SUPPORT FOR CONTINUOUS MEDIA IN THE DASH SYSTEM[1]

David P. Anderson
Shin-Yuan Tzou
Robert Wahbe
Ramesh Govindan
Martin Andrews

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

October 16, 1989

ABSTRACT

Future distributed systems will support *continuous media* such as digital audio and video, allowing user programs to convert, process, store, and communicate continuous-media data. The DASH project is developing such a system. Our work consists of two related parts. First, we have defined the *DASH resource model* as a basis for reserving and scheduling resources (disk, CPU, network, etc.) involved in end-to-end handling of continuous-media data. The model uses primitives that express workload characteristics and performance requirements, and defines an algorithm for negotiated reservation of distributed resources. This algorithm is embodied IP+, a backwards-compatible extension of the Internet Protocol (IP).

Second, we have developed a distributed system kernel for use as an experimental testbed. The DASH kernel implements the DASH resource model for scheduling of CPU and network access. Its virtual memory system provides efficient data transfer between address spaces. Finally, its implementation is structured using object-oriented programming and message-passing.

[1] Sponsored by the California MICRO program, AT&T Bell Laboratories, Digital Equipment Corporation, IBM Corporation, Olivetti S.p.A, and the Hitachi Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

SUPPORT FOR CONTINUOUS MEDIA IN THE DASH SYSTEM

David P. Anderson
Shin-Yuan Tzou
Robert Wahbe
Ramesh Govindan
Martin Andrews

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

October 9, 1989

1. INTRODUCTION

Future distributed systems will support *continuous media*[1] such as digital audio and video, allowing user programs to convert, process, store, and communicate continuous-media data. The DASH project is developing such a system. Our work consists of two related parts. First, we have defined the *DASH resource model* as a basis for reserving and scheduling resources (disk, CPU, network, *etc*.) involved in end-to-end handling of continuous-media data. The model uses primitives that express workload characteristics and performance requirements, and defines an algorithm for negotiated reservation of distributed resources. This algorithm is embodied IP+, a backwards-compatible extension of the Internet Protocol (IP).

Second, we have developed a distributed system kernel for use as an experimental testbed. The DASH kernel implements the DASH resource model for scheduling of CPU and network access. Its virtual memory system provides efficient data transfer between address spaces. Finally, its implementation is structured using object-oriented programming and message-

[1] The term "continuous media" is used to emphasize that information flows continuously over real time. We avoid the more common term "multimedia", which is often used to refer to graphics and still images as well.

passing.

1.1. Hardware Technology Trends

The DASH project is motivated by two emerging hardware technologies. First, hardware for the conversion and compression of continuous-media data (digital video and audio) is now available. For examples, Intel's Digital Video Interactive (DVI) hardware [1] can compress full-motion NTSC-quality video to a data rate of 1.2 Mbps. Analogous future technology for HDTV-quality video may produce data rates in the range of 40 to 100 Mbps. Advances in other areas (CPU speed, memory size, parallel disk arrays, *etc.*) will enable all system hardware components to handle continuous media.

Second, wide-area high-speed digital networks are becoming feasible due to the development of optical-fiber transmission and fast packet switching [2]. Such a network will provide high point-to-point throughput (10 to 100 Mbps) with guaranteed performance, and will eventually be as pervasive as the current telephone network, reaching most homes and offices.

The combination of these two trends will expand the use of high-performance workstations into the consumer market. These end-user machines will combine the role of workstation, HDTV receiver, telephone, FAX machine, and high-fidelity sound system. Shared server machines, with massive storage capabilities, will assume the role of television station, CD collection, and public library.

1.2. Future Application Areas

The hardware technologies described above will allow general-purpose computer systems to support distributed interactive applications with continuous-media interfaces. On-line databases of continuous-media information will be created, including video entertainment, music, advertising, and education [3,4]. Interfaces to this data may be interactive and personalized, and may include speech recognition or synthesis.

Most forms of remote human communication (conversations and conferencing), as well as facilities for collaborative work, can be handled in the same general-purpose system. In short, the distributed system of the future will subsume many existing information systems, including television (broadcast and cable), radio, publishing (printed and audio), telephone and FAX, physical mail. The personal workstation will serve as a universal point of interaction with these unified media.

Along with support for these new applications, the future system will ideally provide the functions of current distributed systems. Users can run programs, concurrently and in protected address spaces, on both workstations and server machines. These programs can offer globally-accessible services, and can access such services. In other words, the system will conform to the "open system" model of current distributed systems, rather than the customer/provider model of telephone companies.

1.3. System Software Structure

An operating system kernel for our hypothetical "future system" allows user-level processes to directly handle continuous-media traffic. In the same way that UNIX processes handle byte streams, processes can input, output, store, process, and communicate real-time streams of continuous-media data. With this capability, a flexible client/server architecture for continuous media, analogous to current window-server architectures such as X and NeWS, will be possible. We call this the Media Server (see Figure 1). A client of the Media Server can send it a stream of compressed video data and have it displayed in a window. The client can concurrently send a stream of audio data (perhaps from a different source) and have it played synchronously with the video data. The client, if it wishes, can redirect the audio data through a user-level process to subject it to filtering or other signal processing.

Several systems involving continuous media have been implemented, including Etherphone [5], IMAL [6], DVI [1], and VOX [7]. However, existing general-purpose operating systems lack

