ACM Operating Systems Review, Vol. 28, No. 2, April 1994, pp. 51-64

The System Architecture of the Heidelberg Transport System

Lars C. Wolf
Ralf Guido Herrtwich

IBM European Networking Center
Distributed Multimedia Solutions
Vangerowstr. 18
D-69115 Heidelberg

{lwolf, rgh}@dhdibmip.bitnet

Abstract: HeiTS, the Heidelberg Transport System, is designed for communication of continuous-media data. The purpose of HeiTS is the exchange of digital audio and video with quality of service guarantees. The system environment of HeiTS has to coordinate real-time and non-real-time functions. Most parts of HeiTS run in user space; it uses some kernel-based mechanisms such as a real-time CPU scheduler and device drivers. On top of HeiTS exists a control system which allows for the construction of applications controlling continuous-media streams. This paper describes the architecture of HeiTS and the mechanisms of its system environment we have implemented on AIX and OS/2 platforms.

1 Introduction

The integration of continuous-media data streams such as digital audio and video into general-purpose computer systems imposes new requirements onto operating system mechanisms. Processing of such streams has to obey timing requirements which are uncommon for traditional workstation systems. Resource scheduling has to take into account timing aspects through the use of real-time techniques instead of following fairness policies only. Additionally, continuous media lead to high data rates. This high-volume data makes it impossible for applications to touch every single data item within the continuous-media stream.

HeiTS, the Heidelberg Transport System, provides the ability to exchange streams of continuous-media data with quality of service (QoS) guarantees – where applications can specify the requirements they have for the transport service. To provide these QoS guarantees, the protocols of HeiTS are embedded into an environment which provides real-time techniques and resource management [HerWol92].

The HeiTS implementation is arranged for fast protocol processing during the data exchange phase. This is reflected by the use of connection-oriented protocols (which are also important for resource management), the use of special support functions such as buffer management, and the use of an upcall structure throughout the system. Therefore, HeiTS differs in many ways from other communications systems. To explain these differences, this paper provides an architectural survey of HeiTS.

The outline of this paper is as follows: In the remainder of the introduction, a brief overview of the structure and characteristics of HeiTS is given. Then, the components, the environment and the transport system interface of HeiTS are explained. Section 3 discusses how messages are processed within the system. Finally, we explain how higher layers may be built on top of our system before we conclude with some remarks about status and future work.

1.1 Characteristics of HeiTS and its Environment

HeiTS has the following features not commonly found in other communication systems:

- Continuous-media exchange with QoS guarantees
- Upcall structure
- Resource management and real-time mechanisms

HeiTS transfers continuous-media data streams from one origin to one or multiple targets via multicast. HeiTS nodes negotiate QoS values by exchanging flow specifications to determine the resources required – delay, jitter, throughput and reliability [Fer90].

Data is processed within a single thread as far as possible – from the source to the sink of data, even for the exchange of data between HeiTS and its environment. Therefore, processing of messages is done with upcalls [Cla85] instead of using a thread per layer as in a server model [Svo89], minimizing the overhead of crossing thread boundaries.

The use of a resource management is necessary for QoS provision. The resource management is the central location for information about resource availability and schedulability of connections. It also performs the actual scheduling of the resources.

1.2 Implementation Level

HeiTS is currently implemented on two platforms[1]: AIX on IBM RISC System/6000, and OS/2 on IBM PS/2. Most of the code is implemented in user level. Only a few parts, e.g., the real-time CPU scheduler, are implemented inside the operating system kernel because they need kernel services and data structures. Although a user-level implementation is unusual for a communication system our reasons for this were:

- easier porting between AIX and OS/2,
- easier porting to other vendors' platforms,
- easier testing.

A user-level implementation is also in line with the current de-kernelization trend, i.e. with moving functions out of the operating system kernel. This is especially important for communication software; in [LefMcK89] it has been found that these modules make up a large part of the kernel code. We believe that our user-level implementation facilitates moving to a micro-kernel version of HeiTS.

It is often believed that kernel-level implementations have an inherent better performance than user-level code. Similar to results obtained for micro-kernel operating systems which are as fast as operating systems with a monolithic kernel (e.g., Mach in [GolDea90]), first measurements on the AIX version of HeiTS indicate that our user-level implementation does not

1. Implementations for A/UX on the Apple Macintosh and for Windows are under development.

lead to performance degradation. Our network layer protocol runs faster in user level than the IP implementation in the kernel [DelHer93b].

1.3 Support Modules

HeiTS uses several support mechanisms:

- buffer management system (BMS),
- resource management system (RMS), and
- stream management system (SMS).

Products such as MMPM/2 [IBM92] provide system components for stream management that can be used as an SMS for HeiTS. For the other support mechanisms, there are either no such subsystems available or we felt that these systems were not appropriate for our purpose, e.g., the well-known *mbufs* [LefMcK89] buffer management system is available inside the operating system kernel only and provides limited flexibility for buffer sizes. In particular, we have suggestions for BMS and RMS as HeiBMS (Heidelberg Buffer Management System) [KroMcK93] and HeiRAT (Heidelberg Resource Administration Technique) [VogHer93].

