An Architecture for Multimedia Data Stream Handling and Its Implication for Multimedia Transport Service Interfaces

Ralf Guido Herrtwich

IBM European Networking Center
Tiergartenstr. 8
D-6900 Heidelberg
Germany

Abstract. *A multimedia software system architecture should reflect that — for performance reasons — digital audio and video data is routed through different paths in a computer than traditional data. We propose a multimedia architecture based on the notion of "stream handlers" which generate, filter, forward or consume multimedia data within a real-time environment. Stream handlers provide control operations to determine the content and direction of a multimedia stream. A multimedia transport system is discussed as an example of a multimedia stream handler.*

1.0 Introduction

The large volume and tight timing constraints of digital audio and video data have led to software architectures of multimedia systems where data obtained from an input device (e.g., a CD-ROM) is delivered to an output device (e.g., a video decompression board) across the fastest possible path. If no processing of data by the computer is needed, the preferred way of moving digital multimedia data through the system would be a direct device-to-device transfer, avoiding as much intermediate storage as possible. Today's device adapters usually do not support this mode of operation, but some of the device attachments currently built include this feature.

Multimedia programming systems already assume the paradigm of "streaming" data from sources to sinks: Proposals for programming multimedia applications (cf. [1, 2]) commonly assume that the user opens devices, establishes a connection between them, then starts the data flow, and returns to other duties. Later the data flow is stopped, the connection is dissolved, and the devices are closed. A typical application program would look like this:[1]

```
handle_in = create ("audio_source");
handle_out = create ("audio_sink");
stream = connect (handle_in, handle_out);
open (handle_in, "/dev/microphone");
open (handle_out, "/dev/speaker");
start (stream);
 ...
stop (stream);
close (handle_in);
close (handle_out);
disconnect (stream);
dispose (handle_in);
dispose (handle_out);
```

In the following, we investigate the consequence of this approach for distributed multimedia systems. In such systems, an application obtains multimedia data from the network through the *transport service interface* (TSI). TSIs common today, e.g., the Berkeley Socket Interface or the System V Transport Layer Interface (aka. the X/Open Transport Interface, XTI), require the user to take each data packet and forward it explicitly through *send* and *receive* functions. This violates the scheme that multimedia data should flow without the application's immediate involvement. It also constitutes a potential performance bottleneck: The TSI typically is an entry point into the operating system kernel. Each function call and each data item passing across the TSI implies a user-kernel transition.

In this paper, Section 2 introduces an architecture for handling multimedia data in a computer system. This architecture is then applied to a multimedia TSI in Section 3, using XTI as a starting point. This work is part of a project at the IBM European Networking Center in Heidelberg to develop and implement the Heidelberg High-Speed Transport System (HeiTS), an end-to-end communication system for multimedia data transfer [3, 4] While the findings of

[1] At a higher level of programming, in particular at the user interface, one would, of course, rather use graphical abstractions for connecting multimedia objects.

0-8186-2755-7/92 $03.00 © 1992 IEEE

this paper resulted mainly from investigating the integration of HeiTS into a UNIX environment, they should be applicable to other system platforms as well.

2.0 Handling Multimedia Data Streams

Characteristical for multimedia data are the temporal dependencies of its presentation to a human user. To take them into account, multimedia data should be handled in a *real-time environment* (RTE), i.e., its processing should be scheduled according to the data's inherent urgency. On a multimedia computer system, the RTE coexists with a *non-real-time environment* (NRTE) which deals with all data that has no timing parameters associated with it. Most of today's computer systems contain an NRTE only.

Multimedia I/O devices in general are accessed from both environments: Data such as a video frame is passed to them from the RTE, whereas control operations such as a camera zoom come from the NRTE. From an application viewpoint, the RTE is shielded by the NRTE: The application only performs control functions and is not involved in the actual data transfer. To optimize the function of the RTE, a *buffer management subsystem* as described in [5] should be used to minimize data flow across the bus, to avoid data copying in memory, and to facilitate direct adapter-to-adapter data transfers.