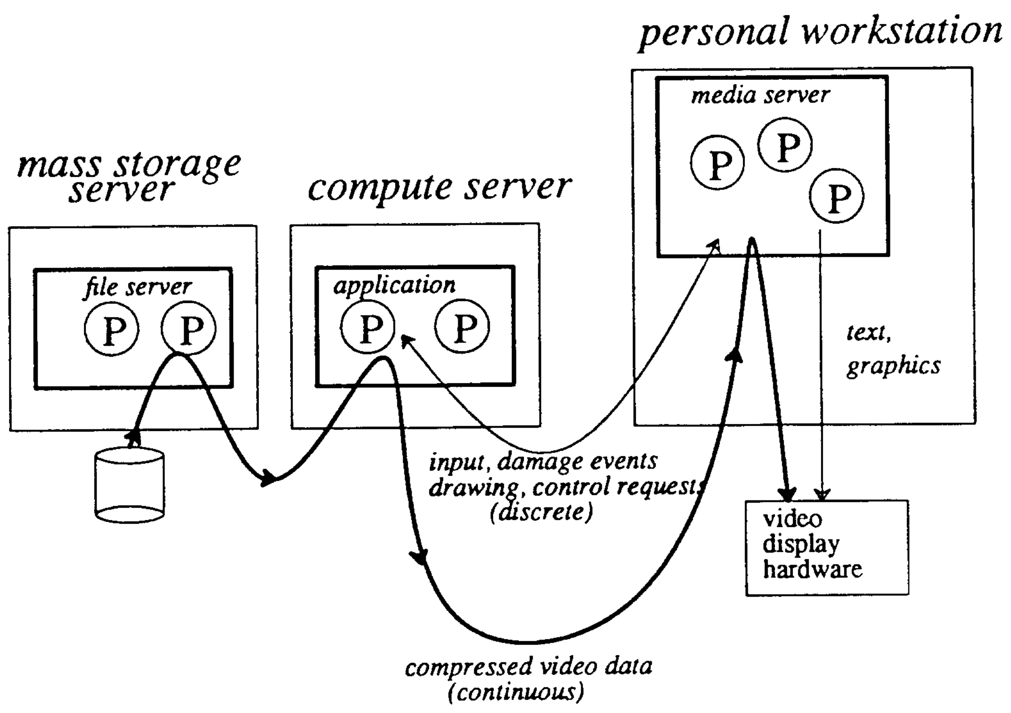

Figure 1. Software and communication structure for integrated continuous media.

the real-time capabilities needed to handle continuous media data. As a result, the above projects handle continuous-media data outside user programs (sometimes in separate analog networks) or are restricted to single-program, single-user applications. DASH is trying to remove these limitations, and to provide a flexible and uniform platform for continuous-media applications.

1.4. The DASH Kernel: Goals and Features

The DASH kernel was developed as a testbed for experiments in system design, so we wanted it to be well-structured and easily modifiable. It was not designed for production use, for timesharing, or to support mass storage. Had these been goals, we would simply have modified an existing system. Instead, we built a kernel from scratch. The DASH kernel is implemented in the object-oriented language C++. It is about 30,000 lines long, of which about 10,000 are comments. The kernel implements protocols (TCP, UDP, and NFS) that allow it to interoperate with existing operating systems. It currently runs on the Sun 3/50, and is being ported to IBM PS/2 workstations with DVI hardware.

The DASH kernel has several properties that we feel are crucial for supporting continuous-media I/O. The kernel uses preemptive deadline-based process scheduling, and is based on a real-time message-passing model. It is capable of moving data between user processes, and between I/O devices, at predictable rates and with minimal overhead.

The role of network file access in DASH is different than in most distributed systems. Programs running on the DASH kernel normally access files in one of two ways: reading or writing a file of continuous-media data, or executing a program. Hence we are not concerned with issues such as recovery, client caching of read/write files, and so on.

2. CONTINUOUS MEDIA: REQUIREMENTS AND PROBLEMS

In this section we discuss the performance requirements of applications that use continuous media, and the limitations of existing hardware, network protocols, and operating systems in fulfilling these requirements.

2.1. Performance Requirements for Continuous Media

Figure 1 depicts a continuous-media application. A stream of compressed digital video is stored in a file server. An application, executing on a compute server, reads the video data and forwards it to the ''media server'' running on the user's personal workstation. (Alternatively, the video stream might originate from a camera attached to the workstation of a second user, as part of a video conference.) The user can perhaps start and stop the video by clicking mouse buttons.

Such applications have stringent performance requirements. First, real-time transfer of continuous media data has a minimum throughput requirement. High-quality audio requires about 1.4 Mbps, compressed DVI video about 1.2 Mbps, and compressed HDTV about 40 Mbps.

Second, interactive applications have maximum delay constraints. For conversations, delays of about 50 milliseconds are acceptable. For highly-synchronized activities such as distributed musical rehearsal, the delay limit may be as low as 10 milliseconds.

We believe that a system for continuous-media applications should provide the following features:

- An application should be able to express its performance requirements and its workload to the operating system, and to "reserve" the resources necessary to ensure that level of performance. The system can turn down the request if sufficient resources are not available.
- If the system grants a reservation, it should allocate and schedule its resources so that the requirements are met in all cases except for a hardware or software failure. Concurrent activity (network traffic, other programs running on the hosts, *etc.*) should not cause the guarantees to be violated.

In the next section, we argue that current distributed systems, even "real-time" systems, do not have these properties.

2.2. Shortcomings of Existing Systems

Application performance requirements are end-to-end. In the example of Figure 1, the video data is handled by a sequence of hardware devices: the file server's disk, memory, I/O bus and CPU, the network and possibly intervening gateways, the compute server hardware, another network, and finally the workstation and its display device. Throughput and delay are a function of all the devices. Each device is shared with other hosts or with other applications running on the given hosts. When contention occurs, data is queued and therefore delayed. A *scheduling policy* decides the order in which queued requests are processed.

We classify scheduling policies as *careful* if they are based on application requirements and provide some form of throughput and delay guarantees, and *carefree* otherwise. (Carefree policies may be designed to satisfy some other criteria, such as fairness or aggregate throughput, but this does not concern us here.) Examining the various layers of current general-purpose networks and distributed systems, we see that carefree policies dominate:

- Ethernet-type network media access protocols are nondeterministic and therefore carefree. Token ring networks provide an inflexible per-host guarantee based on maximum token rotation time. FDDI-II and BISDN networks will offer more useful guarantees.
- Internetwork protocols, such as the DARPA Internet Protocol (IP), typically offer no guarantees. Packet queueing in hosts and gateways is carefree.
- Reliable request/reply IPC used for remote procedure call (RPC) and remote object invocation, is prevalent in modern distributed systems. This IPC style is ill-suited to continuous-media data, especially over wide-area networks, because 1) the flow of continuous-media data is inherently asynchronous, and 2) data must be pipelined to achieve adequate performance. Reliable stream-oriented protocols such as TCP are also non-optimal, because retransmission may conflict with real-time performance and reliability is often not needed for continuous media.
- General-purpose operating systems such as UNIX use carefree CPU scheduling policies. Furthermore, such systems have ''hidden'' scheduling delays due to non-preemption of processes in the kernel, sequential processing of software interrupts, *etc*. Disk-head scheduling in most existing file systems is carefree. Conventional real-time systems often use careful scheduling, but usually only for delay bounds and not throughput guarantees. In addition, these systems are typically concerned with fault-tolerance, which is not a major concern for continuous media.

Is careful scheduling necessary? In the distant future, GigaFLOP CPUs and Terabit networks may provide acceptable performance regardless of scheduling policies. However, we conjecture that, in the foreseeable future, hardware resources will suffice for the requirements of continuous-media applications, but only if they are scheduled carefully. We call this condition the "window of scarcity" (see Figure 2). Carefree scheduling, even within the window of scarcity, will work correctly in a limited range of load conditions, *i.e.*, if the hardware is sufficiently fast and there is little concurrent usage. Outside this range, the application will experience unpredictable timing errors.

In the following two sections we describe two ways in which the DASH system supports continuous media: by careful end-to-end resource scheduling, and by a VM mechanism for efficient data movement within a host.