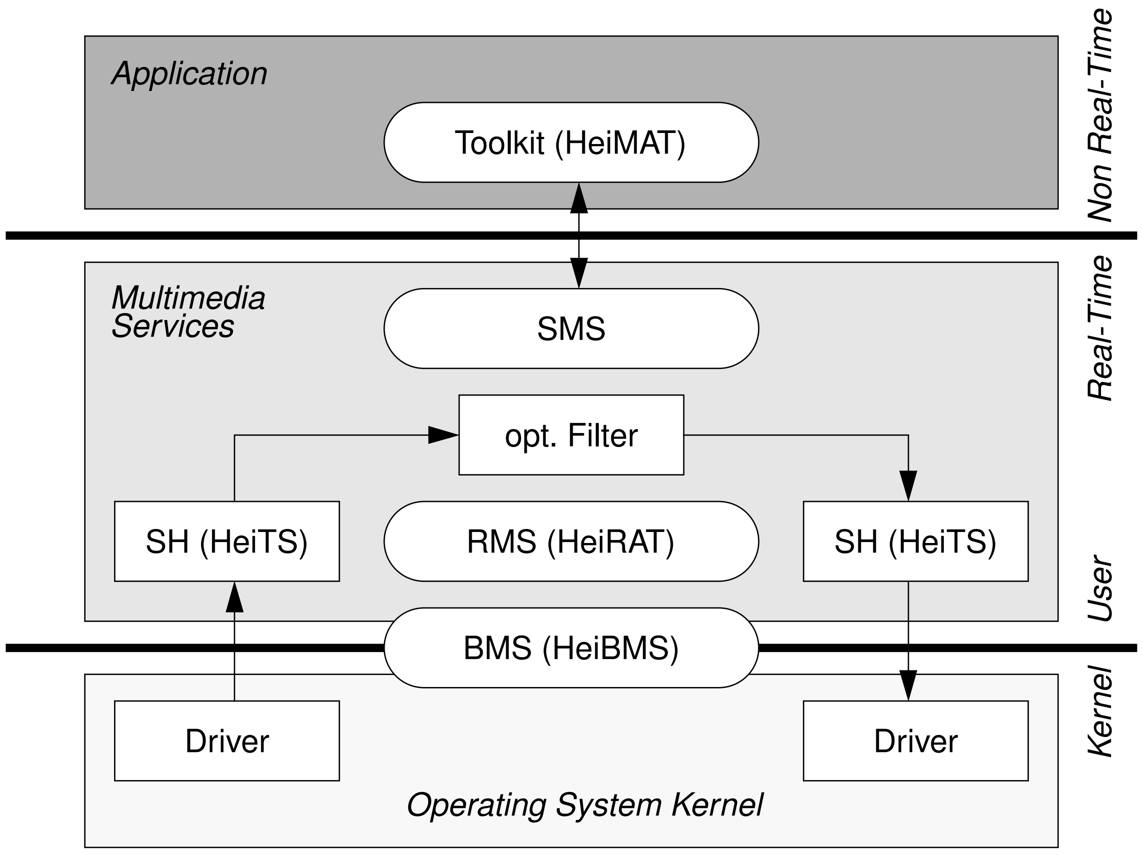

Figure 1: The Environment of HeiTS.

HeiBMS contains functions to operate on data blocks of varying sizes with a minimum number of copy operations and provides functions designed to support communication of continuous-media data; it may be used in HeiTS but on top of HeiTS (by an application) as well. HeiRAT provides resource calculation and scheduling functions to offer real-time processing.

On top of HeiTS, modules exists which allow the construction of applications controlling continuous-media streams. The SMS belongs to this upper part and provides mechanisms to arrange modules handling continuous-media data. The structure for the flow of data in the system is shown in Figure 1.

In the non-real-time environment, only functions which have no strict time requirements are executed, e.g., functions to establish a new connection. Inside the real-time environment, the processing is scheduled with regard to time criticalness, here the processing of continuous-media data streams is performed through modules called stream handlers (SH). Some SHs are associated with input/output devices like a display or network (i.e., HeiTS will be encapsulated into a network stream handler). Other SHs modify a continuous-media stream without I/O operation (filters). User applications have access to the real-time environment via the SMS only, specifying their requests for creation and combination of SHs inside the real-time environment.

