A Component-based Architecture for Streaming Media

Alexander Eichhorn, Winfried Kühnhauser
Institute of Practical Informatics and Media Informatics
Technical University of Ilmenau
{alexander.eichhorn@rz, winfried.kuehnhauser@prakinf}.tu-ilmenau.de

Abstract. This paper describes the design and implementation of a component based distributed multimedia application that was developed to investigate a new design principle for resource-efficient layered system architectures. In order to provide flexible and adaptable application scenarios, different applications are composed from three basic generic component types. Combining different component types with different types of interaction and locating them on different nodes of a distributed system then results in large distributed multimedia applications where each application has different types of sophisticated resource requirements.

Beyond the scope of our test-bed we believe that this component-based architecture is well suited for general large scale streaming media applications such as video-conferencing and video on demand services.

1 Introduction

Distributed multimedia applications dealing with continuous real-time streaming media within networks of computer systems generally require a large amount of different types of computing resources. Video streaming media generally require a high network bandwidth. Unreliable networks and varying network loads require physical memory for buffering stream data. Copying between application and I/O buffers, compression algorithms and stream format conversion require large amounts of CPU cycles. Additionally, many distributed multimedia applications have quality of service requirements such as timeliness, confidentiality, or robustness with respect to transient communication failures. All these properties require additional CPU cycles or bandwidth for reserving resources or encrypting data streams.

It is a well known fact today that general purpose operating systems as well as general purpose communication paradigms for distributed systems do not cope very well with such types of applications. A major core of the problem has been found in the isolation imposed by layered system and protocol archi-

tectures [Ten89, CT90, CWWS92, SS93, XP99]. Within this context, we are currently investigating a new concept based on *knowledge transfer*. Knowledge transfer overcomes the strict layer separation in traditional layered architectures by explicitly sharing knowledge like state informations, state changes (events) and strategies for event handling and resource allocation between applications and system layers.

Fundamental to the concept of knowledge transfer is the observation that layered architectures not only enclose algorithms and data structures within the layers but also hide knowledge that in many cases would be useful on other levels in order to implement more sophisticated policies. As an example, the knowledge that a mobile component of a distributed system currently uses a wireless communication link might trigger a multimedia application to choose a different stream format that is more robust to partial packet loss. On the other hand, policies on the lower layers that deal with packet loss are often more efficient if they know about certain stream properties.

The basic idea of knowledge transfer is that information local to a specific layer (or, more general, subsystem) that will provide other subsystems with a more educated decision base is shared among the subsystems in a well-defined and controlled way. Knowledge itself is represented in a common and uniform way among all subsystems.

For simulating and measuring the effects of knowledge transfer, we are currently developing a distributed multimedia platform. In order to be able to investigate current and future technologies for compression, error control, data transport, resource management and adaption in the context of continuous media we designed the basic architecture with scalability and flexibility in mind. Furthermore the components, interfaces and interactions are general enough to serve as basic building blocks in various application scenarios, like video conferencing, video-on-demand and video control.

Section 2 describes design principles and constraints for satisfying the different requirements in quality, reliability, distribution, interaction and dynamics of all of these application classes. Section 3 is an overview of the general system architecture, and section 4 outlines the basic design issues for each of the architecture's components. Section 5 addresses some lessons learned for distributed multimedia platform design.

2 Design Principles for Multimedia-Systems

The principles we consider fundamental for building scalable and adaptable distributed architectures for streaming media are based on the natural constraints of continuous media and large-scale heterogenous distributed systems. We did not consider implementation related issues, such as heterogeneity and system support of actual operating systems and communication protocols. These aspects are subject to shorter development cycles and we are looking for more general concepts, drawing conclusions for future development.

The abstraction of a stream, a continuous flow of periodic data chunks with

an inherent isochronous nature, is well suited for the description of continuous media. Time affects the validity of data: late delivery is valueless, and early delivered data is also less valuable because it requires buffer resources in order to maintain its isochronous nature. Processing, transmission and presentation of continuous media, such as video and audio, will impose significant demands on all resources (CPU, memory and communication bandwidth) for the foreseeable future, because increasing capabilities are instantly consumed by the demands for higher quality and quantity. Continuous media streams are partially tolerant to loss of data because of the human perception characteristics. The same reasons allow the scaling of media content in different dimensions (temporal, spatial, colour space) without an perceivable decrease in presentation quality. However, for domains with highest requirements in quality and reliability such as medical applications or air traffic control, these assumptions do not hold.