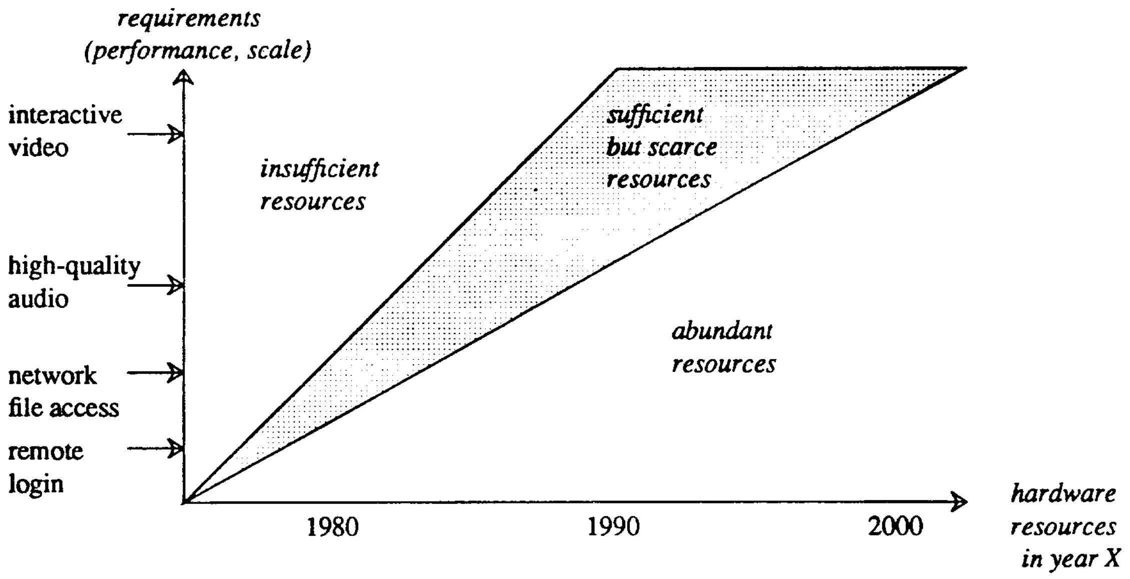

Figure 2. The window of scarcity. In the shaded region, hardware resources are sufficient to handle the performance requirements of applications, but only if they are allocated and scheduled in accordance with those requirements.

3. RESOURCE SCHEDULING FOR CONTINUOUS MEDIA

As a basis for careful end-to-end resource scheduling, we have developed the *DASH resource model.* The components of the model are the following. A *resource* is a device that stores, manipulates, or communicates continuous-media data. Resources may be reserved in *sessions.* Sessions of resources may be combined into *end-to-end sessions.* In this section, we define these components in greater detail and discuss their implementation. We then describe an extension to the DARPA Internet protocol family that incorporates the DASH resource model.

3.1. Modeling the Flow of Continuous-Media Data

Before we can discuss the parameterization of end-to-end delay and throughput, we need a description of the "message arrival process" at a particular interface in the system. The DASH resource model uses an abstraction (proposed by Rene Cruz [8]) called a *linear bounded arrival process (LBAP).* An LBAP has the following parameters:

S = *maximum message size*	*(bytes)*
R = *maximum message rate*	*(messages/second)*
B = *maximum burst size*	*(messages)*

In any time interval of length t, the number of messages arriving at the interface may not exceed

$$B + Rt$$

The long-term data rate of the LBAP is SR bytes per second. The burst parameter B allows short-term violations of this rate constraint, modeling programs and devices that generate "bursts" of messages that would otherwise exceed the rate constraint.

Prior to analyzing delay, it is convenient to define a function $b(m)$ representing the logical "backlog" of the arrival process. This is the number of messages by which the arrival process is "ahead of schedule" (relative to its long-term rate) when message m arrives, and is not necessarily the number of queued messages. $b(t)$ is defined by

$$b(m_0) = 0$$

$$b(m_i) = max(0,\ b(m_{i-1}) - R(t_i - t_{i-1}) + 1)$$

where t_i is the arrival time of message m_i. Using *b(m)*, we define the *logical arrival time, l(m)*, of a message *m* as

$$l(m_i) = t_i + \frac{b(m_i)}{R}$$

Intuitively, *l(m)* is the time *m* would have arrived if the LBAP strictly obeyed its maximum message rate. This is used in computing end-to-end delay (see Section 3.5). Figure 3 shows the relationship between backlog and logical arrival time.

3.2. Modeling Resources

The DASH resource model treats hardware devices (CPUs, disks, networks, *etc.*) in a uniform way. Devices that handle continuous-media data are required to be scheduled in a way that is compatible with this model. The flow of data into and out of a device is modeled as a set of

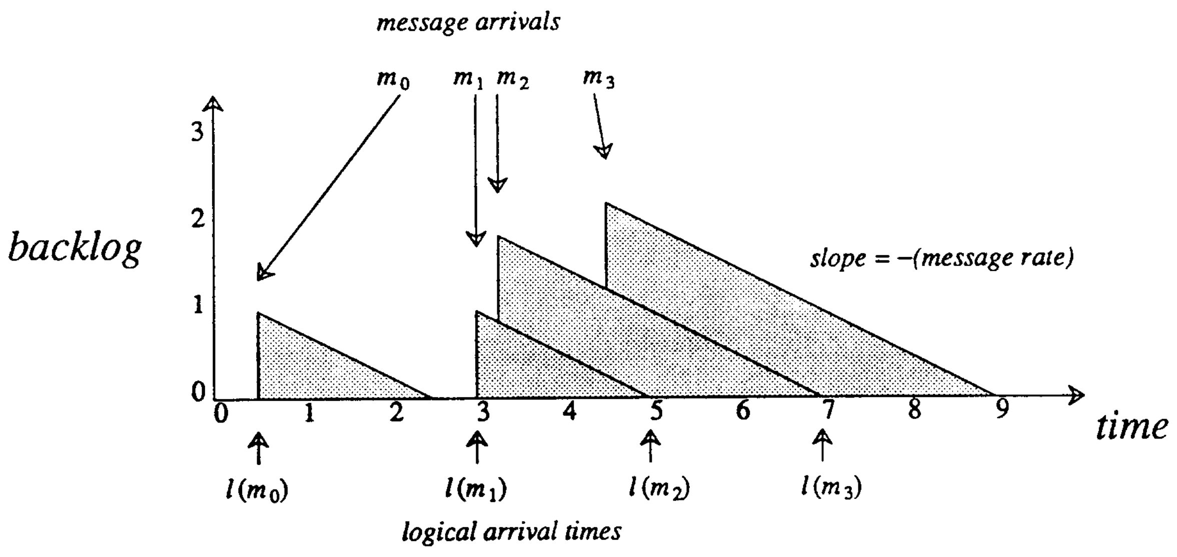

Figure 3. The backlog function *b(m)* shown as a function of time. A message's logical arrival time is based on the backlog at its actual arrival time.

