[HHS+91]

HHS+91

Dietmar Hehmann, Ralf Guido Herrtwich, Werner Schulz, Thomas Schütt, Ralf Steinmetz; **HeiTS - Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System;** 2nd International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, 18.-19. November 1991, Tagungsband erschienen als "Lecture Notes in Computer Science", Springer Verlag, 1992.

Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System

Dietmar Hehmann
Ralf Guido Herrtwich
Werner Schulz
Thomas Schütt
Ralf Steinmetz

IBM European Networking Center
Tiergartenstr. 8
D-6900 Heidelberg 1

1.0 Introduction

The **Heidelberg High-Speed Transport System** (HeiTS) is a new-generation end-to-e communication system currently under development at the IBM European Netwo ing Center (ENC) in Heidelberg. HeiTS is aimed at a heterogeneous environme comprising several computers with different operating systems and a variety o underlying local, metropolitan, and wide-area networks. It incorporates both e system and gateway communication functions.

In new high-speed networks, an integration of different traffic types can be observe Whereas, e.g., Ethernet was designed for data traffic only, FDDI supports data-ty traffic (asynchronous) as well as voice and video traffic (synchronous), albeit i rudimentary form. Such an integration in the physical network should lead to integrated communication system as a whole. HeiTS is designed to support fast da traffic and multimedia communication, in particular the transfer of digital audio a video. The HeiTS prototype is conceived to be a generic basis for high-speed da transport applications such as CAD file transfer, and multimedia communicati applications such as video distribution.

For many years, networks were the bottleneck of data transmission. Processi equipment in the end-systems and gateways was faster than the transmission lines that bandwidth usage had to be optimized over processing. With the upcomi high-speed networks, this paradigm is changing: Shortage of resources is now in t nodes. To achieve high performance, both architecture and implementation of HeiTS are oriented to optimize processing over bandwidth use.

HeiTS is primarily directed to two platforms within IBM's Small Systems line: t PS/2 under OS/2 and the RISC System/6000 under AIX. This imposes the burde to interface HeiTS with two disjoint operating systems, but the common Microcha nel bus architecture of both machines offers the opportunity to use identical comm nication network adapters on both systems. The primary networks to be operate by HeiTS are Token Ring, FDDI, and Broadband-ISDN. For B-ISDN, a first ve sion will attach to an STM network, later versions will interface with an ATM ne work.

This paper discusses our implementation decisions for HeiTS. Section 2 introduces the overall multimedia system architecture into which HeiTS will be integrated. Section 3 elaborates on the support functions that serve as building blocks for the construction of HeiTS. Section 4, finally, discusses the actual communication functions that HeiTS provides. For the objectives of the system please refer to an earlier paper [5].

2.0 Overall Multimedia System Architecture

HeiTS is just one module within a general platform for multimedia applications. It, therefore, has to fit into the overall architecture of such a platform. This section discusses the external constraints on the implementation of HeiTS which result from the need for integrating HeiTS with a surrounding system.

2.1 Stream Handlers

Multimedia data usually enters the computer through an input device and leaves it through an output device (where storage can serve as an I/O device in both cases). It is less common that the data is generated by the computer itself, i.e., calculated or interpreted by the CPU. Nevertheless, this case occurs in simulation and control applications; a multimedia architecture needs to take it into account, too.

An entity generating or consuming a raw stream of continuous multimedia data is commonly called a *stream handler*. The term "raw" shall indicate that no stream-handler-specific data is contained in the stream. For example, in a transport system (which would be a typical stream handler), the data must not contain any protocol headers or trailers; they have no meaning to other stream handlers, e.g., those responsible for video output. We make no restriction on the encoding of a raw stream. In particular, a stream can be compressed or uncompressed. Any stream should be typed to prevent it from being directed to the wrong stream handler.

Stream handler functions are often distributed among the main CPU and on-board processors. In case of video decompression adapters most of the stream handler software executes on the on-board DSP of the adapter. Additional software completes the stream handler to access the board from an application. Whenever the adapter does not deliver raw data, but some additional information, the stream handler software that executes on the main CPU becomes more complex. Multimedia file or transport systems are examples of such stream handlers. In applications where multimedia data is generated or consumed by the computer, a stream handler may run on the CPU only.