In heterogenous distributed systems the performance and characteristics of the involved hardware and networks may vary in orders of magnitude. One end of the scale is populated by cutting edge server and desktop systems and reliable high performance networking technologies with (partial) support for real-time delivery. On the other end are embedded and mobile devices with very restricted CPU and memory resources and the generally low bandwidth and high failure rates of wireless networks. In such scenarios additional types of resources like energy preservation and presentation capabilities become important. Furthermore we must also consider features of specialised hardware, embedding complex coding and encryption algorithms, which is likely to be integrated in future mobile and stationary equipment.

Conserving resource utilisation: Distributed multimedia applications in general require many and also many different computing resources for processing, storing and transmitting media streams. Consequently, many techniques have been developed that aim at an efficient resource usage. Network bandwidth is reduced by various classes of coding and decoding algorithms (*codecs)* [HWA97, WSL00] with different impacts on data quality. In order to reduce the amount of physical memory needed for buffering and the amount of cache and TLB pollution, thread-based processing models and zero-copy IPC mechanisms are used.

However, these techniques often come with side effects. Codec algorithms for example that on the one hand reduce communication bandwidth on the other hand require additional CPU resources. They also influence the buffer management and communication failure strategies, because in several compressed video stream formats the loss of different stream sections is of different importance. Consequently, techniques reducing the requirements for one type of resource might imply that additional resources of different types are needed, resulting in a global and highly complex optimisation policy.

Guarantee-oriented resource management: In order to achieve timely delivery and processing, applications need guarantees that the necessary resources

will be allocated and scheduled at the required times. To accomplish this task, a reservation based management for all resources is needed. QoS managers must implement functionality for negotiating contracts about the desired resources with applications, to perform admission control and scheduling and to notify applications when guarantees can't be kept. Applications must also comply to the contract by not exceeding their share.

Adaption: In situations of failures and especially in error-prone wireless networks the concept of reservation is not always successful. Here guarantees must be backed up by adaption policies to cushion the impacts of unexpected events.

Adaption policies may be part of many different system layers. On the user interface layer of applications users may wish to balance the resource usage with other activities in a coarse grained way and on a larger time scale. Applications which typically know best how to automatically respond by reconfiguring components and system services, are able to adapt more quickly and preciously. On lower system levels adaption can be achieved by exploiting application specific knowledge within system services, which tends to be the most efficient place for making decisions.

One way to achieve fine grained resource management and adaption is to use a *staged application design*, where stages are concurrent and are linked by asynchronous event queues [WC01] that avoid synchronisation delays. Resource shortages can be addressed by a combination of adaption strategies and sophisticated stream format-aware scheduling algorithms which discard data units before they are processed.

End-to-end view of the system: Optimal decisions for the whole system are only possible when various aspects from all involved subsystems of a distributed multimedia system are taken into account. As an example, recent compression techniques combine data compression at the sender with error concealment [WSL00]. As a consequence all local algorithms for resource management, adaption, error and flow-control and compression have to cooperate by exchanging information, events and feedback.

Separating the handling of media and control The fundamental differences between communication model and resource requirements of media streams on the one hand and control flows on the other imply to separate both. The handling of streaming media requires active processing while control mechanisms are structured according to the more passive client/server paradigm. The blocking of clients and servers as well as the request/reply pattern is inappropriate for media handling, which must keep the strict timing constraints. The separation also allows direct streaming of data between specialised processors (DSP's for coding or encryption) or kernel subsystems without passing user processes. This in turn meets the first principle of conserving resource utilisation by avoiding context switches and utilising hardware capabilities.