LBAPs. Each device in the system has a *resource object* that serves as its manager. A resource may act as a *source*, a *sink*, or a *handler*. A source produces LBAPs and a sink consumes LBAPs. Resources such as disks may act as either a sink or source. A handler accepts LBAPs, producing output LBAPs. We assume that handlers do not modify the message sequence, so the incoming and outgoing LBAP for handler resources must have the same values for S and R.

The *actual delay* of a message m in a handler resource is the time interval between its arrival at the input interface and its arrival at the output interface. The *logical delay* of m is the interval between the m's logical arrival time and its arrival at the output interface. In other words, logical delay is the actual delay minus the amount by which the message arrives "ahead of schedule". Logical delay, rather than actual delay, determines end-to-end delay bounds (see Section 3.5).

Clients may request *sessions* with resource objects. Each session has associated sets of LBAP parameters for its input and/or output interfaces. Handler sessions also have 1) a *maximum logical delay* for messages in the resource; 2) a *minimum actual delay* for messages in the resource; 3) a *minimum unbuffered delay*, the portion of actual delay during which the message is not stored in host memory. For network resources, this includes the network propagation time, which may be significant in wide-area networks. Parameters 2) and 3) are used to calculate buffer space needs.

The *class* of a session may be either *guaranteed* or *best-effort*. For guaranteed sessions, a resource reservation is made, and the delay parameters hold unless a failure occurs. For best-effort sessions, no reservation is made; the delay parameters are "hints" to the resource, and may be exceeded (or messages may be dropped) if the resource becomes loaded.

3.3. Interface to Resource Objects

We now describe a "generic interface" to resource object. This interface is used by an end-to-end scheme, described in Section 3.5, in which resources are first reserved and then relaxed. The interfaces of actual resource objects are variants of this generic interface.

reserve() *(create a new session with minimum possible delay and outgoing burst size)*
input parameters:
- maximum message size
- maximum message rate
- incoming burst size
- class (guaranteed or best-effort)

output parameters:
- session ID
- maximum logical delay
- outgoing burst size
- minimum delay
- minimum unbuffered delay

relax() *(relax the parameters of an existing session)*
input parameters:
- session ID
- new maximum outgoing burst size
- new maximum logical delay

output parameters:
- new incoming burst size
- actual outgoing burst size

free() *(cancel an existing session)*
input parameters:
- session ID

output parameters:
- none

Actual interfaces vary according to the particular device. For example, the `reserve()` operation on a network resource would include a destination address, and the `reserve()` operation on a CPU resource would include CPU time per message instead of message size. The mechanism by which messages pass between resources in unspecified; the interface might use interrupts, message-passing, or function calls. Source resources typically provide `start(ID)` and `stop(ID)` operations. Once started, the resource produces an output message stream based on the session parameters.

3.4. Buffer Management

In the DASH resource model, messages are not normally lost. For a given resource, this requires reserving enough buffer space to accommodate the input burst size plus messages being processed in the host:

buffer space = input burst size + (message rate)(maximum buffered delay)

where

maximum buffered delay = maximum logical delay – minimum unbuffered delay

When several sessions on a given host are chained (*i.e.*, messages traverse the resources in sequence) a formula similar to the above gives a tighter bound on the number of buffers needed for the entire chain of sessions. The input burst size is that of the first session in the chain, and the maximum buffered delay is summed over the sessions.

A second buffer allocation problem arises when the receiving end of an application must deliver messages at a constant rate to the output device (*e.g.* audio or video converters). Suppose the first message of a stream arrives with minimum delay. If the application outputs the first message immediately, and the second arrives with maximum delay, there will be a pause in the output between the two. The application must therefore buffer messages to ensure that there are no pauses in the output. For each data path, the *minimum overall delay* is defined as the sum of minimum actual delays in the path, and the *maximum overall delay* as the sum of the maximum logical delays. Assume that the source resource generates messages fast enough to maintain a nonzero backlog. If all messages are delayed in the receiving application so that the total delay is at least the maximum overall delay, there will be no pauses on the output. The amount of buffer space needed for this purpose is

(maximum overall delay – minimum overall delay)(message rate)

3.5. End-to-End Sessions

In a continuous-media application, data passes through a sequence of resources, perhaps located on different hosts. Sessions reserved with these resources can be combined to form an *end-to-end session*. The DASH resource model defines an establishment protocol for end-to-end sessions. This protocol involves negotiation between the resources; the application's allowable end-to-end delay is divided between the resources. It is important to note that maximum end-to-end delay is the sum of maximum *logical* delays in the resources; the amount by which a resource works "ahead of schedule" is not counted as delay in the next resource.

Roughly speaking, the establishment protocol works as follows. First, maximum-performance sessions (*i.e.*, sessions with the smallest possible delay and burst size) are reserved at each resource. Second, the excess delay, if any, is distributed among the resources, and burst size limits are relaxed if possible.

The establishment protocol is carried out by *host resource managers*, each of which is responsible for the resources within its host[2]. Initially, the host resource manager at the source host is given a client request that specifies the resources involved, the message size and rate, and the end-to-end delay requirements (a *target* and *maximum* value, denoted T and M). The protocol has two phases. The first phase traverses the hosts from the source to the sink, and the second phase proceeds in the reverse direction. In the first phase, a *request* message, with the following contents, is relayed between host resource managers.

end-to-end session unique ID
message size and rate
target and maximum end-to-end delay
number of resource traversed
output burst size from the previous host
cumulative sum of maximum logical delays

[2] In a real-world implementation, the task of the host resource manager may be divided between several agents.

In response to a request message, a host resource manager does the following:

(1) The `reserve()` operation is done on each local resource in sequence (towards the sink). The parameters to the operation include the description of the incoming LBAP. If the resulting cumulative logical delay exceeds M, a *failure* message is propagated back to the sender, and the host resource managers `free()` all sessions.

(2) Based on the incoming burst size and the buffer delays in local resources, the host resource manager reserves buffer space on the host as described in Section 3.4. If this allocation fails, the request is rejected as above.

(3) The host resource manager determines the next host on the path, and relays the request message to it, with the last three fields appropriately updated.

If the request has not been rejected at the end of the first phase, the cumulative logical delay C is less than M. If in addition $C < T$, the difference $T - C$ is called the *excess delay*. In the second phase, the excess delay is divided among the resources in the path[3]. A *reply* message of the following form is relayed between host resource managers:

end-to-end session unique ID
remaining excess delay
acceptable burst size into next host

In response to a reply message, a host resource manager does the following:

(1) The `relax()` operation is done on each local resource in sequence (towards the source). Extra buffer space is allocated to accommodate increase delays. The manager may allocate additional buffer space to accommodate a larger incoming burst size.