2.1 Stream Handlers

Any entity handling multimedia data in the RTE is called a *stream handler*. Typical stream handlers are filter and mixing functions, but parts of a communication subsystem through which multimedia data passes can be treated in the same way. Each stream handler has *endpoints* for input and output through which *data units* flow. The stream handler consumes data units from one or more input endpoints, it generates data units through one or more output endpoints.

Multimedia data usually enters the computer through an input device, a *source*, and leaves it through an output device, a *sink* (where storage can serve as an I/O device in both cases). Sources and sinks are operated by device drivers. To interface to stream handlers, each device driver contains some stub software that consumes or produces data units just like a regular stream handler does. We consider this software as a stream handler in its own right. Such a stream handler has either only input or only output endpoints. Stream handlers with only one kind of endpoints also occur when data is generated by the computer itself, i.e., calculated or interpreted by the CPU (e.g., in visual simulation or speech recognition systems).

Applications access stream handlers by establishing *sessions* with them. A session constitutes a virtual stream handler for exclusive use by the application which has created it. Depending on the required quality of service (QOS) of a session, an underlying *resource management subsystem* as described in [6] multiplexes the stream handler among the sessions.[2] To provide service guarantees (e.g., to obtain guarantees on how long it takes for some input data unit to be transformed into an output data unit), a workload specification for the session is needed. It can be derived from the data units consumed and generated in the session. To describe the arrival of data units at a session endpoint, a model such as *linear-bounded arrival processes* can be used [7]. QOS can be described in terms of session throughput, reliability and delay as explained in [8].

In a typical system, multimedia data flows repeatedly from output endpoints to input endpoints. The connection of endpoints forms an acyclic, unidirectional graph as described in [9]. Connections can be established on either a stream handler or a session basis.

- Connections based on stream handlers are static, they cannot be modified by applications. A typical static connection is, e.g., established between a Token Ring adapter and the multimedia network entity above it. If only one multimedia transport entity exists above the network entity the connection between both entities should also be static. Stream handlers with fixed connections appear to the application as a single stream handler.
- Session-based connections are dynamically created by the application through corresponding control functions. The inclusion of a mixer function to direct several audio inputs to one audio output is a typical example of a dynamic connection. Whenever the same stream handler can interface to a variety of other stream handlers it should be connected dynamically.

Beyond static or dynamic connection, the granularity of stream handlers is up to the system designers. The following trade-offs can be observed: A finer granularity offers the potential for tighter resource management and better resource utilization (depending on the management strategy), but more management overhead for stream handler connection – and potentially larger overall connection setup times. A

[2] In fact, it multiplexes the capacity of the underlying physical resources.

coarser granularity reduces management work, but may lead to weaker resource utilization [10].

There are several ways to implement sessions and their connections. The straightforward manner is to associate one thread with each session and to use buffers for communication between sessions. This causes unnecessary overhead for (common) cases where data units are merely forwarded from session to session with only minor modification. An example is the passing of an incoming message from network to transport entity and finally to an audio output device. In this case, a single thread could accompany the message through the entire system, avoiding the overhead of message passing. The decision on the use of buffers or threads can be based on the rate of thread execution in one session.

All threads in the RTE are real-time scheduled. Several real-time scheduling techniques lend themselves to be used for periodically recurring thread executions: Rate-monotonic and earliest-deadline-first scheduling can both be used [11]. These techniques are also characterized by simple schedulability tests which leads to faster session establishment. Whether the preemptive or non-preemptive versions of these scheduling techniques are used depends on the context switching overhead in the particular system [12].

2.2 Stream Handler Control

To manage the RTE data flow through the stream handlers of a multimedia system, control operations are used which belong to the NRTE. The functions form the *stream management subsystem* within our multimedia architecture. Two classes of operations are distinguished: those provided by all stream handlers and those specific to the individual stream handler.

Stream-handler specific operations usually determine the content of a multimedia stream and apply to particular I/O devices. Devices may be grouped into classes and their control operations may be derived generically as suggested in [13]. For storage devices, typical control operations include *fast_forward*, *reverse* and *seek*. Other operations are *zoom* for cameras and *volume* for speakers.

Operations to establish sessions and to connect their endpoints are provided for all stream handlers. In the sequence of their execution, these operations include:

- *create/dispose* (applied to stream handlers, yielding a session including its endpoints),
- *connect/disconnect* (applied to pairs of session endpoints, yielding stream identifications), and
- *start/stop* (applied to stream identifications).