In a modern system which follows the microkernel approach, one will arrive at the following three-level code structure for a stream handler implementation:

- *Hardware portions* of stream handlers are executed on an adapter board. In many cases, one will not be able to change these stream handler portions. However, some modern hardware stream handlers are microprogrammable.
- *Device drivers* constitute the stream handler portion executed inside the operating system kernel to interface to the hardware adapter.
- *Software portions* of stream handlers execute in user space, on top of the operating system kernel. From a software engineering viewpoint, the majority of the stream handler code should belong to this portion. Stream handlers which do

not require special hardware support (e.g., simple filter functions) can be implemented in software only.

This layered stream handler structure is continued within each stream handler portion. In particular, the stream handler implementation of a transport system will contain the traditional communication layers.

2.2 Threads

Stream handlers need to obey the inherent real-time requirements of audiovisual data: They have to deliver their output before a certain deadline to make it available in time for its presentation or consumption. In addition, they may have to reduce jitter between the delivery of adjacent output items or synchronize the presentation of their data items with those of other stream handlers.

The following implementation structure makes it easy to take these timing criteria into account: All stream handler software portions are encoded as functions of a single task. This task assumes the role of an *audio/video server* which – somewhat similar to the X server – provides a common input/output environment for time-critical multimedia data. The functions of the AV server are executed by *threads* which escort a single piece of multimedia data from input to output. They wait to obtain the data from the device driver, then execute the layered functions of the input stream handler in an upcall fashion [2], execute the functions of the output stream handler in a downcall fashion and finally submit the data to the output device through the corresponding device driver functions.

Threads, much better than messages themselves, can take the timing requirements of multimedia data into account. Each thread can be scheduled according to the urgency of the data item it handles using real-time scheduling techniques. It also can be synchronized with the execution of other threads through well-known process synchronization functions. An appropriate synchronization point is the switch from the upcall to the downcall segment. A thread can also be paced at this point by delaying its execution for some time.

2.3 Stream Management

A multimedia application runs on top of the AV server outside of the real-time environment. Multimedia data usually does not pass through the application; the application merely manages the flow of data in the AV server. To manage the data flow, we distinguish between *device control operations* that determine the content of a multimedia stream and *stream control* operations that determine its direction.

Device control operations depend on the individual I/O device. Devices may be grouped into classes and their device control operations may be derived generically as suggested in [10]. For storage devices, typical control operations include *fast_forward*, *reverse* and *seek*. Other operations are *zoom* for cameras and *volume* for speakers.

Stream control operations are the same for all stream handlers. They include

- *open/close* (applied to individual stream handlers, yielding *stream handles*),
- *connect/disconnect* (applied to pairs of stream handles, yielding *stream identifications*), and

- *start/stop* (applied to stream identifications).

The *open* function is a hybrid between a device control and a stream control operation. It usually requires information specific to the stream handler that also determines stream content. Except for stream handlers that cannot be multiplexed (such as those for microphones and speakers), opening the stream handler is not enough: In a file system, the file to be accessed needs to be known. In a transport system, the address of the communication partner is required. In a video display, the area where to display the data on the screen is needed. This information is provided by handler-specific parameters of the *open* call.

The *connect* function generates a thread to escort data from the input to the output stream handler. When the connection is established, system resources are allocated to ensure that the thread can perform its function according to the application's requirements (on time, with a certain reliability, etc.). In distributed applications, such resource reservation has to be made from end to end, including the network.

In addition to the above functions, an application can also specify that a connection (= thread) shall be synchronized. In this case, the above-mentioned synchronization mechanism is enabled and threads are potentially delayed.

The following picture shows the overall architecture of HeiTS:

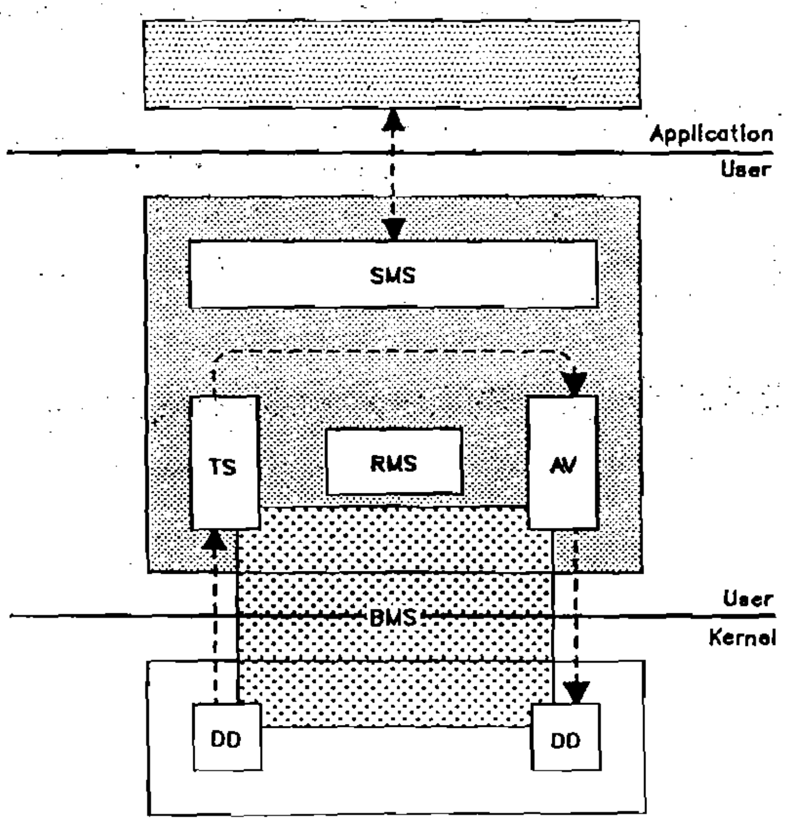

Figure 1. HeiTS: Architecture

The devices drivers (DD), the transport system (TS) and the audio/video handler (AV) are examples of stream handlers. Data flows from a network adapter through the corresponding device driver to the transport system, where the communication functions are provided (see 4.0), passed over to the audio/video handler and written to the adapter using the AV-device driver. The buffer management system (BMS) enables the handling of the data between the different stream handlers without copied them (see 3.1). The resource management system (RMS) allocates and manages the resources (see 3.2). The stream management system (SMS) provides the interface to the application.

3.0 Support Functions

In protocol implementations, the protocol machine contributes only a small fraction to the overall processing time. Most computation power goes into support functions such as data administration, communication between modules (e.g., processes), etc. This is even more true for light-weight protocols with their streamlined protocol machines.

In HeiTS our goal is to handle data in real-time. First of all, this means that some delay bounds for the data handling can be guaranteed. As delay bounds for audiovisual data are tight, this automatically translates into fast data handling. To handle data efficiently, a sophisticated *buffer management* is needed. To schedule the resources for real-time data handling, we need a *resource management* which reserves the resources in HeiTS and guarantees the commitments made.

3.1 Buffer Management

Conceptually, data always flows from one stream handler into another. However, if a significant portion of the stream handler is realized in software and makes use of the CPU, this flow of the data should not imply costly data copying, in particular copying from kernel to user space and back.

The buffer management system (BMS) enables the transfer of the data "below" the stream handlers to achieve higher performance. In this case, special device capabilities such as direct adapter-to-adapter transfer can be utilized and the BMS hides differences between buffers on different adapters and in main memory.

The BMS not only avoids copying while data is flowing between stream handlers, it also provides features needed for efficient protocol implementations such as chaining of buffer fragments (headers from different layers, data, possibly trailers) and locking (e.g., to keep buffers for retransmissions).

A BMS buffer consists of one or more blocks of memory (called fragments) which are linked together. The information describing the buffer is contained in a buffer descriptor, so no buffer management information has to be stored in the fragments.