(2) The reply message, with the last two fields updated, is passed to the next host towards the source.

[3] For simplicity, we currently divide excess delay uniformly among resources. It is likely that a more sophisticated policy would be preferable.

When the manager at the sending host is done, it notifies the sending client, which can then have the source resource start sending data over the end-to-end session. The receiving client may have to regulate the data flow as described in Section 3.4.

3.6. Resource Implementation

We now discuss the implementation of resources, *i.e.*, how to design schedulers for disks, CPUs and networks that provide the interface described above. Our discussion uses an abstract model of a resource (see Figure 4). In this model, a resource contains several components. The *manager* implements the `reserve()`, `relax()` and `free()` operations. Each session may have an associated *regulator* to limit its outgoing burst size. The *scheduler* determines the order in which messages are handled by the hardware *device*.

A variety of scheduling policies can be used; the main criterion is the ability to bound logical delay. Analyses for round-robin, FIFO, and non-preemptive deadline scheduling are given in [9]. For round-robin and FIFO scheduling, all sessions inherently have the same delay bound.

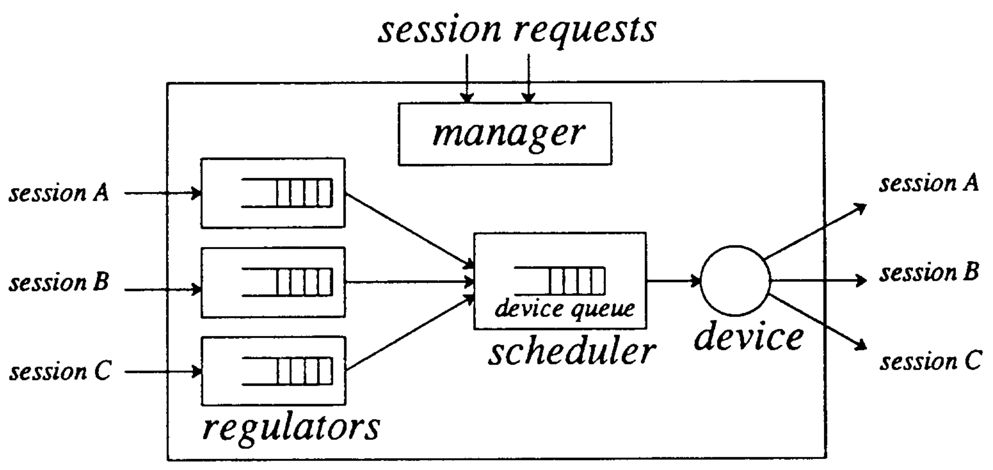

Figure 4. A model of the internal structure of a resource.

Hence the resource cannot accommodate a wide range of requirements.

With deadline scheduling, every message is assigned a deadline of its logical arrival time plus the maximum delay associated with its session. Messages are processed in order of increasing deadline. Deadline scheduling is an attractive policy because 1) it allows a range of maximum delays and 2) it is optimal in the sense that if any scheduling policy can safely accommodate a set of sessions, then deadline scheduling does so [10]. We have implemented a `reserve()` operation for deadline scheduling as a simulation of the resource under worst case load. The algorithm and its performance are described in [9].

Schedulers may be designed to handle best-effort as well as guaranteed sessions, and perhaps to handle non-real-time traffic as well. The same choice of scheduling policies is available for best-effort traffic. For simplicity, guaranteed traffic can be given strict priority over best-effort traffic. With additional scheduler complexity, it may be possible to schedule best-effort messages ahead of guaranteed messages in some cases. Non-real-time traffic can be given lowest priority, perhaps with an mechanism for starvation avoidance.

Resources such as networks may use regulators to limit their outgoing burst size. In our resource implementation model, regulators strictly precede the scheduler, and they can also be used for enforcing the incoming burst size. When a message is received, the regulator computes the current backlog for the session (see Section 3.1). If the backlog exceeds the maximum incoming burst size, the regulator may block the client or drop the message. After accepting a message, the regulator may delay the message before giving it to the scheduler. The outgoing burst size of the resource is given by

(burst size into scheduler) + (maximum actual delay in scheduler) (message rate)

This expression determines the allowable burst size between the regulator and the scheduler.

3.7. The CPU as a Resource

The DASH kernel supports programs (applications, servers and protocols) that process streams of continuous-media messages, using a constant amount of CPU time per message. CPU scheduling is based on the DASH resource model. The CPU scheduler provides a resource object interface, allowing clients to reserve sessions with guaranteed delay and throughput (CPU time per real-time second). Process scheduling is based on *deadlines*, *i.e.*, the real time by which a process must finish handling its current message. A process must call the scheduler to change its deadline before handling the next message (this is done automatically by the message-passing system; see Section 5).

The scheduler maintains two deadline-sorted queues: one for processes handling messages on guaranteed sessions, the other for best-effort sessions. The policy is simple: the guaranteed process with the earliest deadline is executed. If there is none, the earliest best-effort process is executed. If there is no best-effort process, non-real-time processes are executed round-robin with time-slicing.

Process scheduling is *preemptive*; a process *A* can be interrupted and descheduled in favor of a process *B* with an earlier deadline. This preemption can occur even if *A* is executing in the kernel (in contrast, most UNIX systems disallow preemption in the kernel). Kernel preemption requires short-term locking (by interrupt-masking or spin locks) of shared data structures. However, it yields the benefit not only of improved real-time response, but also the possibility of concurrent kernel execution on a shared-memory multiprocessor.

3.8. Discussion

We now explain some of the design decisions in the DASH resource model. The most fundamental decision was the choice of the linear bounded arrival process (LBAP) to model data traffic. A variety of other models, both statistical and deterministic, could be used instead. These models can represent a broader range of traffic properties, such as long-term burstiness. LBAP, however, has the advantage that it is straightforward to guarantee delay bounds for a variety of scheduling disciplines. Many continuous-media forms, including DVI [1], use a constant-rate data flow and fit the LBAP model well. The LBAP model also allows for devices or applications that produce data in bursts. If these bursts eventually exceed the session's burst size, it is possible to provide flow control so that the application is temporarily slowed down to the session's long-term rate.

The end-to-end session establishment protocol is conservative; it reserves minimum-delay sessions for the duration of the first phase. If a request *A* arrives at a resource that is involved in the first phase of another request, *A* may be rejected needlessly. The protocol could be modified so that a new request is blocked until pending requests are completed (and their reservations have been relaxed). The establishment algorithm assumes that there is a fixed route between source and sink, and that the session's messages will traverse this route. This assumption may not hold in networks with dynamic routing. Finally, the model in its present form does not address multicast. Because continuous media will often be used for conferencing, multicast is an important issue and we plan to extend the model to include it.