An additional set of common operations are the *open/close* functions. *open* is executed after a connection to another stream handler was established, but before the data stream is started. It is used to associate a certain QOS with the session; reservation needed to achieve this quality is a consequence of this operation. *open* also provides stream-handler specific parameters to enable this reservation and to specify the exact session operation. In a file system, *open* determines the file to be accessed. In a transport system, it provides the address of the communication partner. In a video display, it names the area where to display the data on the screen.

A separation of the *create* and *open* operation (uncommon for most I/O abstractions such as the file system) takes care of the following problem: The reservation of a session often requires to know to which other sessions this session is connected (as I/O behaviour may be different depending on the connection). Yet, a connection cannot be made before the objects to be connected have been created. In addition, once the local connections of stream handlers are known, reservation in a distributed system (involving multimedia transport stream handlers on the different nodes) can commence in a single path (see Section 3).

2.3 Stream Formats

Session endpoints can only be connected if they generate and consume data streams of corresponding types. When a *connect* operation is executed a type check should be performed.

Digital audio and video are encoded as sequences of bytes. Such sequences are commonly referred to as *ropes* [14]. Ropes usually have an internal structure resulting from the data they contain and the way in which this data was sampled. We refer to the periodically recurring units of a rope as *frames* and to the constituents of frames as *segments*. Corresponding segments of adjacent frames may form ropes themselves. For example, all audio segments of a movie rope constitute the movie's soundtrack rope. To stress their relation to the parent rope, "subropes" are sometimes also called *strands* [15].

Usually the content of a rope is not important to the stream management subsystem. However, some information contained in the data may be useful to achieve a certain QOS when the session is in operation. To obey the desired session delay, the time stamp of a data unit may be used to determine its deadline. To provide the user-requested reliability, a segment may indicate how important a particular data unit is (e.g., in a differentially encoded video stream such as MPEG key frames are more important than interpolated frames); in overload situations the least important data units can then be discarded first.

The *type* of a rope is determined by the frame rate and the frame format. Ropes of similar types may be grouped to *classes*. In principle, it should be possible to translate a rope of one type into another type of the same class, e.g., CD-DA encoded audio may be transformed into DAT format. In practice, the translation algorithm may be arbitrarily complex and require special hardware. Classes can form hierarchies according to various criteria. For audio, they could be constructed according to content ("voice" and "music") or presentation ("stereo" and "mono").

Frames can be of fixed or variable size. A fixed size is common for audio and video which is not compressed at all or not compressed using content information. Positional information that is available beforehand, i.e., without looking at the frame, can be used to identify the different segments of a fixed-size frame. Compressing multimedia data exploiting content information, e.g., through Huffman coding or intra-value compression (as it occurs in storing differences between adjacent video samples), usually leads to frames of different sizes.[3] These frames need to contain a specification of where to find their different segments (e.g., as offset or length information).

While referring to multimedia data as "ropes" when concentrating on their static aspects such as the encoding, we prefer to talk about "streams" when multimedia data dynamically flows through a system. Apart from this, both terms are interchangeable.

2.4 Stream Weaving and Unraveling

A rope R may consist of two strands S1 and S2. We find it useful to connect a session endpoint yielding R with two endpoints consuming S1 and S2. To accomplish this, the software connecting endpoints must contain functions to *weave* and *unravel* ropes. Reasons for the inclusion of these functions are the following:

- Hardware can be more flexible if it processes simple ropes, preferably containing only one medium. This permits to improve capture and delivery algorithms for different media independently. On the other hand, an interleaved storage and transport of strands in a common rope format facilitates multimedia synchronization. Functions to weave and unravel ropes close the gap between both requirements.
- In today's multimedia systems, a large variety of different rope formats such as DVI-AVSS and MPEG are used. Even in the future, different standards for a single medium will be used depending on quality requirements: Not all applications require HiFi-quality audio, for many of them, voice quality is sufficient. The use of different rope formats may eventually require rope translation. For efficiency reasons, this translation should be accomplished as a sum of strand translations. Weave and unravel functions are a prerequisite for this process.
- If no special hardware for audio and video processing is available – as in many of today's experimental multimedia systems – processing has to be accomplished in software. For this software, weave and unravel functions constitute basic building blocks.[4]