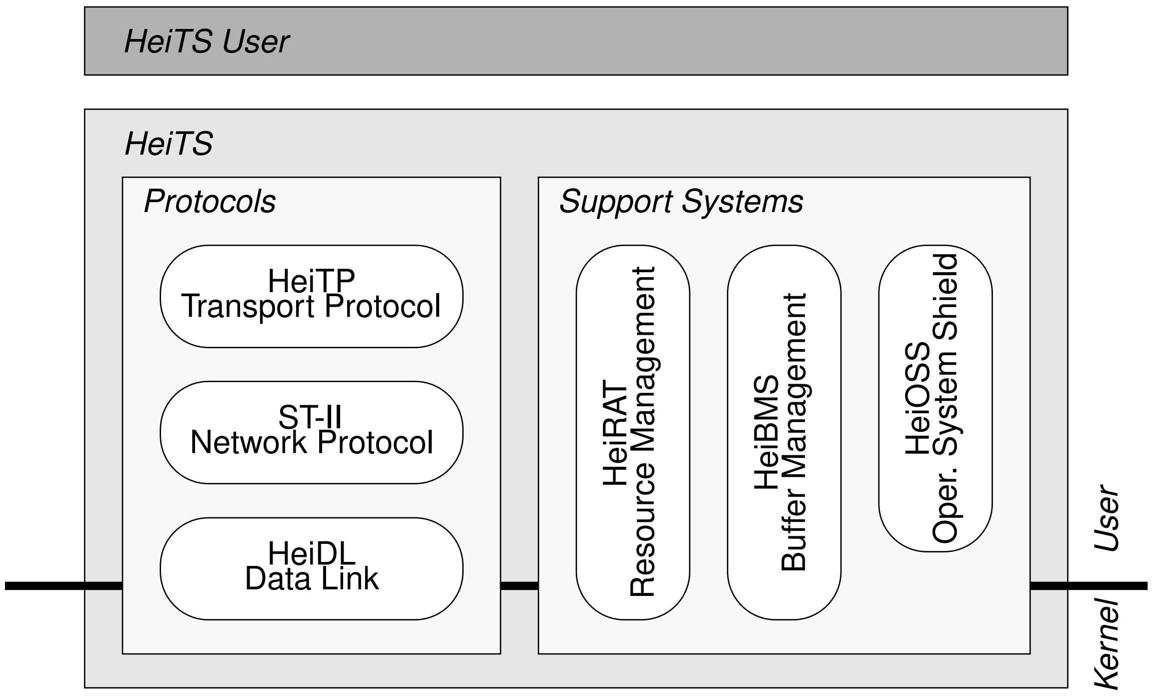

Figure 2: HeiTS Components and Their Implementation Location.

2 Components

HeiTS provides protocols of transport, network, and data link layer, apart from protocols HeiTS contains also components for resource management, buffer management, and operating system abstraction (Figure 2). This section provides a short overview of these components.

2.1 Protocols

The Layer 3 and Layer 4 protocols used within HeiTS are the network layer protocol ST-II [Top90] and the transport protocol HeiTP [DelHal92]. Both protocols are connection-oriented and enable the transfer of continuous-media data.

ST-II is a protocol of the Internet family, it is related to IP and both protocols can coexist in one system, however, there are several differences:

- ST-II is connection-oriented based on ‘streams’. A stream is a simplex connection between an origin and one or more targets. If the underlying network supports multicast, the amount of packets which have to be exchanged, e.g., in multimedia conferences, can be reduced [TwaHer93].
- During the creation of a connection, resource requirements are negotiated by the exchange of ‘flow specifications’.

ST-II provides most functionality needed for multimedia data transfer. However, there is still need for a transport layer service which (as HeiTP) provides the following mechanisms not supported by ST-II [DelHal92, DelHal93]:

- Segmentation of data packets (reassembly if requested by the user).
- Flow control to prevent the sender from over-running receivers. A rate-based mechanism is used since traditional sliding-window techniques are believed to be not appropriate for continuous-media data transfer.
- Error detection with user selected handling options, e.g., to ignore, to indicate, or to correct transmission errors.
- Media scaling to adjust the transmitted data streams to changes in network characteristics such as throughput.

At the bottom of HeiTS exists HeiDL (Heidelberg Data Link) [Twa93] providing the data link layer protocol and the interface to networks. HeiDL functions are used by the network layer to send and receive data, and to establish multicast groups. Data packets received are distributed to established ST-II connections. Depending on the capabilities of the used network type (Token Ring, Ethernet, FDDI, B-ISDN,...), HeiDL transmits outgoing data packets with respect to their time-criticalness.

2.2 Transport System Interface

The native interface of HeiTS is structured as an upcall interface, i.e. functions are called in a way that reflects the flow of data [Cla85, p. 171]:

"An upcall need not go precisely upward. Our goal is that procedure flow should map on to the natural flow of control in the program, whether that is up, down, or sideways."

Within the upcall interface, the "application" (the transport service user on top of HeiTS) registers upcall functions in the transport layer. These functions are called when a connect indication or a data packet has been received. This way, the processing of data packets is done directly after the packet has been received and will not be shifted to a later point in time (as it may happen in a downcall interface, in which the application has to retrieve the data from the transport system).

