The Design of a QoS-Controlled ATM-Based Communications System in Chorus

Geoff Coulson, Andrew Campbell, Philippe Robin, Gordon Blair, Michael Papathomas, and Doug Shepherd

Abstract—**We describe the design of an application platform able to run distributed real-time and multimedia applications alongside conventional UNIX programs. The platform is embedded in a microkernel/PC environment and supported by an ATM-based, QoS-driven communications stack. In particular, we focus on resource-management aspects of the design and deal with CPU scheduling, network resource-management and memory-management issues. An architecture is presented that guarantees QoS levels of both communications and processing with varying degrees of commitment as specified by user-level QoS parameters. The architecture uses admission tests to determine whether or not new activities can be accepted and includes modules to translate user-level QoS parameters into representations usable by the scheduling, network, and memory-management subsystems.**

I. Introduction

In the recent past, significant research has been carried out on high-speed communications systems for distributed real-time and multimedia applications. A surprisingly small amount of this work, however, has considered the issues that arise when ATM and other high-speed networks are interfaced to *conventional* workstations running standard multiprogrammed operating systems such as UNIX. Rather, the research has tended to focus on network issues [5], [22] or has made specific assumptions about the end points of multimedia and real-time communication. Some researchers, for example, have assumed specialized end-systems such as CODEC's or multimedia enhancement units [18], [25]. Others (e.g., [21]) have considered specialized real-time operating systems unable to support conventional applications or conventional modes of operation such as dynamic process creation.

The research reported in this paper is aimed at providing UNIX-compatible system software support for distributed real-time and multimedia applications in an environment of conventional workstations and high-speed networks. Our specific aims are:

- to support a heterogeneous system consisting of PC and workstation end-systems connected by ATM, Ethernet, and proprietary high-speed networks,
- to enable real-time and multimedia applications to enjoy predictable performance in both communications and processing according to user-provided QoS parameters,
- to retain the ability to run standard UNIX applications alongside real-time applications.

Our approach is to use a microkernel operating system, specifically Chorus [7], to underpin both UNIX and real-time applications. A standard UNIX SVR4 "personality" included with Chorus is used to support UNIX applications. Our extensions to Chorus described in this paper are used to support real-time applications.

Our previous work in the field of distributed real-time and multimedia-application support has concentrated on API issues [14], CPU scheduling issues [13], transport issues [9], and network architecture [11]. Complementary to these areas, the present paper focuses on the *resource management strategies* used in our Chorus extensions. The three major resource classes considered are *CPU cycles, network resources* and *physical memory.* In this paper we focus on *end-system-related* communications issues rather than Internet or network resource-management issues (although we do cover resource allocation in the ATM network environment). Broader network and internetworking issues are discussed more fully in [11].

The paper begins by providing, in Section II, some necessary background material on Chorus. Next we present, in Section III, an overview of the architecture of our real-time support infrastructure. This consists of:

- an application programmer's interface (API) at which QoS requirements can be stated,
- a *CPU scheduling framework* that minimizes kernel context switches in both application and protocol processing,
- an ATM-*based communications stack,* which features an enhanced IP layer for internetworking,
- a framework for QoS-*driven memory management*, and
- a framework for *flow*[1] *management*, which integrates the management of resources in both end-systems and the network.

We then, in Section IV, investigate the management of CPU, communications, and memory resources in this architecture. The various resource management functions are categorized as either *static* or *dynamic* as defined in [11]. In essence, static QoS management deals with connect-time issues such as QoS *translation* (i.e., deriving resource quantities from QoS parameters) and *admission testing* (i.e., determining whether new sessions can be created given their specific resource requirement and current resource availability).

Manuscript received March 23, 1994; revised September 19, 1994. This work was supported in part by the Swiss FNRS through Grant 8220-037225; by CNET, France Telecom, as part of the SUMO project; and under U.K. Science and Educational Research Council Grant GR/J16541.

The authors are with the Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster LA1 4YR, U.K.

IEEE Log Number 9407070.

[1] The term *flow* is used to refer to the end-to-end passage of data from a source application, down through the source protocol stack, across the network, up through the sink protocol stack, and eventually to the sink application.