3.9. Adding the DASH Resource Model to TCP/IP

The DASH resource model defines a framework for stream-oriented communication with end-to-end performance, but does not dictate a particular network architecture or transport protocol. The model can be implemented as part of a family of specialized protocols for continuous media, or as an extension of a standard protocol hierarchy. To facilitate interoperation with existing systems, we taken the latter approach, and have implemented the model as an extension of the DARPA Internet TCP/IP protocol suite.

We defined a protocol called IP+ for establishing sessions in networks. IP+ cooperates with the client program (both sending and receiving ends) to perform end-to-end session establishment. Each session is associated with an instance of an upper-level protocol (*e.g.*, a TCP connection or a Sun NFS file handle). The scenario for a successful session establishment shows how IP+ works (see Figure 5):

(1) The sending and receiving clients agree on end-to-end session parameters (see Section 3.5) and set up a transport-level connection.

(2) The sending client reserves local resources other than CPU, computes its CPU requirements and those of the transport protocol, and passes these in a session request call to the

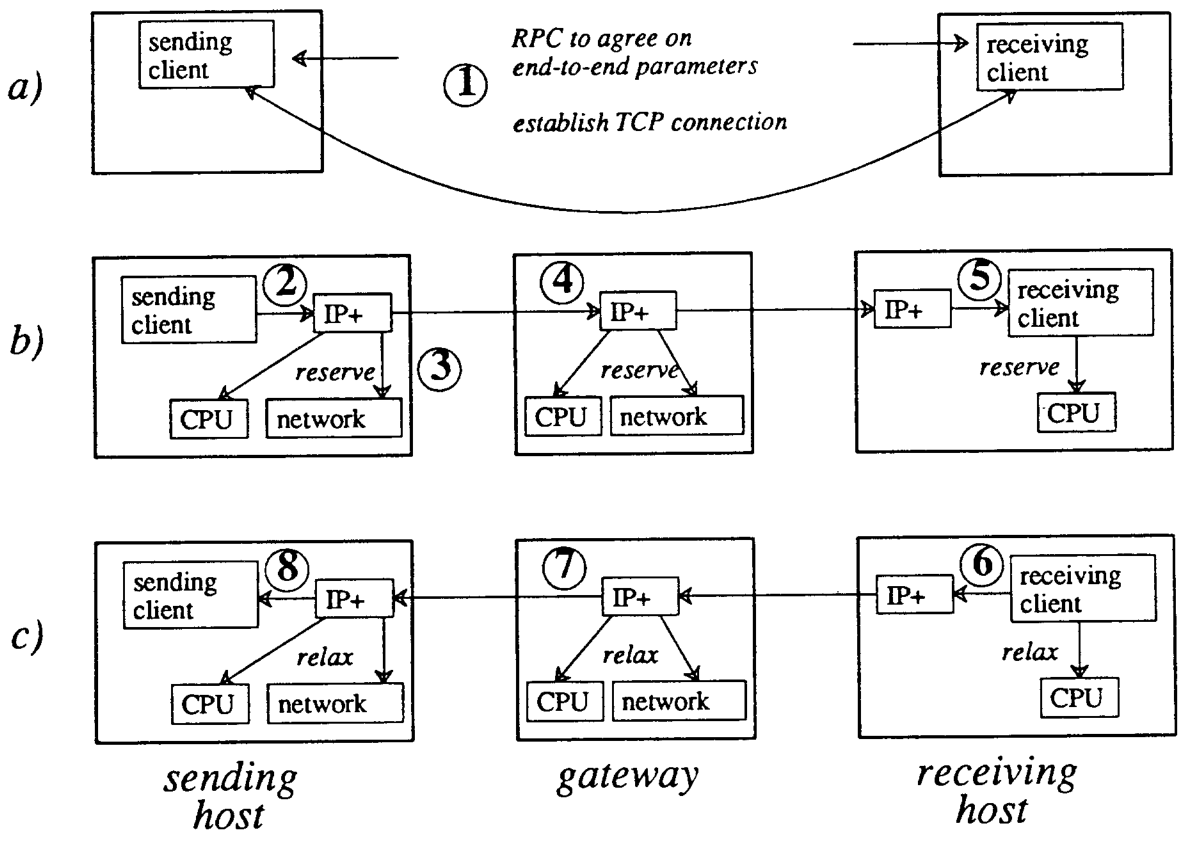

Figure 5. Establishing an end-to-end session using IP+.
In a), the clients create a transport-level connection. The first pass of the IP+ session establishment protocol is done in b), and the second pass in c).

local IP+ module.

(3) The IP+ module on the sending host reserves CPU and network resources, allocates buffer space, and sends a session request message to the next hop (gateway or receiver). (IP+ assumes that routing is static for the duration of a session.) An RPC protocol is used for exchanges between IP+ modules.

(4) A sequence of IP+ modules in gateways carry out the first pass of the end-to-end session establishment algorithm described in Section 3.5, reserving CPU and network resources.

(5) The IP+ module at the receiving host notifies the receiving client, which reserves the remaining resources and buffer space.

(6) The receiving client calculates excess delay, relaxes the reservations of local resources, and replies to the IP+ module.

(7) IP+ does the second pass of the establishment algorithm, relaxing reservations and adjusting burst sizes.

(8) The IP+ module at the sending host returns to the sending client. The client relaxes its local resources, completing the end-to-end session.

An IP+ session establishment can fail due to inadequate resources, in which case an error code is returned to the sending client. It is also possible for a *partial session* to be established. This occurs if some of the participants (gateways or hosts) do not implement IP+. In this case the IP+ session is established among a subset of the participants. The sending client is notified, and may decide whether to continue the session. No performance guarantees can be made for a partial session, but those sites that are involved in the session can still schedule their resources carefully. In some cases (*e.g.*, if the performance bottlenecks are within the sites involved in the session) this may be adequate.

Because partial sessions are allowed, IP+ is backwards compatible with TCP/IP in the sense that applications using IP+ will function, albeit perhaps with inadequate performance, if they communicate with (or through) hosts that do not implement IP+. In addition, applications that use only TCP/IP need not be modified for IP+.

4. HIGH-THROUGHPUT LOCAL INTERPROCESS COMMUNICATION

One goal of the DASH kernel is to allow programs running in protected user-level virtual address spaces (VASs) to directly handle continuous-media data. Furthermore, servers such as file servers and user-interface servers may run as user processes, and will also handle continuous-media data. In some cases the data traverses several user VASs on a single machine. Therefore the kernel must be able to move data efficiently between the kernel VAS and a user VAS, or between two user VASs.