2.3 Operating System Shield

Since we implement HeiTS on different operating system platforms we use an intermediate layer between the native operating systems and our transport system code to ease moving code between platforms. This 'Heidelberg Operating System Shield' (HeiOSS) provides methods (as far as possible implemented as macros to avoid costs of function calls) for semaphores, message queues, and shared memory.

2.4 Buffer Management

Protocol packets sent across a network consist of several parts; the data an application wants to transfer and the protocol headers of each protocol layer. *Buffers* are the representations of packets in memory. HeiBMS avoids data copy operations in the methods it provides to insert or remove data, to split a buffer, and to build a logical buffer out of multiple smaller memory pieces containing data and headers.

HeiBMS is used by the whole HeiTS protocol stack and may also be used by software layers on top of HeiTS. Additionally, as far as possible, we map other memory usage schemes such as UNIX' mbufs to HeiBMS and vice versa to avoid otherwise needed copy operations.

2.5 Resource Management

The resource management system HeiRAT is the focal point for resource allocation. Based on the QoS parameters delay, jitter, throughput, and reliability specified during connection setup it performs calculations to determine required system resources and administrates these resources.

HeiRAT manages all the resources which are critical for the execution of continuous-media data processing: CPU time, network bandwidth, and memory. For CPU, for instance, a rate-monotonic scheduler [LiuLay73] is used.

Not all users need the same *kind* of QoS. For some users it is important to get the specified quality during the whole time without any degradation, others may accept some quality degradation, especially if this service is not as expensive as it would be otherwise. The first kind of QoS is necessary for production-level applications, e.g., in a movie studio. The second kind of QoS is especially useful for playback consumer applications. Based on these QoS classes, different methods for resource reservation can be used.

Guaranteed QoS needs a *pessimistic* resource reservation which may lead to an underutilization of resources – worst-case assumptions have to be used, thus, the full amount of resources which might be needed during the live-time of the applications has to be reserved exclusive for this application even if average resource utilization is much lower. If no strict guarantees are needed one can use an *optimistic* approach, which reserves less resources, e.g., for the average workload only and accepts some chance of missing required resources which means degradation of perceived QoS.

The current version of HeiRAT is mainly concerned with applications which require a guaranteed QoS, future versions will provide more support for applications using adaptive methods.

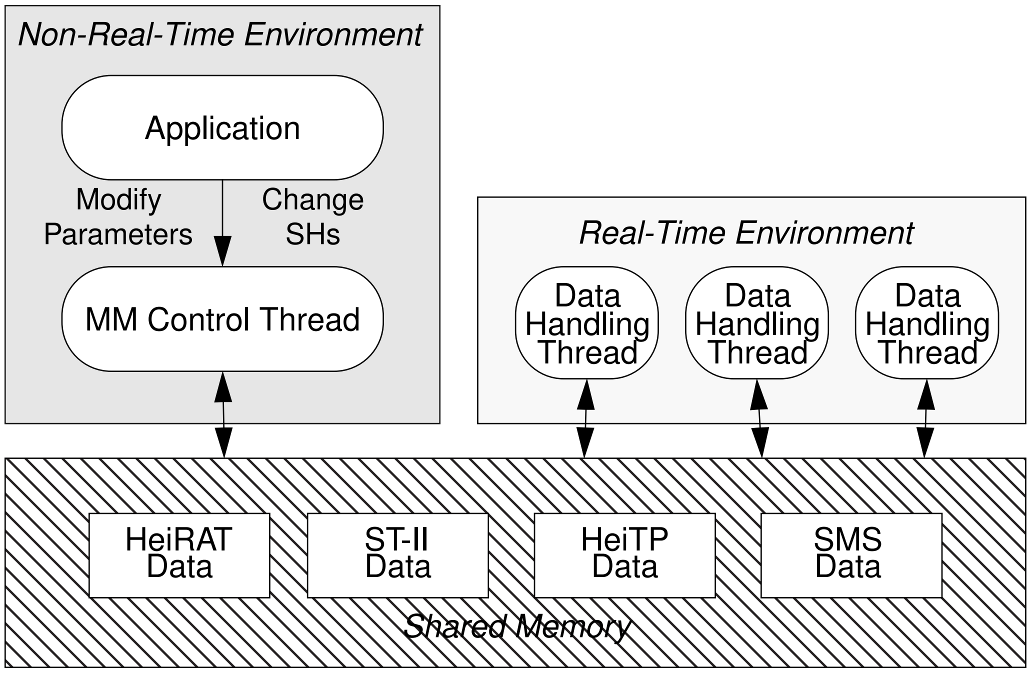

Figure 3: Real-Time and Non-Real-Time Execution Environments.

3 Execution Structure

This section explains how the protocol functions for control and data handling are executed within HeiTS. First, the different environments for real-time and non-real-time functions are discussed. Then, a general description of the process model is given, followed by sections explaining the processing with these activities and the creation of these activities. The last two parts discuss how the usual processing path can be interrupted.