Dynamic resource management, on the other hand, deals with data-transfer time issues. In its full generality, dynamic resource management subsumes *maintenance, monitoring, policing*, and *renegotiation* of QoS levels [10]. The role of the *maintenance* function, which is the only dynamic aspect treated in this paper, is to actually achieve the requested levels of QoS given the resources statically dedicated at resource-allocation time—e.g., by providing suitable scheduling mechanisms and arranging for time-constrained memory access and protocol operation.

In the concluding sections of the paper, we discuss related work in Section V and offer concluding remarks in Section VI.

II. Background on Chorus

Chorus is a commercial microkernel technology that supports the implementation of conventional operating system environments through the provision of "personalities" (for example, a personality is available for UNIX SVR4 as mentioned above). The microkernel is implemented using modern techniques such as multithreaded address spaces and integrated message-based communications. The basic Chorus abstractions are *actors, threads*, and *ports*, all of which are named by globally unique identifiers. Actors are address spaces and containers of resources, which may exist in either user or supervisor space. Threads are units of execution that run code in the context of an actor. They are scheduled according to either a preemptive priority-based or round-robin timeslicing scheme. Ports are message queues used to hold incoming and outgoing messages. The interprocess communication subsystem supports both request/reply messages and asynchronous messages.

Chorus has several desirable real-time features and has been fairly widely used for embedded real-time applications. Its real-time features include preemptive scheduling, page locking, timeouts on system calls, and deferred interrupt handling. Unfortunately, Chorus' real-time support is not fully adequate for the requirements of distributed real-time and multimedia applications, principally because there is no support for QoS specification and resource reservation:

- Although it is possible to specify thread scheduling constraints relative to other threads, *absolute* statements of requirement for individual threads cannot be made.
- In the communications subsystem, the exclusive use of connectionless datagrams makes it impossible to prespecify communications resource allocation.
- Due to the use of a paged virtual-memory system, it is not possible to place bounds on memory-access latency except by the extreme measure of wiring pages.

Note, however, that such limitations are not unique to Chorus: they are shared by most of the other microkernels in current use (e.g., [2], [26]).

III. Architecture

A. Application Programmer's Interface

To remedy its current deficiencies for QoS specification and real-time application support, we have extended the Chorus system call API with new low-level calls and abstractions. The new abstractions, provided in both the kernel and a user-level library, are illustrated in Fig. 1 and described below.

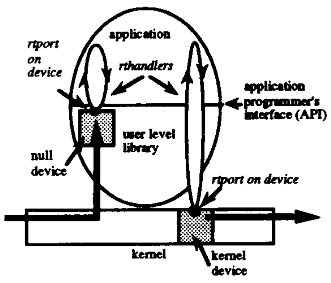


Fig. 1. Devices, rtports, and handlers.

- *rtports*: these are extensions of standard Chorus ports, serving as access points for real-time communications. Rtports have an associated QoS that defines timeliness constraints on communication. They also provide direct application access to buffers thus minimizing copy operations.
- *devices*: these are producers, consumers, and filters of real-time data that support the creation of rtports and provide the memory for their buffers. One special type of device is the *null* device implemented in a user-level library and permits user code to produce/consume real-time data through the use of *rthandlers.*
- *rthandlers*: these are user-supplied C routines that provide the facility to embed application code in the real-time infrastructure. They are attached to rtports at run-time and upcalled on real-time threads by the infrastructure when data are available/required. They encourage an event-driven style of programming appropriate for real-time applications and also avoid the context-switch overhead associated with a traditional *send*()/*recv*()-based interface.
- *QoS-controlled connections*: these are communication channels with a specific QoS.[2] A connection is established between a source and a sink rtport according to a given QoS specification. There are two types of connections: *stream connections* for periodic- and continuous-media data and *message connections* for time-constrained messages. Stream connections are *active* in the sense that they initiate the transfer of data by upcalling a source rthandler (if attached). Message connections differ in that they *passively* wait for a source thread to pass them data via an *ipcSend*( ) call.
- *QoS handlers*: these are upcalled by the infrastructure in a similar way to rthandlers but are used to notify the application layer when QoS commitments provided by connections have been violated.

In addition to these features, the API includes calls for dynamically renegotiating the QoS of open connections and

[2]QoS-controlled connections are *abstractions* and are uniformly used for both remote and local communications. In the remote case, they are implemented in terms of the communications architecture described in Section III-C. In the local case, they are implemented in terms of optimized memory-mapping mechanisms described in Section IV-E.