To weave and unravel ropes requires knowledge about the rope format. For weaving, a *rope assembler* is needed; unraveling requires a *rope parser*. Assemblers and parsers can be considered as stream handlers which are inserted automatically into the connection graph. For ropes with fixed-size frames, generic assemblers and parsers can be used which use positional information. In case of ropes with variable-size frames, format-specific assemblers and parsers need to be provided. They have to "understand" rope content and structure.

The programmer identifies different rope segments by means of *segment identifications*. Names of different segments do not need to denote disjunct stream portions: The *video* segment can include the *time stamp* segment. Segment identifications can be used to connect endpoints. For weaving, all segments of the outgoing rope have to occur exactly once in the incoming ropes. For unraveling, not all segments need to be consumed: If a handle yields a movie rope containing an audio and a video strand it can be connected to an audio output only; the video is then discarded.

A special form of unraveling is *stream duplication*. Here, the segments which the next sessions receive are not disjunct; the same segment may be received by more than one input endpoint. A typical application for stream duplication occurs in a multimedia communication system which forwards incoming data to a number of destinations across different links.

[3] The use of color lookup tables for data compression, while exploiting content information, still results in fixed-size frames.

[4] Interestingly, the way in which many prototype systems operate today, i.e., without hardware compression, coincides with the long-term goal of multimedia systems: flexible handling of media in software. Dedicated hardware seems to be a temporary phenomenon to achieve reasonable quality within the next decade of scare resources [16].

Even if strands are transferred separately between endpoints, they cannot be treated independently if they have either a common origin or a common destination. We call this a *stream group*. Streams within a stream group cannot be started or stopped independently. Either these operations are disabled (e.g., in a video broadcast where none of the participants may pause) or an operation applied to one stream of the group applies to all of them.

2.5 Application Involvement

Some applications have the need to correlate discrete data such as text and graphics with continuous streams or have to post-process multimedia data (e.g., to display the time stamps of a video stream like a VCR). These applications need to obtain segments of multimedia at the stream handler interface.

Applications can be provided with multimedia data by means of a *grab* function which identifies the segments the application wants to receive. These segments are then copied to the application as if stream duplication took place. Unlike to data transfer in the RTE, no delivery time and minimum throughput can be guaranteed for this operation, i.e., data units loose their temporal properties once they enter the NRTE.

If an application needs to generate or transform multimedia data in a time-critical fashion, it has to provide a stream handler which is included in the RTE. The stream management subsystem provides functions for dynamically linking these stream handlers.

2.6 Synchronization

Stream synchronization is one of the functions which need to be provided by the stream management subsystem. Synchronization is performed by delaying the execution of a thread taking data through a session. When buffers are used, synchronization can be achieved by delaying the receive operation on the buffer. Synchronization is specified on a connection basis. Any connection may be synchronized, but usually synchronization will be used only for connections to sinks; only here the synchronization is apparent to the human user.

Synchronization can be expressed using the notions of "clocks" [9] or "logical time systems" [15]. These abstractions serve as a reference system to determine time points at which the processing of data units shall commence. For regular streams, the rates can be used to relate data units to synchronization points. Sequence numbers would accomplish the same. Time stamps are more versatile means for synchronization as they can also be used for irregular traffic. Yet their use implies stream interpretation.

3.0 A Multimedia Transport Service Interface

Any multimedia I/O facility needs to conform to the system architecture outlined in the previous section. Therefore, a multimedia transport system will be built as a stream handler on top of a multimedia network adapter. Its stream handler functions constitute a multimedia TSI.

3.1 General Considerations

At a traditional TSI, two kinds of data appear: control information for connection management and user data. According to the architecture presented in the previous section, user data is only transferred within the RTE. Hence, the usual *send* and *receive* functions are not available if the transport system is realized as a stream handler. A *grab* function as described in Section 2.5 is available.