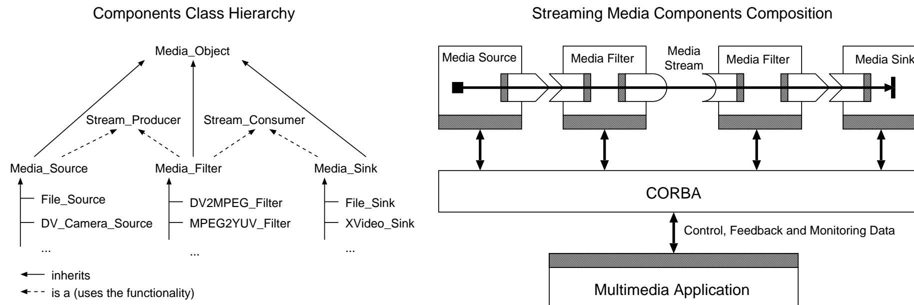

Fig 1. Streaming-Media Components: Class Hierarchy and simple Application Scenario

3 Overall System Architecture

Apart from the design principles outlined in section 2 our architecture is based on the concepts of object orientation. The basic classes are typical elements of a streaming-media architecture: sources, filters and sinks (figure 1). Media streams always flow from sources to sinks. Typical media sources (stream producers) are files, multimedia databases, video cameras, video grabber and tv-tuner cards. Media sinks (stream consumers) are also files and multimedia databases, but in addition, display windows (XVideo), decoder cards or video-cut cards. Filters, which are consumers and producers at the same time, change contents and characteristics of the incoming media streams and produce outgoing media streams of different types. We can divide filters into three subclasses. Format converters (media codecs) modify the data format of a stream. Time converters (synchroniser, smoothing filters for jitter reduction and caches) change the temporal behaviour of media streams by delaying data in a controlled fashion. Scaling converters are directly modifying the contents of a stream by dropping parts of the information. Scaling can take place in different dimensions (temporal, spatial, colour space, frequency or amplitude scaling). Some scaling methods are codec dependent, others are codec independent. Meaningful scaling requires knowledge about structure and contents of a media stream and has to take the behaviour of media decoders into account. In order to compose more complex application scenarios (multicast streaming, fault tolerant communication, load balancing), the components additionally contain mechanisms for forking and joining media streams.

Besides the pure media-specific components there are components and classes that allow for a reproduceable simulation of environments and the investigation of individual system components and algorithms. Useful simulation tools are load-generators for stress testing basic system resources (CPU, memory, bus bandwidth), operating system services (IPC, resources schedulers, network services) and network resources (link bandwidth, router queues). Other components for testing the robustness of codecs as well as the fault-tolerance of communi-

cation architectures are jamming filters and noise generators. These allow a reproduceable simulation of network characteristics like variations in bandwidth, delay, jitter, loss, bit error rate, burst errors, short-term disconnections and link failures. So we are able to investigate the behaviour of our components as well as system services and future networking technologies (W-LAN, GSM, UMTS, Satellite networks), even if they are not yet physically available.

The uniform structure of the control interfaces and in particular the general and open definition of the data interface, which is capable of attaching different communication modules, allows arbitrarily complex compositions of our components (figure 1). Protocols and IPC mechanisms for communicating the media streams are exchangeable. The basic functionality of each component, inherited from the class *Media_ Object*, contains flexible modules and interfaces for configuration, adaptation, feedback, monitoring and an envelope for the core processing algorithms, like media codecs, file and database handling and camera control. The ability to communicate media streams between components stems from the classes *Stream_ Producer* and *Stream_ Consumer*. These allow the flexible binding of streaming-capable communication modules as input resp. output for components. Communication modules always exist as pairs of senders and receivers, which use IPC mechanisms and network protocols provided by the underlying system. An uniform buffer management between all local components avoids expensive copy operations where possible.

The interface of each component has four logical interface types for configuration, feedback, monitoring and streaming data. The configuration interface is responsible for controlling the basic architecture (classes *Media_ Object*, *Stream_ Producer* and *Stream_ Consumer*) and the special functionality of sources, filters and sinks. Interactions involve connection management (create, destroy, connect, disconnect), internal configuration (setting special parameters) and stream handling (play, pause, stop, fast forward, rewind). The feedback interface forwards all not internally treatable events to other components or to the application. The monitoring interface provides access to raw or statistically processed monitoring informations and test outputs. Components can be configured to continuously push monitoring data outwards or to provide collected informations on request. In order to make test scenarios centrally controllable and configurable, we use CORBA as middleware technology. This allows freedom with the selection of the concurrency model (process- or thread-based) and provides us with the advantages of distribution transparency. It is well known that the RPC mechanisms of CORBA are ill-suited for the transfer of continuous media streams [MSS99]. To overcome these drawbacks, our data interface is implemented by communication modules, which support continuous media streaming.