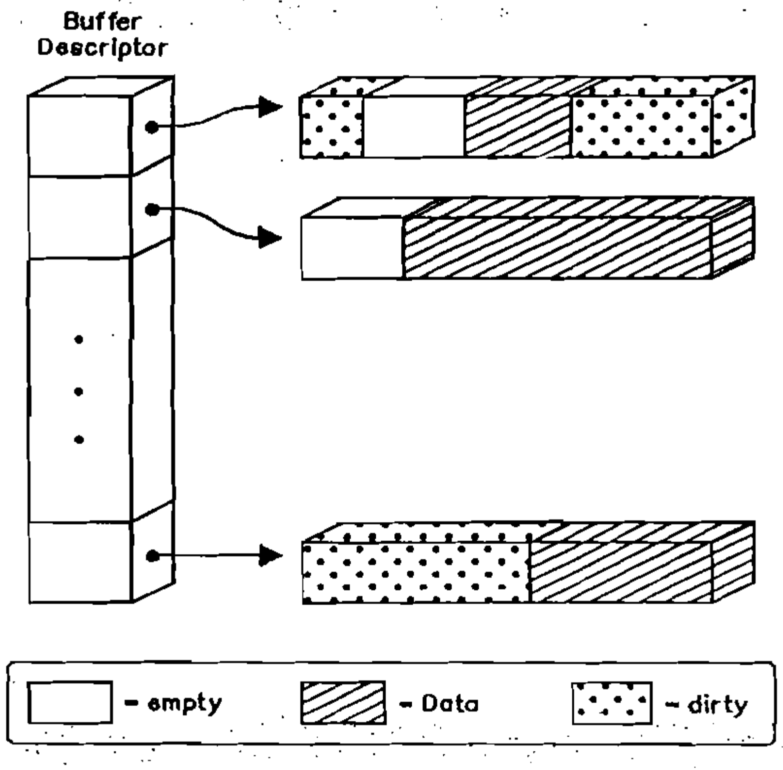

Figure 2. BMS: Buffers and Fragments

A fragment consists of three parts:

- The *data area* is filled with information.
- The *empty area* is free and can be used to store information (e.g., to add a protocol header).
- The *dirty area* cannot be used (it can contain adapter specific information).

The pointers to the different areas are stored in the buffer descriptor. Space in a fragment is allocated from the back to the front, so each layer can add its header. If there is no empty space left in the fragment, a new fragment is allocated and is linked to the current buffer. So it is not necessary to copy the buffer.

A side effect of this scheme is, that segmenting and recombining of data units is possible without copying the data. To segment a data unit into, e.g., two pieces, the BMS allocates a second buffer descriptor which points to the same fragment(s) and only changes the pointers to the different areas accordingly.

Normally the last stream handler handling the buffer gives the buffer back to the BMS after the data is copied to the external device and the buffer is freed. Under some circumstances a stream handler may want to keep the data for later use, e.g., the transport layer may keep a buffer for retransmissions if it has to provide a reliable service to its user. In this case the respective stream handler can tell the BMS to *lock* the buffer. When a locked buffer is returned to the BMS, it is not actually freed until the lock is removed.

3.2 Resource Management

The resource management system (RMS) allocates resources for connections to guarantee a certain *throughput, delay* and *reliability* [6]. The workload model used in HeiTS is the *Linear Bounded Arrival Process (LBAP)*, which was introduced by Cruz [4] and employed, for example, in SRP [1]. Whenever a new connection is established, the RMS makes sure that this connection does not violate performance guarantees already promised to other connections. There are two types of connections: best effort and guaranteed.

The RMS consists of submodules for each resource (e.g., local resources like CPU processing capacity and network resources like bandwidth) to perform schedulability testing, reservation, and resource scheduling. The RMS also reserves buffer space. The buffers needed for a connection are allocated statically from a buffer pool. The amount of buffer space to be reserved is based on the throughput and the burst size.

Let us discuss the reservation and scheduling of the CPU as an example for the resource management techniques employed. For any new connection, a schedulability test is performed: Based on the maximum message rate of a connection and the processing time needed per message it is calculated whether the acceptance of this new connection could violate guarantees for other connections. If this is the case the new connection is rejected. The values calculated for accepted connections are stored in a local database. This information is used by the RMS for further schedulability tests and by the scheduler for scheduling the thread processing the message.

For CPU scheduling we are currently using three priority classes:

1. critical threads,
2. critical threads that have used up their processing time, as specified by their workload specification, but require further processing (this is based on our optimistic assumption, that this message can make up for the lost time in later resources and is especially useful for best effort connections), and
3. threads that are no stream handlers (the normal system threads).

Currently we investigate to use a fourth class for workahead messages, but it is not clear if the potentially better CPU utilization is worth the scheduling effort (which consumes CPU capacity itself).

The scheduling within the different classes is currently based on a modified rate-monotonic scheme, where the priority of a thread is based on the message rate for the respective connection. For incoming messages the urgency is calculated and depending on the outcome the message is passed to the thread handling the connection directly or it is hold back by the scheduler.

In the future we plan to also use deadline scheduling where the priority of a thread is calculated based on the urgency of an arriving message.

4.0 Communication Functions

Using the support environment introduced in the previous chapters, HeiTS realizes the function of the lower 4 layers of the OSI reference model as a stream handler providing endsystem-to-endsystem transfer of multimedia data items.

4.1 Design Decisions

Based on the application requirements either a reliable or an unreliable communication path between a sender and a single or multiple recipients can be established by HeiTS. QOS parameters are used to specify such requirements. Formally, the services provided are based on the ISO transport service standard document [7]; however suitable enhancements had to be defined to cover typical multimedia requirements. Additionally, the use of certain ISO defined optional facilities had to be restricted to ensure isochronity requirements could be met.

