reference to the corresponding socket for the transmission of data. A *JNISStreamEndPoint* directly corresponds to an *SEP* at the CORBA level. When a new *JNISStreamEndPoint* is created, a handle (integer value) identifying the position of the *SEP* record within the configuration manager is propagated to the Java level.

When a bind operation to a multimedia device is performed on the same host, this handle is passed to the native code of the device (JNIDevice). After this call to bind, the StreamEndPoint and the device on one host are “connected”, which means that they know how to access each other at the native code level. If, for example, an encoder device is already bound to an ATM CBR StreamEndPoint, when the start operation is performed on the device object, the device object receives the SEP record utilizing the handle from the bind call. Depending on the data in the SEP record, the native code then forwards the data to the corresponding network device. In this case, the socket data structure for the ATM CBR connection would be used.

This mechanism enables the interconnection of different devices with different SEPs. The JNI interfaces provide those operations required for the propagation of the calls in the CORBA interface (start, stop, pause, etc.).

4.4 JMF Integration

Within JMF (Java Media Framework) [10], some player objects (JMF devices) which allow the playback of registered media types as well as the registration of custom media types and players are made available. Table 1 shows the media formats and protocols supported:

Table 1 JMF 1.0 Media Formats [10]

Audio	AIFF, AU, DVI, G.723, GSM, IMA4, MIDI, MPEG-1 Layer 1 & 2, PCM, RMF, WAV
Video	Apple Graphics (SMC), Apple Animation (RLE), Cinepak, H.261, H.263, Indeo 3.2, Motion-JPEG, MPEG-1
Files	AVI, QuickTime, Vivo
Protocols	File, FTP, HTTP, RTP

Future versions will add capture, transmit and transcode functionality in JMF (JMF 2.0) [10]. The access to the JMF playback functionality is provided in different abstraction layers within Java. In the highest layer, the player can be started via a *URLStreamHandler* which specifies the location and the transport protocol by means of a URL (Unified Resource Locator). Although this is the simplest way to access the media player, this method provides less flexibility. To be able to deploy our own QoS-aware transport protocol as well as our own device control, we access the media player at a lower layer. Figure 2 shows the objects involved and their dependencies. We build our *OwnDataSource* and *OwnSourceStream* objects based on the *DataSource* and *SourceStream* class that will be connected, for instance, with our *nativeATM StreamEndPoint*. The *OwnDataSource* object uses our JNI of the QoS-aware *StreamEndPoint* device. Registration of the

nativeATMSEP is achieved through the *getStream* method from the *PullDataSource* class. A push model is also available. The *OwnDataSource* object is registered in the *Manager*. Using the return of the *getContentType* method, the *Manager* selects an appropriate *Player* and registers the *OwnDataSource* object within this *Player*.

Figure 2 Integration of JMF Player Functionality

4.5 Interoperability of Java and Native Interfaces

For portability reasons, it is our objective to program the majority of the interfaces in Java. As mentioned above, there are some valid reasons for using native code. The four possibilities of connecting a device with a *StreamEndPoint* are shown in Figure 3. Each native object has its representation on the Java side. Therefore, the connections between Java and native objects are as simple as Java-Java connections. Connections between two native objects are more complicated, as only their references have to be passed through the Java layer and the data is directly transmitted between the native objects.

Figure 3 Interoperation of Java and Native Interfaces

5. THE PROTOTYPICAL IMPLEMENTATION AND RELATED WORK

The prototypical implementation consists of a management application that allows the remote configuration and management of the communication system and the multimedia devices with the corresponding server counterparts. Currently, the management application enables the setup of a connection between two devices. A FileSender, FileReceiver, MPEG-2 encoder and decoder are supported. As communication links, an ATM CBR connection and a best-effort TCP connection are implemented. After a setup process, a GUI allows remote access of the devices.

Within the context of CORBA, an architecture for the control and management of audio and video streams [2] has been specified. Initial implementations which partially support this standard are presently in the pipeline [5], [6], but are in an early stage of development so that adjusting them to our needs would be a fairly complex task. Therefore, our approach follows the standard as far as is suitable for our specific environment.

To support ATM within Java, a Java ATM API is under specification by the ATM Forum [7]. As stated earlier in this paper, our goal is to control the flow of multimedia data from the Java level. The flow itself is visible solely at the native level. Nevertheless, for integration with those multimedia devices already available in the Java platform, for example with the Java Media Framework (JMF), it is necessary to propagate the multimedia flow up to the Java level. In [8], we have described how this can be accomplished.

6. SUMMARY AND FUTURE WORK

The paper describes several architectural concepts for the support of a distributed multimedia environment. The architecture is based on CORBA and Java and enables the remote configuration, control and management of the multimedia devices deployed and the streams between them. Devices which are not accessible from the Java platform are integrated using the Java

Native Interface. The object-oriented character of our architecture enables the transparent integration of the various devices and StreamEndPoints.

In the future, we plan to extend the application to support additional devices, such as a video server (SGI Mediabase), as well as additional StreamEndPoints such as RSVP. In addition, we are striving to make multimedia flows available at the Java level to support a wider range of applications generically, so that either mechanism may be used as an alternative.

References

- [1] "The Common Object Request Broker Architecture and Specification, 2.0 ed.," Object Management Group July 1995.
- [2] D. McGrath, T. Rutt, and J. Ottensmeyer, "Control and Management of Audio/Video Streams," Object Management Group telecom/97-05-07, May 1997.
- [3] D. Curtis, "Java, RMI and CORBA," OMG 1997.
- [4] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*. New York: John Wiley & Sons, 1997.
- [5] Iona Technologies, "OrbixMX - A Distributed Object Framework for Telecommunication Service Development and Deployment," White Paper, April 1998.
- [6] N. S. Sumedh Mungee, Douglas C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," Proc. of HICSS-32 International Conference on System Sciences, minitrack on Multimedia DBMS and the WWW, Hawaii, 1999.
- [7] ATM Forum SAA/API Working Group, T. Jespen and J. Shaffer, "Java ATM API Description - Proposed Outline, Revision 1," ATM Forum ATM Forum/97-1044R1, February 1998.
- [8] R. Eberhardt, C. Rueß, and R. Rusnak, "Communication Application Programming Interfaces with Quality of Service Support," Proc. of IEEE International Conference on ATM (ICATM '98), Colmar, France, 1998.
- [9] "The Java Native Interface Specification (JDK1.1)", Sun Microsystems, May 1997, <http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html>
- [10] "The Java Media Framework API", Sun Microsystems, <http://java.sun.com/products/java-media/jmf/index.html>