Some transport connection management functions become handler-specific control operations: Obtaining a disconnect indication is just as specific to a network as a *zoom* operation is for a camera. However, TSIs differ from other interfaces such as the file system interface in that no client/server relationship can, in principle, be identified for its usage: "Client calls" cross the interface in both directions. There are mainly three models for implementing these calls: messages, downcalls, and upcalls.

Some traditional transport functions can be mapped onto the stream-handler independent control operations. In this context, it is important to distinguish between transport and session endpoint connections.[5] An *open* call for the transport system connects to a destination address, the *connect* call establishes a link to another local endpoint. The *close* and *disconnect* calls work accordingly.

3.2 A Multimedia Version of XTI

To illustrate the implications of the presented architecture, we choose the X/Open Transport Interface (XTI) [17] as an example. We only deal with the connection-oriented mode of XTI because we believe that connection-less service is useless for multimedia communication and its QOS requirements.

3.2.1 Initialization

In XTI, the *t_open* function is used to establish a local transport endpoint. Using the stream handler

[5] In the stream handler sense, of course, not in the sense of the OSI Reference Model.

model, this function is equivalent to a *create* operation.

One parameter of the *t_open* call chooses the transport protocol; this would be the place to select a new multimedia protocol. Note that just as defined for *create, t_open* does not yet determine a QOS for the connection. However, this may lead to problems if different qualities of service are to be provided by different protocols.

In the XTI initialization, a separate *t_bind* call is used to associate a transport address with the endpoint. In the stream handler interface, only one single *create* call is believed to suffice. There would be one transport address directing all incoming multimedia data to the stream management subsystem. This system then takes care of properly directing the data.

The *t_unbind* and *t_close* functions remove a transport endpoint and its binding to a transport address. In combination, they implement the *dispose* operation for the transport stream handler.

3.2.2 Transport Connection Establishment

Once a transport session is created, the *local* connection to other sessions can be made. After this, the session is opened – which can be mapped onto the connection establishment functions of XTI. The opening of a transport session leads to the establishment of a connection to a remote stream handler. It combines the local graphs of stream handler connections to one single global graph.

For a newly created transport session, there are two possible *open* functions: Either an outgoing connection is initiated through the stream handler or an incoming connection is expected. These are two flavors of the *open* call: In XTI, their equivalents are the *t_connect* and *t_listen* functions.

QOS negotiation is part of the *open*. XTI provides the facilities to negotiate options during the connection setup. These options can be used for QOS negotiation. In a *t_connect* call, initial requirements are first delivered to the transport system. In the blocking variant, *t_connect* returns with the finally negotiated QOS. In the asynchronous case, *t_rcvconnect* is used to obtain the negotiated QOS parameters.

For a multimedia application, end-to-end QOS parameters are usually more important than QOS parameters of a single stream handler. For example, when a multimedia file is retrieved from a remote disk not only the delay occurring in the network, but also the delay in the file system and the decompression units contribute to the end-to-end delay. To negotiate QOS in a single path (see Section 2.2), the QOS parameters provided at the multimedia TSI will not just determine the quality of the transport service, but will contain a QOS decision about the entire remaining stream route.

The receiver obtains the peer's request for QOS by means of the *t_listen* call. The transport provider will already have determined or refined some of the achievable QOS parameters such as accumulated delay or tolerable throughput. The receiver uses these parameters for local QOS determination and returns the parameters he has chosen with the *t_accept* call. A *t_snddis* serves as an indication for connection rejection. Unlike *t_accept*, *t_snddis* does not permit the inclusion of QOS parameters as options. To let session acceptance and rejection appear similar at the TSI, one may consider providing a separate *t_reject* function which informs the peer about the QOS that would have been achievable.

The usual disconnect functions are available. They are used as variants of the *close* function. A provider-initiated disconnect resembles a situation where an underlying resource can no longer provide the guaranteed QOS. Depending on the reservation model being used, this situation can also occur for non-transport sessions (cf. [18]).

3.2.3 Data Transfer

XTI is a typical downcall interface: The transport user calls a *t_look* function to obtain transport indications. This mechanism is used for both the provision of control information (e.g., an incoming connection request) and the provision of data. In a multimedia TSI, control information and data are passed differently. The data functions are not visible at the interface; only a *grab* function as described in the previous section is available.