Specific design decisions have been made in the following areas:

- *Calling conventions:* Downcalls are used for outbound communication (i.e., requests and responses in the OSI terminology), upcalls for incoming indications and confirmations.

  This architectural decision ensures in particular that incoming data can not only be offered for processing to the application – as is standard practice in today's data-oriented communication systems –, but that any required processing can directly be initiated by HeiTS at the correct time. Side effects of this design decision include reduced elasticity buffer requirements and that immediate connection-specific processing can be done in the user-provided indication and confirmation routines.

  Entry points into these user provided functions are passed to the transport subsystem at the latest possible occasion.

- *Multicast:* Multicast is supported by the network layer where the topology of the network is known. Two forms of multicast are distinguished: In traditional "sender-initiated" multicast, the sender enumerates its communication partners. In "receiver-initiated" multicast, a receiver may join an existing communication (probably without even informing the sender about its presence).

- *Multiplexing:* Multiplexing is not supported for time-critical traffic above the data link layer, i.e., a data link connection will always be mapped onto a single transport connection. It is necessary to support multiplexing in the data link layer since some networks support only one physical connection. Its exclusion for network and transport layer allows for easier identification of the receiving process for incoming data.

- *Splitting:* Splitting is not supported, i.e., a single upper-layer connection does not use more than one lower-layer connection. In particular, one transport connection never sends data over two or more network adaptors. Splitting was once used to let a fast processor output data to several slow networks, increasing the overall throughput of a connection. In an environment, where networks become faster than processors, splitting becomes obsolete.

- *Segmentation:* Segmentation should be avoided, but is supported by HeiTS. Segmentation to and reassembly of very small data units (such as ATM cells), however, shall be accomplished in hardware – HeiTS is not optimized for this function.

- *Flow control:* Flow control consists of end-to-end flow control to prevent the receiver from being flooded with data and access control to prevent the network from being overloaded. For time critical unreliable traffic, HeiTS applies a rate-based control scheme for connections and enforces the rate of connections through leaky bucket algorithms. For conventional reliable data communication the standard techniques are used.

4.2 Implementation Structure

Internally the communication subsystem is structured into 3 layers:

- *Transport Sublayer:* The transport subsystem provides reliable and unreliable end-to-end communication services enhanced by provisions for multimedia data transfer. The use of these provisions results in an isochronous data delivery whenever a respective quality of service was negotiated. As a starting point, an extended ISO transport service is considered. A modified ISO transport protocol class 1 [8] has been implemented. Other protocols (e.g., XTP) are under consideration.

- *Network Sublayer:* The focus of the network layer work is in the areas of multicast support and LAN/WAN internetworking. Two different protocols are currently being implemented for experimentation: As a result of previous ISO work, a modified X.25 version is used for unicast experiments in gateway scenarios. The Internet protocol ST-II [12] is used to experiment with multicast communication over guaranteed-performance channels.

- *Connectivity Subsystem:* The connectivity subsystem provides a data link service interface to HeiTS. On most of the already available adapters for high-speed networks a data link protocol is implemented. The connectivity subsystem hides the different interfaces of the drivers for the various network adapters. Network adaptors currently under consideration are 4 and 16 Mbit Token Ring, FDDI, and B-ISDN.

A stream handler interface is built on top of the HeiTS stack.

4.3 Sample Session

Let us discuss a "typical" example scenario from the transport system interface perspective. Assume the head of a small company wants to give his Monday morning speech (a monologue, of course) to his employees sitting in their offices with their multimedia workstations switched on and ready to listen to their boss. When the chief is ready to begin, we will see the following events at the transport service interface:

```
Chief                           Employees

                                ts_open_sap (MMSAP, ..., @ts_conn_ind, @ts_disc_ind, ...)
ts_open_sap (MMSAP, ...)
ts_conn_req (MMSAP, Employeelist,
           {error_indication, 1.4 Mbps, 25 SDUs per second, 250 msec constant delay},
           @ts_conn_conf, ...)
                                >ts_conn_ind (...)
                                ts_conn_rsp (..., @ts_data_ind, ...)
>ts_conn_conf (...)
ts_data_req (..., @first_picture_data)
                                >ts_data_ind (...)
.                               .
.                               .
.                               .
ts_disc_req (...)
                                >ts_disc_ind (...)
ts_close_sap (MMSAP)
```

Figure 3. Unidirectional Live Distribution of Compressed TV

In this example the symbol " > " is used to identify upcalls. "@" stands for "address of".