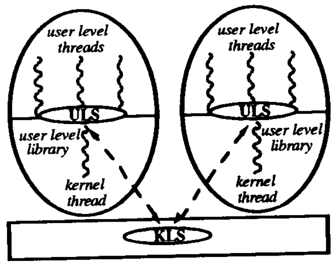


Fig. 2. Split-level scheduling architecture.

for building pipelines of "software signal processing" modules for local continuous-media processing. It also has synchronization primitives based on eventcounters and sequencers that incorporate the notion of *deadline inheritance* [15], whereby a "worker" thread carrying out a task on behalf of a calling thread inherits the deadline of the caller. Full details of the continuous media API are specified in [14] and [15].

B. Scheduling Architecture

The scheduling architecture exploits the concept of *lightweight threads*, which are supported in a user-level library and multiplexed on top a single Chorus kernel thread per actor. In this context, we refer to Chorus kernel threads as *virtual processors* (VP's). The scheduling architecture is a *split-level* configuration [17] consisting of a single kernel scheduler (KLS) to schedule VP's, and per-actor user-level schedulers (ULS's) to schedule lightweight threads on those VP's (see Fig. 2).

The advantage of lightweight threads and user-level scheduling is that context-switch overhead is minimal. On the other hand, the drawback of user-level scheduling is that, by definition, it cannot ensure that CPU resources are fairly shared across multiple actors. This is the role of kernel-level scheduling. The split-level architecture combines the benefits of both user-level and kernel-level scheduling by maintaining the following invariants:

1) each ULS always runs its most urgent[3] lightweight thread, and
2) the KLS always runs the VP supporting the *globally* most urgent lightweight thread.

The scheduling invariants are maintained via a KLS/ULS information exchange realized in terms of shared KLS/ULS memory areas and software interrupts [17]. The shared memory area is divided into per-VP areas, each of which contains the urgency of the most urgent runnable lightweight thread known to its associated VP (along with some other information as described below). These urgency values are read by the KLS on each kernel-level rescheduling operation to determine the next VP to schedule. Software interrupts are used by the KLS to inform VP's of the occurrence of real-time events in a timely fashion. Such events include timer expirations (used to implement preemption in user-level scheduling), and data arrivals from local kernel devices or from the network. Software interrupts are always targeted at VP's but can be initiated either by kernel components (e.g., the KLS) or by library code in other application actors (see Section IV-E).

The scheduling scheme also embodies the notion of *conditional urgency*. This allows not only the urgency of the most urgent runnable lightweight thread to be taken into account by the KLS (as above), but also the urgency of currently *blocked* threads. The implementation, which again exploits the shared memory area, uses per-VP *conditional urgency* sets that contain {*thread_id, event, urgency*} triples. In each triple, *urgency* represents the urgency that thread *thread_id* would have if only *event* was available to unblock it. The *urgency* values must all be greater than the urgency of that VP's most urgent runnable lightweight thread and the *event* values must all refer to events expected from an external source. Thus, when the KLS has an event to deliver, it will run the VP to which *event* is addressed iff there is a matching triple in that VP's conditional urgency set *and* the indicated *urgency* value is globally more urgent than that of any other lightweight thread.

To avoid potential violations of the scheduling invariants, we implement the system call interface seen by lightweight threads in terms of nonblocking system calls [24]. If lightweight threads performing system calls were permitted to block their underlying VP, they would also necessarily block all other lightweight threads multiplexed on that VP. Then the scheduling invariants would be violated if one of these other threads happened to be the globally most urgent. Nonblocking system calls avoid this problem by returning *immediately* from system calls, thus allowing ULS's to block the calling lightweight thread at the library level while continuing to run other lightweight threads on the actor's VP. The results of calls are eventually notified to the ULS via software interrupts. On receipt of such an interrupt the ULS stores the result in the data structures of the original lightweight thread and then lets it "return" from its system call. Thus application code sees only blocking system calls (as per standard Chorus), and the complexities of nonblocking calls are masked by library code.