The system has to have control about any data forwarding in order to achieve timeliness. The downcall mechanism of the *t_rcv* operation seems to be particularly unsuitable for this purpose. The threads to implement sessions either require upcalls or a message-oriented interface. In addition, downcalls make no sense if there is only a single function left for indication.

4.0 Conclusion

We have shown that while multimedia data requires special treatment in a computer system because of its real-time requirements, a typical high-level I/O interface fits nicely into the software architecture of future multimedia systems. We have used XTI as an example in this paper, but other interfaces such as that of a file system can easily be incorporated in the architecture. It should also be noted that our considerations about a multimedia TSI can also be applied to higher levels of communication if they exist.

When discussing the stream management subsystem, it is important to keep in mind that its services rely on the availability of the underlying buffer and resource management subsystems. For a more complete view of all multimedia support functions the reader is referred to [5, 6].

References

[1] First International Workshop on Network and Operating System Support for Digital Audio and Video, International Computer Science Institute, TR 90-062, Berkeley, Nov. 1991.

[2] Second International Workshop on Network and Operating System Support for Digital Audio and Video, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, to appear.

[3] D. Hehmann, R.G. Herrtwich, R. Steinmetz: Creating HeiTS: Objectives of the Heidelberg High-Speed Transport System, F&E-Projektberichte, GI-Jahrestagung 1991, Darmstadt, Oct. 1991.

[4] D. Hehmann, R.G. Herrtwich, W. Schulz, T. Schütt, R. Steinmetz: Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System, Second International Workshop on Network and Operating System Support for Digital Audio and Video, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, to appear.

[5] B. McKellar, W. Schulz, K. Reinhardt: The Heidelberg High-Speed Transport System – Buffer Management Subsystem, IBM European Networking Center, Working Specification, Oct. 1991.

[6] C. Vogt, R.G. Herrtwich: The Heidelberg High-Speed Transport System – Resource Management Subsystem, IBM European Networking Center, Working Specification, July 1991.

[7] D.P. Anderson, R.G. Herrtwich: Internet Communication with End-to-End Performance Guarantees, Telekommunikation und multimediale Anwendungen der Informatik (GI-Jahrestagung 1991), Informatik-Fachbericht 293, Springer-Verlag, Heidelberg, Oct. 1991.

[8] R.G. Herrtwich, R. Nagarajan, C. Vogt: Guaranteed Performance Multimedia Communication Using ST-II Over Token Ring, IBM European Networking Center, submitted for conference publication.

[9] R.G. Herrtwich: Time Capsules – An Abstraction for Access to Continuous-Media Data, Journal of Real-Time Systems 3, 355-376, Dec. 1991.

[10] D. Hehmann: Personal communication.

[11] C.L. Liu, J.W. Layland: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of the ACM 20, 1, 47-61, Jan. 1973.

[12] C.W. Mercer, H. Tokuda: Priority Consistency in Protocol Architectures, Second International Workshop on Network and Operating System Support for Digital Audio and Video, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, to appear.

[13] R. Steinmetz, H. Schmutz, B. Schöner, M. Wasmund: Generic Support for Distributed Multimedia Applications, IEEE ICC 1990, Atlanta, Apr. 1990.

[14] D. Terry, D.C. Swinehart: Managing Stored Voice in the Etherphone System, ACM Transactions on Computer Systems 6, 1, 3-27, Feb. 1988.

[15] D.P. Anderson, R. Govindan, G. Homsy: Abstractions for Continuous Media in a Network Window System, International Conference on Multimedia Information Systems, Singapore, Jan. 1991.

[16] D.P. Anderson, S. Tzou, R. Wahbe, R. Govindan, M. Andrews: Support for Continuous Media in the DASH System, 10th International Conference on Distributed Computing Systems, Paris, May 1990.

[17] X/Open Company Ltd.: X/Open Portability Guide – Networking Services, Prentice Hall, Englewood Cliffs, 1988.

[18] R.G. Herrtwich: The Role of Performance, Scheduling, and Resource Reservation in Multimedia Systems, Operating Systems for the 90s and Beyond, Lecture Notes of Computer Science, Springer-Verlag, Heidelberg, to appear.