Implementing additional components is very easy: The basic functionality and thus all advantages of the architecture, like simple composition, adaptability, monitoring mechanisms, communication modules etc., are inherited from the base class *Media_ Object*. Only the specific algorithms must be integrated into the core of the component. For a new type of filter only the integration of the

concrete codec algorithms into the general filter class is required. In addition it is possible to integrate specialised algorithms and data structures for monitoring, resources management and adaptation. Since communication mechanisms and system support are transparent to the internal architecture, all existing components will profit from improvements without modifications. The great importance we attached to clear and general interfaces pays off in the reuseability of all components in different application scenarios, like video conferences, video-on-demand, distance learning and video control.

4 Basic Design Issues for System Components

The internal architecture of the components (class *Media_ Object*) is governed by a clear separation of time-critical media stream handling and internship control. From an abstract point of view each component consists of a control instance, the *Watcher*, a concurrent and always active stream handling instance, the *Worker*, and an input queue containing sections of an incoming medium stream. Workers produce, process and consume media streams and transfer them to the input queues of other application components using streaming-oriented communication modules (see figure 2).

Stream data arriving asynchronously at the data interface are buffered in the input queues. They are processed whenever appropriate resources are available and then sent synchronously to the next application component. Concatenating application components thus results in a staged processing chain. Whenever the input queue of a component becomes empty, the corresponding worker blocks, releasing CPU resources for components with higher loads.

In situations with scarce resources, stream data will accumulate within the stream buffers, and strategies for dropping buffer contents will be needed. As the importance of buffer contents depend both on time as well as on content (e.g. frame types), simple drop-tail FIFO ordering strategies are replaced by time-aware and content-aware algorithms that depend on a specific stream format. These algorithms are activated by events such as high-water marks in buffers or the exceeding of buffer processing deadlines. The staged design allows for a fine-grained reaction to situations with low resources both within each component as well as it allows to respect the importance of different stages within the processing chain.

In order to avoid physical copies at the data interfaces, a single unified stream buffer management is used for all application components. Buffers are passed by reference, and reference counters allow for the sharing of read-only buffers. Explicit synchronisation is not required, because only producers request writable buffers and only consumers read and release buffers. Other important properties of the buffer management are scatter/gather operations for collecting several buffers and buffer metadata that provide application-specific information about buffer contents and are used to implement time-tracks, content-dependent scheduling and caching algorithms. In connection with real time protocols (RTP), which already use headers for time information, we thus achieve

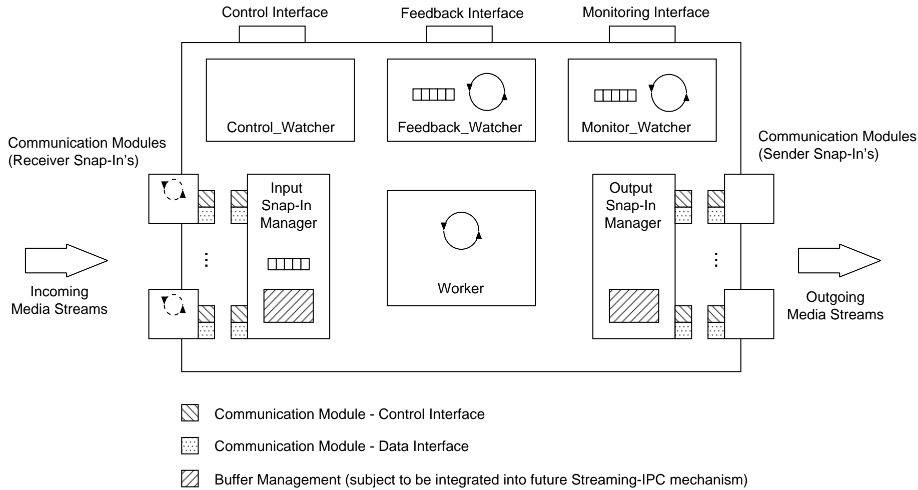

Fig 2. Internal Structure of Components

a contiguous time-track from the stream source to the sink.