The implementations of software interrupts and nonblocking system calls also exploit the shared kernel/user memory area. To deliver a software interrupt, the kernel places an event identifier and parameters in the shared-memory area and then alters the program-counter field of the user VP's context structure (also in shared memory) to point to a well-known entry point in the ULS. Thus, when the VP is next scheduled by the KLS, the VP immediately enters the ULS, which picks up the event identifier and parameters and schedules a lightweight thread to deal with the event. The implementation of our variant of nonblocking system calls, which, because of the analogy with software interrupts, we refer to as *asynchronous system calls* [15], is similar. The user-level library places an operation identifier and parameters in shared memory and then sets an "operation request" bit. The KLS, when it runs at the next system clock tick, notices that the operation request bit is set and copies the user's parameters to the appropriate VP or kernel-server thread as determined by the operation identifier. Note that both software interrupts and asynchronous system

[3]The notion of "urgency" is dependent on the scheduling policy used (e.g., it would be *deadline* for EDF scheduling and *priority* for rate monotonic scheduling). The issue of scheduling policies is deferred until Section IV-C.

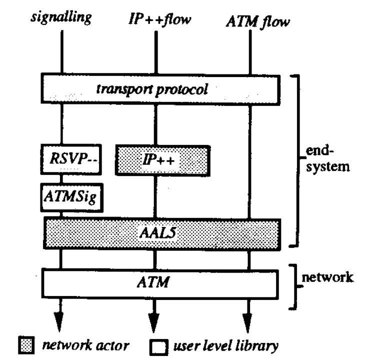


Fig. 3. Communications architecture.

calls avoid a special domain crossing; the call is actually effected the next time the recipient context (i.e., the ULS or the kernel) gets control by other means.

C. Communications Architecture

The standard Chorus communications stack was designed for the support of connectionless datagram services and uses retransmission strategies to enhance reliability. In contrast, our communications architecture (see Fig. 3) is intended to support QoS-controlled connection-oriented communications and configurable error control. Because of these disparate design goals, we have initially designed our stack to operate entirely separately from the existing Chorus facilities (however, we do intend in the future to integrate the functionality of the two stacks in a unified architecture).

1) Abstract Layering: The communications architecture enforces a strong distinction between communication for signaling purposes (i.e., connection establishment, network resource management, and connection tear-down), and user data-transfer purposes. The AAL5 and ATM layers are common to both the signaling and the user data stack and are described below, as is the transport layer. The signaling stack specific layers comprise an upper network sublayer for resource management in IP routers and a lower network sublayer for resource management in ATM switches. The IP layer is a subset of the existing RSVP network resource reservation protocol [32]. The ATM signaling protocol, called ATMSig, is a subset of the ATM Forum's UNI 3.0 [4]. Note that the signaling stack also includes a reliable signaling message protocol over AAL5, which is a subset of the service-specific connection-oriented protocol (SSCOP) (not shown in Fig. 3).

The user data stack is positioned alongside the signaling stack. The upper architectural layer is a connection-oriented transport protocol [8], which provides for QoS specification at connection time (including configurable error control), in service QoS renegotiation, and end-to-end flow control (via a rate-based mechanism). Other transport-layer functions, such as admission control, resource reservation, performance monitoring, and dynamic QoS maintenance are supported outside the transport protocol proper by the scheduling, connection, and memory-management subsystems described in this paper.

The user stack's IP layer, called IP++, allows us to interwork outside the ATM network in a heterogeneous environment. It offers QoS-enhanced facilities along the lines of those proposed in Deering's simple Internet protocol plus (SIPP) [33]. In particular, IP++ uses a packet header field called a *flow-id* to identify IP packets as belonging to a particular connection or *flow*, and a *flow-spec* (see Section IV-D) to define the QoS associated with each flow. Flow-specs are held by IP++ routers[4] and used to determine the resources dedicated to the router's handling of each IP++ packet on the basis of its flow-id. The state held by routers is initialized at connection set up time by the RSVP signaling protocol. Below the IP layer we use an AAL5 ATM adaptation layer service to perform segmentation and reassembly of IP packets into/from 53-byte ATM cells.

The lowest layer of our architecture is based on the Lancaster Campus ATM network. This delivers ATM to a mix of workstations, PC's, and multimedia devices designed at Lancaster [25]. It also interconnects a number of Ethernets and interfaces to the rest of the U.K. via a 10Mb/s SMDS connection to the UK SuperJANET 100 Mb/s Joint Academic Network. The PC's that run the system described in this paper are directly connected to a 4×4 ATM switch via ISA bus interface cards.