3.1 Real-Time and Non-Real-Time Environment

A multimedia application consists of several logical parts:

- The time-critical continuous-media processing part which reads data from a source, performs operations on the data, and writes the data out to a sink.
- Non-time-critical functions which do not operate on continuous-media data. Such functions either build the user interface or control the real-time environment.

Since real-time and non-real-time operations are separated into distinct environments, the amount of data exchanged between functions of the two environments has to be considered. The user interface operations share only a moderate amount of information with the real-time environment: mainly, they control data representation by setting parameters like the volume of audio output. The other non-real-time functions control the real-time environment and exchange, therefore, a larger amount of data with real-time functions: e.g., while establishing a connection they create control data structures which are accessed during the time-critical data transfer phase.

Sharing data structures between connection user and connection establisher is natural, yet the use of a different mechanism (like message exchange) between user interface and real-time environment yields better shielding of system parts without severe drawbacks due to communication costs. Figure 3 shows the resulting structure.

An important distinction between real-time and non-real-time environment is that memory used in the real-time environment is pinned in physical memory (avoiding delays induced by paging). For non-real-time functions the memory may be paged out to external memory without severe harm.

3.2 Process Model

All processing in the user-level part of our implementation is done by ‘threads’: separately schedulable entities sharing address spaces. In particular, threads share the address space in which data structures are dynamically created.

In its current version, AIX provides only heavy-weight processes which do not share data structures created dynamically. On top of such a process, a coroutine package can be implemented, however, these entities are not separately schedulable by the kernel and suffer from well-known problems like blocking of the process (and all threads within the process) due to I/O operations.

In our system, threads corresponding to the above definition are implemented in the following way: An AIX process builds the activity for a thread, data structures are shared using a memory management system which provides us with *malloc* and *free* functions which replace the standard functions from the system library. The new functions operate on a shared memory segment protected by semaphores (Figure 4), thus, all memory allocated by a process is accessible to other processes using the memory management.[2]

2. This is in opposite to standard shared memory mechanisms which construct a memory segment that exists in addition to memory segments owned by individual processes. Allocation of memory happens inside of memory areas accessible to the allocator only and not in the shared region.

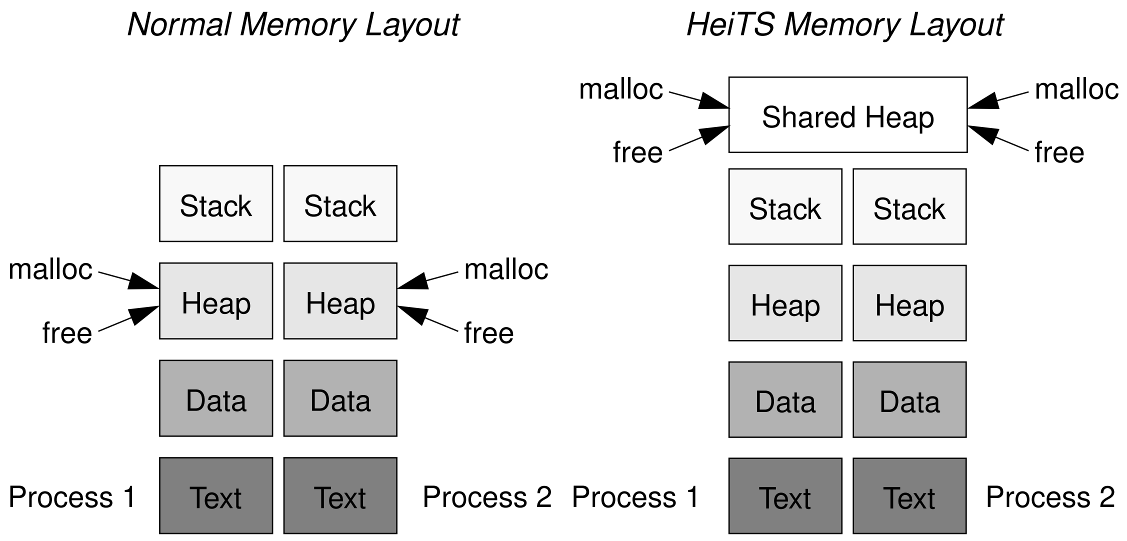

Figure 4: Memory Usage by HeiTS.

3.3 Processing within the Transport System

The transport system consists of a control and a data handling part. Control and data handling functions are executed in separate threads, these threads perform function calls between protocol layers, for the send and also for the receive part.

All functions of the control part are executed for all connections within one non-real-time thread. This thread performs the operations for the creation, modification, and destruction of connections for all protocol layers. For the data handling part, a thread per connection is used. Such a data thread executes all the functions within the protocol stack following the upcall model. The steps of protocol processing are shown in Figure 5.

On the sending side of a data connection, this scheme is straight-forward; the send functions of all protocol layers are called in turn under the thread of the producer.