In order to hide details of buffer and queue administration, scheduling, address space boundaries and communication mechanisms from the Workers, streaming-capable communication modules (senders and receivers) exist, which serve inputs and outputs of the data interface. Current modules implement local IPC mechanisms (such as Unix pipes, shared memory and thread communication) as well as tying up to network protocols such as UDP and RTP. In order to closer investigate new transport mechanisms and architectures for IPC and protocol processing, communication modules can also be implemented as separate components. An additional administrative instance, the *snap-in manager*, contributes here substantially to the flexibility and expandability of the overall architecture. Snap-in managers control several different sender and receiver modules. The two basic types, input snap-in managers and output snap-in managers are implemented by the classes *Stream_Producer* (manager for transmission modules) and *Stream_ Consumer* (manager for receipt modules). Producers basically duplicate data streams by means of several sender modules, while consumers collect several incoming data streams into a single one. Snap-in managers simplify several otherwise complex scenarios such as efficiently supporting heterogenous senders and receivers or local duplication of streams. e.g. for archiving or multiplexing to receivers via networks of different physical properties. Parallel receivers allow for dynamic load balancing as well as for fault tolerance strategies without reconfiguring the overall application architecture. Especially, in systems with mobile components, roaming between different physical networks can be achieved without any interruption of the stream flow.

The overall architecture is controlled and monitored by different types of *watchers* that are part of each application component and execute concurrently.

A configuration watcher controls the worker, the communication end points, the feedback watcher and watcher of the sensoric system. While the configuration watcher is activated by calls to the RPC-interface of a component, all other watcher types listen to events delivered by an event notification system. The feedback watcher responds to asynchronous events by either enforcing a local application-specific adaption policy or forwarding the events to outside receivers. The watcher of the sensoric system collects data from sensors within an application component (such as buffer resource utilisation, work loads, fault rates) and responds to corresponding queries.

Since the Unix API lacks a general event notification scheme, additional threads wait for events from independent sources (e.g. select() for RSVP acknowledgements and the receive paths of receiver snap-in's), which currently requires additional watcher resp. receiver threads to mediate events.

Some aspects of the architecture have not yet been discussed. Among them is a stream-oriented kernel-based IPC mechanism, which combines a global, zero-copy buffering mechanism for streams with an efficient mechanism for control flows. A second aspect is the sensoric system that provides insight into the modus operandi of our multimedia architecture which, as mentioned in the introduction, is the major driving force behind this work. Last but not least, the interfaces for resource reservation mechanisms and for QoS-oriented resource managers of the underlying system platform (CPU, I/O-Buffers, network bandwidth) are subject to future work.

5 Lessons Learned for Distributed Multimedia Platform Design

While the major focus of this paper is the design and implementation of a component-based distributed multimedia application, the driving force behind this work is the need for a flexible and adaptable application scenario for research on resource efficient layered distributed systems architectures. Consequently, while building this application scenario we kept a strong lookout for major areas where the architectural design as well as the performance of the underlying distributed system platform services has a major impact on the application's resource requirements, QoS guarantees, and overall performance. This section summarises five of the most important areas that have been identified.

IPC for streaming media. The common RPC or RMI services provided by operating systems or distributed middleware systems are badly suited for stream like communication paradigms. First, RPC/RMI communication follows an interactive request/reply communication pattern that does not harmonise with data streams. Second, the implementation of distribution-transparent RPC mechanisms require the marshalling of parameters that usually involves copy operations.

Considering the amount of data involved, avoiding copy operations is extremely important. Copy operations on stream buffers require physical memory

resources that reduce the amount of resources available for use by the rest of the system, thus causing higher virtual memory pages misses, increasing disk and bus traffic and polluting the memory cache, effects well known as a source for major performance degradations.