The example illustrates some of the key choices made for the HeiTS design: First, the concept of specialized service access points is used to distinguish multimedia and regular data traffic. Second, a set of QOS parameters is used to specify the application requirements. In this case typical values for the distribution of compressed video are given. In particular, error indication but no correction is specified. This enables the output stream handler to substitute, e.g., a corrupted video frame by either a previous full frame or a zero delta frame which will prevent the error from being visible. However, strong isochronity for the individual pictures and voice sample is required. Third, only addresses are passed at the procedure interfaces whenever data need to be handed over an interface.

5.0 Summary

HeiTS is designed to handle high-speed data applications as well as multimedia data applications within IBM's Small System line (PS/2 under OS/2 and the RISC System/6000 under AIX). The main emphasis in this paper was put on the multimedia aspect. In order to meet the real-time requirements of audiovisual data streams HeiTS uses threads to handle this streams. These threads can be scheduled dependent on their real-time requirements. In order to allow this kind of scheduling the Resource Management System has been implemented in HeiTS. It allows best effort and guaranteed connections, and it supplies the scheduler with the necessary information for real-time scheduling.

Another aspect within HeiTS is to minimize the overhead of data handling. For this reason, a Buffer Management System was defined that allows efficient data handling. This includes segmenting and recombining of data units, chaining and locking of buffers, and other features. With this buffer management all unnecessary data movements can be avoided.

With all these supporting functions defined HeiTS is an implementation of the lower four layers of the OSI Reference Model. It allows multicast on the network layer. multiplexing up to the data link layer, segmentation, and ent-to-end flow control.

Currently a modified ISO transport class l has been implemented. But other protocols like XTP are under consideration for future implementations. There is also still some open issues in the resource management, e.g., the different threads are scheduled based on a rate-monotonic scheme, where the thread's priority is based on the message rate for the connection. This will be replaced by a deadline scheduling, where the priority of a thread is adjusted to the urgency of an arriving message, and is not based on a QOS input parameter.

References

[1] *D. P. Anderson, R. G. Herrtwich, C. Schaefer:* **SRP: A Resource Reservation Protocol for Guaranteed-Performance in the Internet.** ICSI, Berkeley, TR-90-006, Feb. 1990.

[2] *D. D. Clark:* **The Structuring of Systems Using Upcalls.** 10th ACM SIGOPS Symposium on Operating System Priciples, Orcas Island, Washington, Dec. 1985, pp. 171-180.

[3] *D. D. Clark, D. D. Tennenhouse:* **Architectural Considerations for a New Generation of Protocols** SIGCOMM '90 Symposium "Communications Architectures and Protocols", Philadelphia, Pennsyslvania, Sep. 1990, pp. 200-208.

[4] *R. L. Cruz:* **A Calculus for Network Delay, Part I: Network Elements in Isolation.** IEEE Transactions on Information Theory, Vol. 37, No. 1, January 1991.

[5] *D. Hehmann, R.G. Herrtwich, R. Steinmetz:* **Creating HeiTS: Objectives of the Heidelberg High-Speed Transport System.** GI-Jahrestagung, Darmstadt, Oct. 1991.

[6] *R. G. Herrtwich, R Nagarajan, C. Vogt:* **Guaranteed-Performance Multimedia Communication Using ST-II Over Token Ring.** Submitted for publication.

[7] *International Standards Organisation:* **International Standard 8072, Information Processing Systems – Open Systems Interconnection – Transport Service Definition.** 1986.

[8] *International Standards Organisation:* **International Standard 8073, Information Processing Systems – Open Systems Interconnection – Connection-Oriented Transport Protocol specification,** 1986.

[9] *N. Luttenberger, R. v. Stieglitz:* **Performance Evaluation of a Communication Subsystem Prototype for Broadband ISDN.** IEEE Workshop on the Future Trends of Distributed Computing Systems, Cairo, Sep. 1990.

[10] *R. Steinmetz, R. Heite, J. Rückert, B. Schöner;* **Compound Multimedia Objects – Integration into Network and Operating Systems.** International Workshop on Network and Operating System Support for Digital Audio and Video, International Computer Science Institute (ICSI), Berkeley, Nov. 1990.

[11] *D. L. Tennenhouse:* **Layered Multiplexing Considered Harmful.** In: H. Rudin, R. Williamson (Eds.): Protocols for High-Speed Networks, Elsevier (North-Holland), 1989.

[12] *C. Topolcic (Ed.):* **Experimental Internet Stream Protocol, Version 2 (ST-II).** Internet Request for Comment 1190, Oct. 1990.