2) Mapping the Architecture onto Chorus: In implementation, we map the abstract layered communications architecture partly onto per-actor user-level libraries and partly onto a single, per machine, supervisor actor called the *network actor.*[5] The signaling aspects of the transport layer are implemented in the flow-management-protocol actor described in Section III-D. The data-transport aspects of the transport layer are implemented in the same user-level library[6] that supports the API abstraction discussed in Section III-A. This allows the transport service interface to be provided by the library-level rtport and rthandler abstractions defined in that section. The transport protocol communicates with the network actor via software interrupts for receive side and asynchronous system calls for send-side communications (see Section III-B).

Below the transport protocol, the rest of the communications architecture, including the ATM device driver, is implemented in the network actor. The two signaling protocols, RSVP and ATMsig, are not described here as they are considered to be outside the scope of this end-system-oriented paper. In the user stack, the major complexity involved in the IP++ implementation is in supporting the routing function. This is required when the current host is neither the source nor sink of a flow but is merely routing packets from one network

[4]Flow specs are also used to control resource reservation at the ATM level in ATM switches in an analogous way.

[5]Note that this is distinct from the standard Chorus "network actor," which is also called the "network device manager."

[6]In fact, the transport protocol also runs in supervisor space in the case of connections terminated by rtports on hardware devices. This is so that data passing between such devices on the same machine, or between such devices and the network card, do not incur the overhead of passing through user space. The API still permits applications at the user level to monitor the flow of data in such connections by attaching rthandlers.

to another. In this case, CPU and memory resources are dedicated to flows on the basis of a flow-spec supplied by the flow-management protocol (see Section IV-D). Otherwise, the function of the IP++ layer is effectively null as SDU sizes are restricted and no SAR is required at the IP layer (see below).

AAL5 is also implemented in the network actor. A software AAL5 implementation is required because our ATM interface cards only support data transfer at the granularity of ATM cells. The AAL5 implementation uses a single thread on the receive side and per-flow threads on the send side to perform segmentation and reassembly with optional checksumming. The use of per-flow threads reduces multiplexing in the stack to an absolute minimum as recommended in the literature [27]. Currently, the maximum service-data unit size for the AAL5, IP++, and transport layers alike is restricted to 64 Kb. This means that no further segmentation/ reassembly is required above the AAL5 layer.[7] The ATM cards are interrupt-driven and communication between the interrupt service routines and the per-flow AAL5 threads is via Chorus "miniports." See Section IV-D for more details of the low-level cell-handling functions and AAL5 implementation.

D. Memory Management Architecture

The standard abstractions used by the Chorus virtual memory system are *segments, regions* and *mappers*:

- Segments are the unit of information exchange between the outside world (e.g., files or swap areas) and the virtual memory (VM) layer in the kernel. In main memory segments are represented by so-called *local caches* of physical pages.
- Regions are the unit of structuring of actor address spaces. A region contains a portion of a segment mapped to a given virtual address. Regions have associated access rights policed by the VM layer.
- Mappers are supervisor actors that implement the link between external segments and their main-memory representation and maintain the protection and consistency of segments. Mappers are accessed from the kernel via an upcall RPC interface when the kernel needs to bring in or swap out a page of a segment.

The purpose of our extended memory-management architecture, which is built on top of the above abstractions, is to ensure that applications and QoS-controlled connections can access memory regions with *bounded latency*. It is of little use to offer guaranteed CPU resources to threads if they are continually subject to nonpredictable memory-access latency due to arbitrary page faulting.[8] Our design encapsulates most of the QoS-driven memory-management functionality inside a QoS-enhanced mapper called the *QoS mapper*. The roles of the QoS mapper are:

- supplying application actors with memory regions offering latency-bounded access,
- determining whether or not requests for QoS-controlled memory resources should succeed or fail,
- preempting QoS-controlled memory from "low urgency" threads on behalf of "high-urgency" threads when necessary, and
- efficiently remapping QoS-controlled memory regions from one actor to another.

In addition to servicing requests from the kernel VM layer, the QoS mapper is used to implement the connection abstraction in the intramachine connection case (see Section IV-E). User-level code can also invoke the QoS mapper via extended versions of the *rgnAllocate*() and *rgnFree*() Chorus system calls. These respectively allocate and free a QoS-controlled region of memory at connection establishment time.