Consequently, distributed multimedia applications require a highly specialised, minimum copy communication paradigm capable of communicating high data volumes in a single direction. On the one hand, in order to minimise copy operations, approaches integrating IO buffer management at data sources and sinks (network, camera and display controllers), communication system buffer management, and application level buffer management have been developed recently [PAM94, BS96, PDZ00]. On the other hand, applications must be able to tailor the buffer management to application-specific needs, e.g. by pushing individual allocation, deallocation or garbage collection strategies into the buffer management subsystem. As an example, when memory resources for buffering an incoming data stream are desperately low, application-specific buffer attributes that characterise the importance of current buffer contents will help to make the right decision which buffers might be thrown away. Buffering systems using attributed buffer contents are yet subject to ongoing research.

Concurrency. Within any multimedia application component there are several independent activities such as stream data processing, handling of CORBA RPCs, or the response to asynchronous events. Many of these activities are causally independent and may run concurrently. As all activities within an application component share the same address space, parallel activities are implemented by threads instead of regular user processes, thus avoiding TLB and memory cache invalidations caused by regular process switches. However, because threads may wait for independent events, real kernel-supported threads are required. Because these events are asynchronous, independent and may also be time critical, fast event notification services are needed.

Event Notification Services. Within any component of our multimedia application different types of mutually asynchronous events must be handled. Buffer managers report changes of stream buffer states and critical high-water situations, configuration managers report the advent or passing of multiplexing components, and feedback from other application components arrives via the CORBA RPC interface and is signalled to the stream processing threads.

Events thus have different sources, are of different types, and may or may not be time-critical. Event notification services such as the BSD *select* system call do not cope very well with these requirements. On the one hand, the original BSD implementation does not scale very well with the number of event sources and is ill-suited for time-critical event delivery [BM98]. A more recent implementation scales better but still consumes a high amount of CPU time because of properties inherent to the semantics of *select* [BMD99]. On the other hand, while many event types in our multimedia application originate within buffer and configuration managers, *select* is defined only for events that are related to

Unix file descriptors.

While scalability and performance of event notification services is currently addressed in approaches such as mentioned above, these approaches still are restricted to file descriptor - related events. Multiple source event notification systems that may handle more general event types and event sources are subject to ongoing work.

Global Resource Management. In general, three classes of resource requirements are characteristic for multimedia streaming applications. Firstly, in order to cope with stream types such as MPEG (approx. 1Mbit/s), DV input via Firewire (30Mbit/s) or HDTV (1Gbit/s) a high data transfer bandwidth is required, both from the network and from the file system. Secondly, interactive, robust and high-quality audio/video applications require low latency and high responsiveness to asynchronous events. Thirdly, QoS parameters such as frame rate, maximum delay and maximum jitter of video streams must be guaranteed in order to met general quality requirements.

The major bottlenecks that determine throughput, latency, responsiveness and QoS properties have been identified to be the amount of data copied in physical main memory (polluting the memory cache), the amount of context switching (invalidating TLB and memory caches), and the latency of the interrupt system [NS95, Sch96, ABD+98]. Considering these bottlenecks, the above mentioned requirements of multimedia streaming applications are strongly related. As an example, a coding algorithm that reduces the amount of data of a stream and thus reduces the need for network bandwidth on the other hand requires additional CPU time and may also – because the coded stream may be less robust with respect to packet losses – require a higher responsiveness to failure conditions.

As a consequence, traditional resource allocation strategies that focus on one type of resource (CPU schedulers, I/O buffer managers) do not perform very well for multimedia streaming applications. In order to cope with groups of resources that must be allocated in a global and interrelationship-aware way, several research activities currently focus on global resource allocation schemes and exploit new abstractions for resource principals such as process groups [LMB+96, BGzS98, VGR98] or resource containers [BMD99].

Knowledge Sharing The actual resource requirements of multimedia applications are hard to predict. The most important factor of influence is the bitrate produced by codecs, which, using variable-bitrate video codecs rises dramatically when scenes change or have rapidly moving objects. Additionally, failure of the transmission media and the resulting costs for adaption are just as hard to estimate as user interactions in highly interactive applications like teleteaching. As the availability of communication resources in wireless networks cannot be guaranteed, dynamic adaptation strategies are gaining importance.