Due to its asymmetric nature, the receiving side of a connection is different: the receive functions of the consumer and the transport layer are registered at initialization with the next lower layer (transport and network layer, respectively). Then, a *main-loop* function is executed which blocks until data is available on the network interface. If a packet arrives, the interrupt handler distributes the packet according to the information stored in the packet to the control thread or to the data handling thread which serves the connection, then the waiting thread is unblocked. Protocol functions are called in turn and after the consumer has finished data processing, the thread simply returns.

In summary, there is no distinct data thread for HeiTS, the thread is always shared with software on top of HeiTS. Where the receive thread will be created and given to the transport system, will be discussed in the next section.

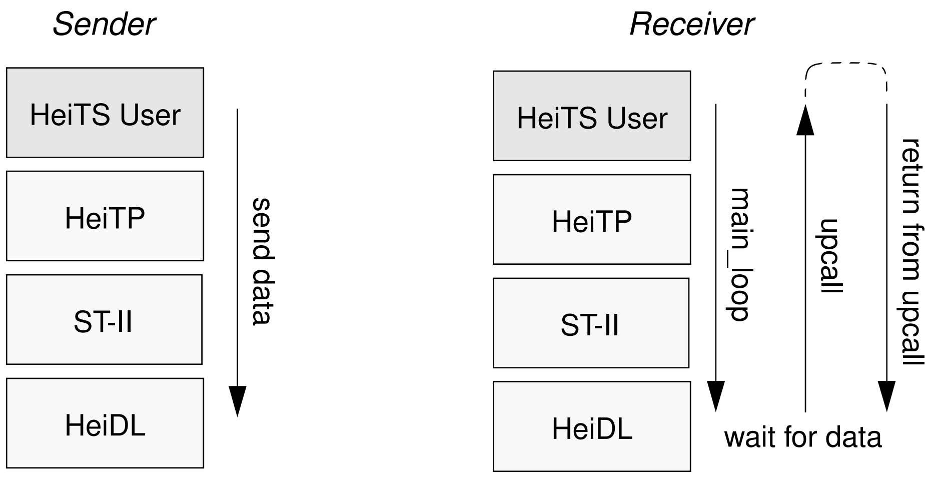

Figure 5: Protocol Processing with Pure Procedure Calls.

3.4 Creation of Threads to Receive Data

As described in the previous section, there exists one data handling thread per connection performing protocol and consumer function processing. These receiver threads are created outside of HeiTS since not HeiTS but the consumer has to accept or reject the connection. If the consumer decides to reject the connection, it calls the transport system to send the reject protocol packet, otherwise, it calls HeiRAT to reserve the required resources, e.g., CPU capacity[3], creates the data handling thread, and calls the accept function of the transport system. The CPU scheduler is informed about the data handling thread and assigns a real-time priority. Then the data handling thread calls the *main-loop* of the transport layer and provides this function with the information about its system resources and the connection it serves.

If an ST-II packet is received containing a connect indication with a target 'down-stream' from this host, the network layer has to perform routing operations (which are defined within the ST-II protocol). For these operations, ST-II performs the reservation and registration functions described above and creates a data handling thread which will execute protocol functions up to the network layer.

3.5 Processing of Control Operations

The HeiTS control thread processes all operations to establish connections, to modify given parameters of a connection, and to release a connection. The origin of such requests could be on the local workstation or on a remote workstation, to get a request from the remote peer, the control thread waits on the network interface for incoming control packets. Since the control

3. The required CPU capacity consists of CPU time for HeiTS plus time for other SHs until the data is stored in a queue or given to an output device, thus until the thread returns (cf. optional SHs in Figure 1); the sum is unknown inside of HeiTS.

thread performs a blocking read on the network interface for incoming control packets, a local user application having such a request needs a method to interrupt the waiting control thread and to inform it about the desired action.

The application which has a request fills a data structure in shared memory. After this, it sends an event[4] and waits until it receives a reply. The control thread is interrupted through the reception of the event, it returns from the blocking read operation on the network interface and returns to the network layer. ST-II detects that no control packet was received and returns to the transport layer which acts in the same way. Finally, the layer on top of HeiTS has the control about the thread. This layer forms the interface to the 'outside world' and knows how to handle the request, it reads the data structure in shared memory describing the requested operation, performs the necessary functions and sends a reply. After the processing of the request is finished, it calls the *main_loop* function of the transport system again.

3.6 Exception Mechanism for Data Handling Threads

Data handling threads on the receiving workstation wait on the network interface for incoming packets of their connection. When a connection is closed, the thread serving it has to be removed. Since the thread may have some system resources like buffer space in use, a method for a clean shutdown is required. For this purpose, a similar scheme as described in the previous section is used – to inform the thread about the 'destroy' request, an event is sent.

Each HeiTS layer and layers on top of HeiTS may have some system resources in use when an exception occurs that requires the thread to exit; the thread returns from the network interface without a packet but with the information to leave the system. Then each layer returns its resources to the system and returns from its main loop to the next layer. Since each layer has to return its resources, no layer is allowed to exit immediately, the highest layer finally exits the thread.