E. Flow-Management Architecture

We have described frameworks for the management of CPU, network and memory resources but have said nothing yet of the relationship between these frameworks. It is the task of the *flow-management architecture*, and in particular the flow-management protocol (FMP) [11], to realize this relationship.

The FMP must arrange, at connection time, for the allocation of suitable CPU, memory, and network resources according to the user specified QoS of the requested connection. The FMP cooperates with the CPU memory management and network subsystems and partitions the responsibility for QoS support among individual resource managers. For example, for remote communications, the FMP partitions the API-level *latency* QoS parameter (see Section IV-A) between the network and the CPU resource managers on each end system.

The FMP is also responsible for *dynamic* QoS management in flows. In this role, it can adapt to degradations in one resource by compensating in terms of another. Ideally, it will do this without either involving the application or violating overall the QoS specification. For example, an increase in jitter caused by the network can be transparently compensated for by an increased buffer allocation at the receiver—as long as the latency QoS is not thereby compromised.

The flow-management architecture adopts a similar split-level structure to the scheduling and communications architectures. First, when a new QoS-controlled connection is requested, a QoS *translation function* (see Section IV) in the user-level library determines the resource requirements of the request. Then, the output of the QoS translator is directed to the FMP that runs in a per-machine FMP actor (see Fig. 4). QoS translation is treated in detail in Section IV.

IV. Resource Management

Prior reservation of resources to connections is necessary to obtain guaranteed real-time performance. This section describes the resource-reservation framework in our system and shows how user-level QoS parameters are used to derive the resource requirements of connections and make appropriate reservations. It also examines some dynamic resource management issues. This paper concentrates on the reservation of *specific* resources (i.e., CPU, memory and network resources)

[7] It would be a relatively straightforward extension to support arbitrarily sized buffers at the API level by supporting segmentation and reassembly in the transport protocol if this proved necessary.

[8] Note that, in addition to buffers, it is also necessary to provide bounded latency access to code and stack regions of QoS-controlled threads if QoS guarantees are to be maintained.

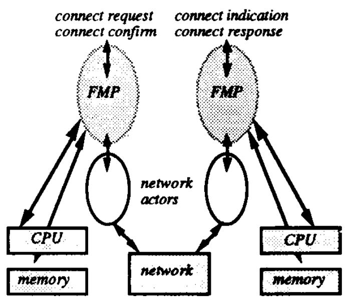


Fig. 4. Flow management architecture.

rather than treating resource reservation as an integrated activity driven by the FMP.

In outline, there are two stages in the resource-reservation process. QoS *translation* is the process of transforming user-level QoS parameters into resource requirements and *admission testing* determines whether sufficient uncommitted resources are available to fulfill those requirements.

A. User QoS Parameters

The QoS parameters visible at the API level are as follows:

```
typedef enum {best_effort, guaranteed} com;
typedef enum {isochronous, workahead} del;
typedef struct {
        com commitment;
        int buffsize;
        int buffsize;
        int priority;
        int latency;
        int error;
        int error_interval;
        int buffrate;
        int jitter;
        del delivery;
} StreamQoS;

typedef struct {
    com commitment;
    int buffsize;
    int priority
    int latency;
    int error;
} MessageQoS;

typedef union {
    MessageQoS mq;
    StreamQoS sq;
} QoSVector;
```

The two structures in the QoSVector union are for stream connections and message connections, respectively. The first four parameters are common to both connection types. *Commitment* expresses a degree of certainty that the QoS levels requested will actually be honored at run-time. If commitment is *guaranteed*, resources are permanently dedicated to support the requested QoS levels. Otherwise, if commitment is *best effort*, resources are not permanently dedicated and may be preempted for use by other activities. *Buffsize* specifies the required size of the internal buffer associated with the connection's rtports. *Priority* is used for fine-grained control over resource preemption for connections; all things being equal, a connection with a low priority will have its resources preempted before one with a higher priority.