Considering media streaming a closed-loop control system, affected by system performance and variations in bandwidth, delay, loss and reliability of commu-

nication links we need feedback mechanisms to be able to adapt. Feedback messages are also affected by communication delays and loss rates of the same transmission path or others (satellite networks), making the control system sluggish. Especially for interactive applications and applications with large bandwidth/delay products, where much data is on the fly, it is impossible for media sources to react to distant problems in time. Multicast scenarios, where only some receivers are affected by bandwidth degradation or higher loss rates, can't use adaption at media sources at all, because unaffected receivers will suffer from the adaption effects too [BT98]. One solution is to preventively use error resilient compression techniques like forward error correction and layered video coding [RhD99, WSL00]. Such approaches use a fixed adaption strategy and leave the decisions about how and when to adapt to the format-unaware system layer. Problems arise when changing the media codec or some basic settings in response to variations in the environment, which would require to update all adaption strategies along the communication path too. A better way is to place parts of the adaption strategies directly at locations close to the source of the potential problem. This allows faster and more specific responses. Thus intelligent adaption can only be achieved by communicating knowledge about events, states of subsystems and adaption strategies. The many unsolved problems in this field have made the definition and exploitation of a general knowledge sharing paradigm an active research area.

References

[ABD+98] S. Araki, A. Bilas, C Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *Proceedings of the 10th International Conference on High Performance Computing and Communications*, November 1998.

[BGzS98] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclips Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[BM98] Gaurav Banga and Jeffrey C. Mogul. Scalable Kernel Performance for Internet Servers Under realistic Loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 1–12, June 1998.

[BMD99] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for Unix. In *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.

[BS96] J. C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, October 1996.

[BT98] J-C. Bolot and T. Turletti. Experience with control mechanisms for packet video in the internet. *Computer Communication Review*, 1998.

[CT90] David D. Clark and David L. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proceedings of the 1990 Symposium on Communication Architectures and Protocols*, pages 200–208, Philadelphia, September 1990.

[CWWS92] Jon Crowcroft, Ian Wakeman, Zheng Wang, and Dejan Sirovica. Is Layering Harmful ? *IEEE Network*, 6(1):20–24, January 1992.

[HWA97] Jane Hunter, Varuni Witana, and Mark Antoniades. A review of video streaming over the internet. Technical Report TR97-10, Distributed Systems Technology Centre, University of Queensland, Australia, 1997.

[LMB+96] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbanks, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.

[MSS99] S. Mungee, N. Surendran, and D. Schmidt. The design and performance of a corba audio/video streaming service. In *Thirty-second Annual Hawaii International Conference on System Sciences*, 1999.

[NS95] K. Nahrstedt and R. Steinmetz. Resource Management in Networked Multimedia Systems. *Computer*, 28(5):52–63, May 1995.

[PAM94] J. Pasquale, E. Anderson, and P. K. Muller. Container Shipping: Operating System Support for I/O-Intensive Applications. *Computer*, 27(3):84–93, 1994.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[RhD99] R. Rejaie, M. handley, and D.estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. *IEEE Infocom'99*, 1999.

[Sch96] H. Schulzrinne. Operating System Issues for Continuous Media. *Multimedia Systems*, 4(5):269–280, October 1996.

[SS93] D. Schmidt and T. Suda. Transport system architectures for high-performance communications subsystems. *IEEE Journal on Selected Areas in Communication*, 11(4), May 1993.

[Ten89] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, Zurich, Switzerland, 1989.

[VGR98] B. Vergehese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared Memory Multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[WC01] Matt Welsh and David Culler. Virtualization considered harmful: Os design directions for well-conditioned services. In *8th Workshop on Hot Topics in Operating Systems*, pages 122–127, 2001.

[WSL00] Benjamin W. Wah, Xiao Su, and Dong Lin. A survey of error-concealment schemes for real-time audio and video transmissions over the internet. In *IEEE International Symposium on Multimedia Software Engineering*, Dec 2000.

[XP99] George Xylomenos and George C. Polyzos. Internet Protocol Performance over Networks with Wireless Links. *IEEE Network*, 13(4), July 1999.