4 Higher Layers

We do not expect, that applications are built directly on top of HeiTS. In most cases, there will be at least one intermediate layer, often there will be more than one. First, a surrounding environment for the execution of HeiTS is needed. This shell provides the SMS mechanisms to access the real-time environment from non-real-time functions, i.e., the operations to chain SHs.

The approach used to communicate between real-time and non-real-time functions is similar to the X Window system approach. Clients, as the sender and the receiver in Figure 6, contact a server (control thread) to get specified tasks executed. Examples of such tasks are to advise the SMS to connect SHs within the server or to set certain attributes on them.

The functions of such a shell provide only the basic mechanisms to get access to the real-time environment. On top of such a shell, we are developing HeiMAT, the Heidelberg Multi-

4. The used event is a kernel mechanism (similar to a UNIX signal) which is also used to unblock a thread waiting for a network packet.

media Application Toolkit [KaeHeh92], which builds more complex abstractions. Two different kinds of functions are contained in HeiMAT, first on the lower level of HeiMAT, operations on a local basis, i.e., abstract devices (all devices are controlled in the same manner), data-type-related functions (applying automatic conversion), and synchronization functions. Secondly, the upper part of HeiMAT provides functions for distributed continuous-media applications, i.e. control of conferences. Since such functions ease the construction of distributed continuous-media applications, developers can concentrate on the application specific tasks.

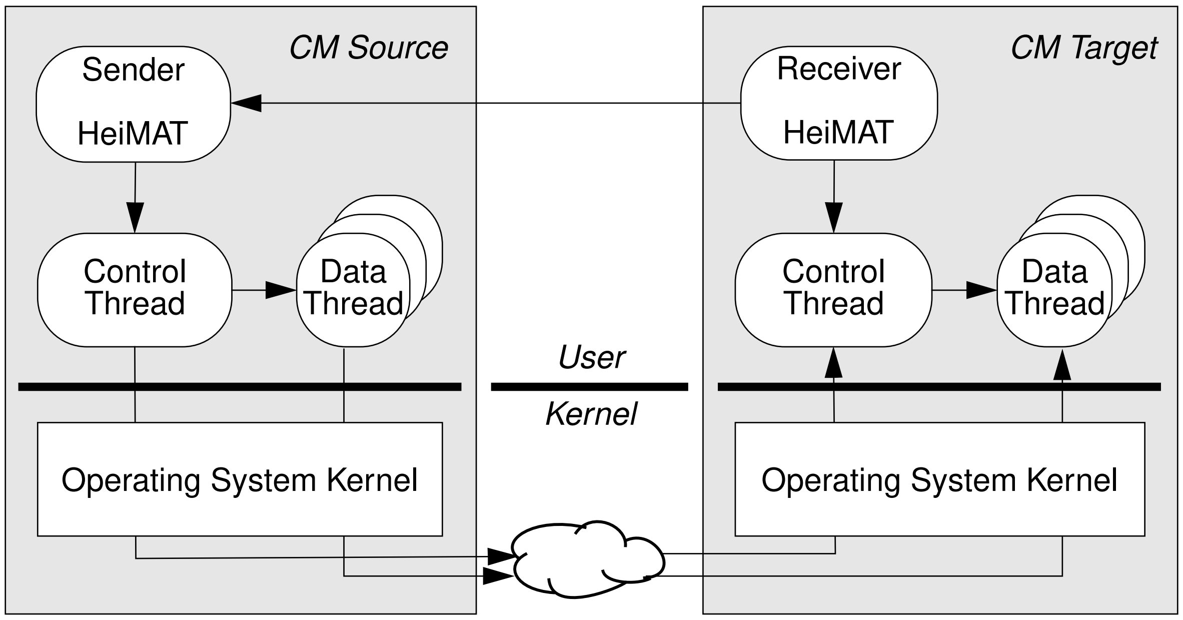

Figure 6: Principal Thread Structure for Distributed Continuous-Media Applications.

5 Status and Future Work

The protocols and support systems have been successfully implemented. Currently, Token Ring and Ethernet networks are accessible via HeiDL. Now we perform measurements on HeiTS and will, based on this data, start performance improvements. Additional future work items are increased support for adaptive resource management schemes and access to more network adapters.

On top of HeiTS several distributed continuous-media applications are under development, examples are HeiPhone and HeiDI. The audio and video conferencing system HeiPhone [KaeWol93] provides the ability for an arbitrary number of users to communicate using images and voice. HeiDI is a video distribution service which allows clients to retrieve videos on demand from a server.

Acknowledgments

We would like to thank all department members which provided us with information about system and component specifics and which spent time with us during discussions. We wish to express our appreciation especially to Werner Schulz who has written parts of a former paper on this topic.