We define the *local IPC throughput* as the aggregate rate at which data is moved between VASs. This will in general be greater than end-to-end application throughput. For example, if a user-level program reads data from the network and moves it to an output device, the data moves between VASs twice (kernel/user and user/kernel). the local IPC bandwidth must be twice that of the actual data rate. If the data traverses more than one user VAS (*e.g.*, if the application and media server run on the same host) then even more VAS crossings occur. In addition, an application may use several concurrent continuous-media channels, and a user might run several concurrent applications. Therefore, if the per-channel throughput requirements are in the range of 1 to 10 Mbps (see Section 1.1), the local IPC throughput requirement may be in the range of 10 to 100 Mbps or more.

Systems such as UNIX use software copying to transfer data between VASs. In transfers between two user VASs, data may be copied twice: from the sender's VAS to the kernel VAS and once from the kernel to the receiver VAS. This approach is undesirable for high-bandwidth data, because copying large amounts of data is both CPU- and bus-intensive. Therefore copying slows down other computations, and slows down DMA transfers of I/O devices.

Some systems avoid software copying by allowing VASs to share a region of common memory for data transfer. We rejected this approach because of security considerations: for example, a user program running on a compute server should not be able to intercept sensitive material from other users' audio connections.

4.1. Virtual Memory Remapping

DASH integrates virtual memory and message-passing, using *remapping* to transfer large messages between VASs without physically copying the messages. A message is represented by a *header* that may contains pointers to *data pages* (for short messages, the header itself contains the data). When a message is sent from one VAS to another, the message header is copied, while the associated data pages are remapped. Figure 6 illustrates the changes of memory maps before a `send()` operation and after the corresponding `receive()` operation.

Other systems, (*e.g.*, Accent and Mach) have also integrated virtual memory and message-passing. DASH refines the use of remapping to emphasize efficiency. Remapping involves other overhead besides updating page table entries. The operating system must manage buffers, update pointers to remapped pages, manipulate kernel data structures, and handle possible sharing. At a high remapping rate, this overhead will be accentuated, reducing or eliminating the advantage of remapping over copying.

DASH reduces remapping overhead by restricting and simplifying the remapping mechanism. First, only virtual pages in a special memory region, the *IPC region*, can be remapped, decoupling the remapping mechanism from other functions of the virtual memory system. Second, a virtual page in the IPC region (an *IPC page*) can only be remapped to the same virtual address in another VAS, (not to an arbitrary virtual address). Hence pointers in a message header remain unchanged when a message is transferred. Finally, IPC pages are allocated by the kernel; the allocation policy ensures that a given IPC page is used by only one VAS at a time. An IPC page that is allocated to the sending VAS is free in the receiving VAS, so no buffer allocation is

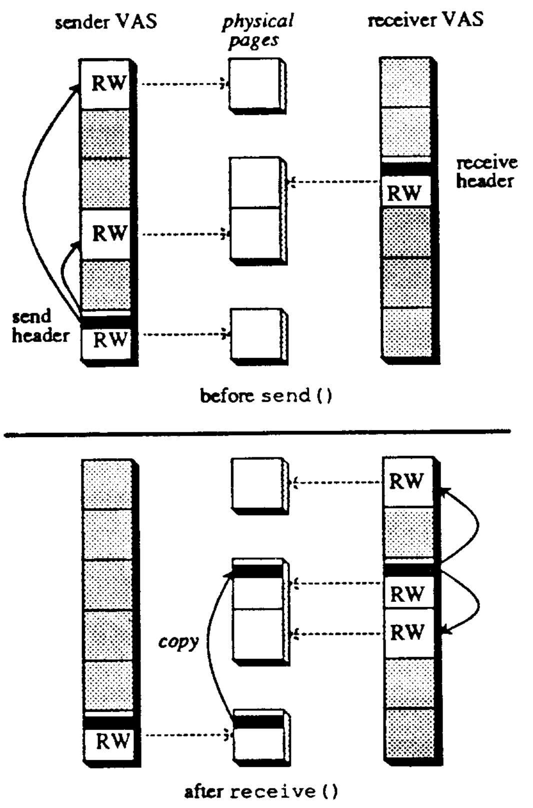

Figure 6. Transferring a message between VASs. Only the message header is copied. Data pages are remapped.

necessary when messages are sent.

The DASH VM system uses *lazy evaluation*, deferring memory map changes when possible. The representation of a VAS's memory map is divided into two parts. The *high-level* memory map is independent of hardware architecture, and is easy to update; we always update it on remapping. The *low-level* memory map, on the other hand, depends on hardware architecture, and may be expensive to update (*e.g.*, on multiprocessor machines with replicated TLBs). A page is mapped into the low-level memory map of a VAS on demand by the page fault handler. Lazy evaluation saves a pair of low-level map and unmap operations if a page is mapped into and out of a VAS without being accessed, but incurs the extra overhead of a page fault if the page is accessed. If a receiver knows in advance that it will access the data pages of a message, it can turn off lazy evaluation to avoid extra page faults. This is done by setting an *immediate access* flag in the `receive()` operation.

4.2. Measurements of Local IPC Throughput in DASH

Figure 7 shows local IPC throughput measurements on a Sun 3/50 workstation running the DASH kernel. The IPC throughput increases with message size, because the weight of the fixed per-message overhead reduces when the number of pages in a message increases. The two horizontal lines represent the bandwidth of pure software copying (per-message overhead is excluded). Clearly, DASH has successfully improved IPC throughput using virtual memory remapping, even for messages containing a single page (8KB). The actual IPC throughput depends on whether lazy evaluation is enabled, and whether data pages are accessed. The highest throughput is achieved when data pages are mapped on demand but are not accessed. When data pages are accessed, as expected, it is more efficient to map data pages on the `receive()` operation than on demand.

5. THE STRUCTURE OF THE DASH KERNEL

We have sketched the functions of the DASH kernel, especially those relating to continuous media. The DASH kernel has other properties that increase its value as an experimental testbed. In particular, it uses two complementary structuring techniques: message-passing (for dynamic structure) and object-orientation (for static structure). The structure of the DASH kernel is described in detail elsewhere [11].

The kernel provides multiple user-level virtual address spaces (VASs). A VAS is a unit of protection and resource allocation. There is one kernel VAS and multiple protected user VASs. Each VAS can be populated by any number of *processes*, which have distinct kernel context

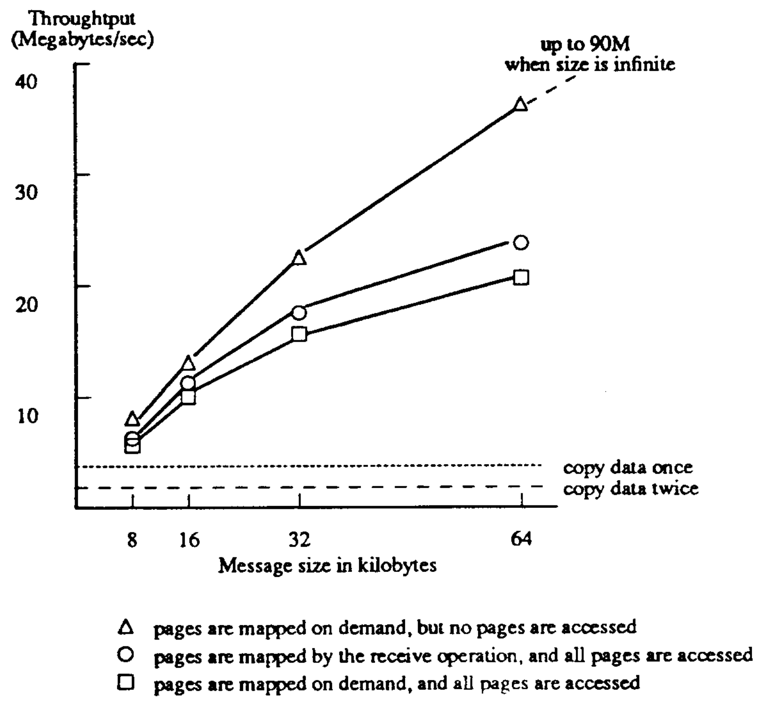

Figure 7. Local IPC throughput as a function of message size.

blocks and can execute in parallel on a multiprocessor.

The DASH kernel, written in the C++ language, is object-oriented. We have found that, compared to other systems, this approach makes the DASH kernel more amenable to development, experimentation and maintenance. DASH is not an "object-oriented system" in the sense of supporting user-defined objects or operations on remote objects. The object paradigm is used only to structure the kernel implementation and user access to local kernel resources. Each VAS has an associated set of *user object references*, small integers that act as capabilities to kernel objects. Processes use them to specify the target of a message-passing operation, or the parameters of a system call; the results of a system call may include user object references.

5.1. Message-Passing Kernel Structure

The dynamic structure of the DASH kernel is based on message-passing. Message-passing is used for interaction between I/O interrupt handlers and processes, for sleep-lock synchronization, and for organizing background processes doing zero-filling and VM page-out. Two basic message-passing "modes" are available: stream and request/reply. Using abstract inheritance, different variants of each mode are available. Some variants are *uniprocess* in the sense that a message is handled by the process that sends it, rather than by a second process.

Message-passing it is integrated with process scheduling (see Section 3.7). A message can carry a class (*guaranteed* or *best-effort*) and a real-time deadline. These parameters are used to schedule any process the handles the message. This scheme is designed for continuous-media data that traverses several processes: real-time requirements are associated with data instead of code or processes.

Finally, user-level processes interact with the kernel (and with processes in other VASs) exclusively by message-passing. System calls, exceptions, and requests to user-level servers are all implemented as message-passing operations. The message-passing system is integrated with the virtual memory system to support high-throughput local IPC (see Section 4.1).

6. CONCLUSION

The DASH project has taken several steps towards supporting continuous media (digital audio and video) in general-purpose distributed operating systems. The DASH resource model provides a uniform basis for reserving and scheduling resources in both networks and hosts. It provides the end-to-end delay and throughput guarantees needed by continuous-media applications. Using the IP+ protocol, the model can be incrementally added to standard operating systems such as UNIX.

The DASH kernel demonstrates the feasibility of using the DASH resource model for CPU and network access scheduling. Its virtual memory system contains a novel mechanism for efficient secure data transfer between virtual address spaces. Performance measurements show that remapping performs significantly better than copying for message sizes of interest. The use of message-passing in the DASH kernel provides improved software structure, and allows real-time deadlines to be propagated with data. Object-oriented programming is used for code reuse and to allow multiple implementations of an interface.

The work described in this paper is only a first step towards usable distributed continuous-media systems. We plan to develop transport protocols and file systems based on the DASH resource model. Ultimately, the resource model should be integrated with a mature OS, such as UNIX or one of its derivatives, that offers a complete user environment. At higher levels, a ''media server'' incorporating continuous media is needed; VOX presents a useful model [7]. The user interface ''look and feel'' must be extended to include continuous media. Finally, application programs must be developed that use and exploit continuous media interfaces.

7. ACKNOWLEDGEMENTS

We would also like to acknowledge the contributions of others who have taken part in the DASH project. Domenico Ferrari was instrumental in organizing the project; he and Dinesh Verma contributed ideas about real-time network communication. Raj Vaswani implemented a large part of DASH kernel; Giuseppe Facchetti, Kevin Fall, Seow-Hiong Goh, Shaun Gordon and Dale Tonogai assisted as well. Thanks also to Ralf Herrtwich and Steve Lucco for helpful comments on this paper, and to Luis-Felipe Cabrera for informed and enlightening discussions.

REFERENCES

[1] G. D. Ripley, "DVI - A Digital Multimedia Technology", *Comm. of the ACM 32*, 7 (July 1989), 811-822.

[2] S. Newman, "The Communications Highway of the Future", *IEEE Communications Magazine 26*, 10 (October 1988), 45-50.

[3] J. H. Irven, M. E. Nilson, T. H. Judd, J. F. Patterson and Y. Shibata, "Multi-Media Information Services: A Laboratory Study", *IEEE Communications Magazine 26*, 6 (June 1988), 27-44.

[4] K. A. Frenkel, "The Next Generation of Interactive Technologies", *Comm. of the ACM 32*, 7 (July 1989), 872-881.

[5] D. B. Terry and D. C. Swinehart, "Managing Stored Voice in the Etherphone System", *Trans. Computer Systems 6*, 1 (Feb. 1988), 3-27.

[6] L. F. Ludwig and D. F. Dunn, "Laboratory for Emulation and Study of Integrated and Coordinated Media Communication", *Proc. of ACM SIGCOMM 87*, Stowe, Vermont, Aug. 1987, 283-291.

[7] B. Arons, C. Binding, K. Lantz and C. Schmandt, "The VOX Audio Server", *Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop*, Ottowa, Ontario, April 20-23, 1989.

[8] R. L. Cruz, "A Calculus for Network Delay and a Note on Topologies of Interconnection Networks", Report no. UILU-ENG-87-2246, University of Illinois, July 1987.

[9] M. Andrews, "Guaranteed Performance for Continuous Media in a General Purpose Distributed System", Masters Thesis, UC Berkeley, Oct. 1989.

[10] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *J. ACM 20*, 1 (1973), 47-61.

[11] D. P. Anderson and S. Tzou, "The DASH Local Kernel Structure", Technical Report No. UCB/CSD 88/463, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Nov. 1988.