References

BoeDam93	S. Boecking, S. Damaskos, L. Delgrossi, A. Fartmann, G. Grolms, R. Hammerschmidt, G. Hoelzing, B. Metzler, I. Milouscheva, G. Schnaars, J. Sandvoss, J. Sokol, B. Weise and M. Zitterbart , The BERKOM-II MultiMedia Transport System (MMT) Version 2.1, Internal BERKOM Working Paper, May 1993.
Cla85	D. D. Clark, The Structuring of Systems Using Upcalls, Proceedings of the Tenth Symposium on Operating Systems Principles, Operating Systems Review Special Issue, vol. 19, no. 5, pp. 171-180, Washington, December 1985.
DelHal92	L. Delgrossi, C. Halstrick, R.G. Herrtwich and H. Stuettgen, HeiTP: A Transport Protocol for ST-II, Proceedings of GLOBECOM' 92, Orlando, FL, December 1992.
DelHal93	L. Delgrossi, C. Halstrick, D. Hehmann, R.G. Herrtwich, O. Krone, J. Sandvoss and C. Vogt, Media Scaling with HeiTS, Proceedings of the First ACM Multimedia Conference, San Diego, 1993.
DelHer93a	L. Delgrossi, R.G. Herrtwich, C. Vogt and L.C. Wolf, Reservation Protocols for Internetworks: A Comparison of ST-II and RSVP, Fourth International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, UK, November 1993.
DelHer93b	L. Delgrossi, R.G. Herrtwich and F.O. Hoffmann, An Implemantation of ST-II for the Heidelberg Transport System, to be published in Internetworking: Research and Experience, 1993.
Fer90	D. Ferrari, Client Requirements for Real-Time Communication Services, Technical Report TR-90-007, International Computer Science Institut, Berkeley, CA, March 6, 1990.
FerBan92	D. Ferrari, A. Banerjea and H. Zhang, Network Support for Multimedia: A Discussion of the Tenet Approach, TR-92-072, International Computer Science Institute, Berkeley, CA, October 1992.
GolDea90	D. Golub, R. Dean, A. Forin and R. Rashid, Unix as an Application Program, in *USENIX Conference Proceedings*, pp. 87-96, USENIX, Anaheim, CA, Summer 1990.
HerWol92	R.G. Herrtwich and L.C. Wolf, A System Software Structure for Distributed Multimedia Systems, Proceedings of the Fifth ACM SIGOPS European Workshop, Le Mont Saint-Michel, France, September 21-23, 1992.
IBM92	IBM, Multimedia Presentation Manager/2, Programming Guide, S41G-2919-00, IBM Corporation, Boca Raton, Florida, 1992.

KaeHeh92	T. Kaeppner, D. Hehmann and R. Steinmetz, An Introduction to HeiMAT: The Heidelberg Multimedia Application Toolkit, Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video, pp. 362-373, San Diego, CA, November 12-13, 1992.
KaeWol93	T. Kaeppner and L.C. Wolf, Architecture of HeiPhone: A Testbed for Audio / Video Teleconferencing, TR 43.9316, IBM European Networking Center, Heidelberg, Germany, 1993.
KroMcK93	O. Krone, B. McKellar, K. Reinhardt, W. Schulz and L. Wolf, The Heidelberg Buffer Management System, Internal Working Document, IBM European Networking Center, Heidelberg, Germany, July 1993.
LefMcK89	S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, in *The Design and Implementation of the 4.3-BSD UNIX Operating System*, Addison-Wesley, Reading, Mass. et al., 1989.
LiuLay73	C.L. Liu and J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Realtime Environment, Journal of the ACM, vol. 20, no. 1, pp. 47-61, 1973.
ParPin92	C. Partridge and S. Pink, An Implementation of the Revised Internet Stream Protocol (ST-2), Internetworking: Research and Experience, vol. 3, no. 1, 1992.
Svo89	L. Svobodova, Implementing OSI Systems, IEEE Journal on Selected Areas in Communication, vol. SAC-7, no. 9, pp. 1115-1130, September 1989.
Top90	C. Topolcic, Experimental Internet Stream Protocol, Version 2 (ST-II), RFC 1190, October 1990.
Twa93	B. Twachtmann, The Heidelberg Transport System: Data Link Layer, Internal Working Document, IBM European Networking Center, Heidelberg, Germany, July 1993.
TwaHer93	B. Twachtmann and R.G. Herrtwich, Multicast in the Heidelberg Transport System, Technical Report 43.9306, IBM European Networking Center, Heidelberg, Germany, 1993.
VogHer93	C. Vogt, R.G.Herrtwich and R. Nagarajan, HeiRAT: The Heidelberg Resource Administration Technique - Design Philosophy and Goals, Kommunikation in Verteilten Systemen, Munich, Germany, March 3-5, 1993.
ZhaDee93	L. Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappala, RSVP: A New Resource ReSerVation Protocol, IEEE Network, pp. 8-18, September 1993.