Latency refers to the maximum tolerable end-to-end delay, where the interpretation of "end-to-end" is dependent on whether or not rthandlers are attached to the rtport. If rthandlers are attached, latency subsumes the execution of the rthandlers; otherwise it refers to rtport-to-rtport latency. When rthandlers are attached a further, implicit, QoS parameter called *quantum* becomes applicable. The value of this parameter is dynamically derived by the infrastructure whenever an rthandler is attached to an rtport. It is defined as the sum of the rthandler execution time and the execution time of the protocol code executed by the same thread directly before/after the rthandler is called.[9] To determine the quantum value, the infrastructure performs a "dummy" upcall of the rthandler and measures the time taken for it to return (a Boolean flag is used to let the application code in the rthandler know whether a given call is "real" or dummy). It is the responsibility of the application programmer providing the rthandler to ensure that the dummy execution path is similar to the general case. Although the value of quantum is dynamically refined as the connection runs, an inaccurate initial value will inevitably cause QoS violations.

Error has different interpretations depending on the connection type. For stream connections, it is used in conjunction with *error_interval* and refers to the maximum permissible number of buffer losses and corruptions over the given interval. In the case of message connections, it simply represents the probability of buffers being corrupted or lost (note that *error_interval* is not applicable to message connections).

For stream connections, there are three additional parameters, *buffrate, jitter*, and *delivery*, which have no counterparts in message connections. *Buffrate* refers to the required rate (in buffers per second) at which buffers should be delivered at the sink of the connection. *Jitter*, measured in milliseconds, refers to the permissible tolerance in buffer delivery time from the periodic delivery time implied by buffrate. For example, a jitter of 10 ms implies that buffers may be delivered up to 5 ms either side of the nominal buffer delivery time. *Delivery* also refines the meaning of buffrate. If *isochronous* delivery is specified, stream connections attempt to deliver *precisely* at the rate specified by buffrate; otherwise, if delivery is *workahead*, it is permitted to "work ahead" (ignoring the jitter parameter) at rates temporarily faster than buffrate. One use of the workahead delivery mode is to more efficiently support applications such as real-time file transfer. Its primary use, however, is for pipelines of processing stages where isochronous delivery is not required until the last stage [14].

[9] Actually there is a third component to the quantum value, which is the per-buffer time taken by per-connection transmit threads at the ATM level. See Section IV-D for details.

Best effort $\begin{cases}\text{- isochronous } (B_I)\\ \text{- workahead } (B_W)\end{cases}$ Guaranteed $\begin{cases}\text{- isochronous } (G_I)\\ \text{- workahead } (G_W)\end{cases}$

Fig. 5. Resource classes.

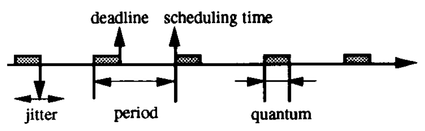


Fig. 6. Periodic thread scheduling terminology.

B. Resource Classes

In the following sections, we distinguish *four* major classes of QoS-controlled connection for resource-management purposes. These resource classes, named G_I, G_W, B_I, and B_W are selected on the basis of the *commitment* and *delivery* QoS parameters described above. They are defined and illustrated in Fig. 5.

In addition to the two best effort classes shown in Fig. 5, a third best effort class, B_C, is distinguished, which refers to nonreal-time Chorus and UNIX threads out of the scope of the real-time extensions. Additionally, all three best effort classes are often grouped together and referred to by the shorthand name B.

C. The CPU Resource

1) QoS *Translation:* For admission testing and resource allocation purposes for stream connections, it is necessary to know the *period* and *quantum* of the threads associated with the connection. The period is simply the reciprocal of the *buffrate* QoS parameter and the quantum is implicitly derived at connect time as explained in Section IV-A. Fig. 6 illustrates the notions of period and quantum together with the related scheduling concepts of *scheduling time, deadline*, and *jitter*.

For message connections, *sporadic server* threads are used at the receive side.[10] One sporadic server per application actor is provided for each of the two applicable commitment classes (viz., G_W and B_W; isochronous delivery is not applicable to message connections), and each sporadic server handles all the message threads in its class. The quantum of each server is set to the *maximum* of the quanta of all the message threads in its class to ensure that adequate processing time is available for any of the server's associated threads. The period of each server is heuristically derived as follows

$$\mathit{period} = \min(\mathit{recv\_latency}_1, \cdots, \mathit{recv\_latency}_n).$$

$\mathit{Recv\_latency}_i$ is the proportion of the total end-to-end latency allocated by the FMP to the receive end-system for message connection i. This method of calculating *period* is a compromise that requires less resource than an optimal period (i.e., the optimal period, $1/\mathrm{quantum}$, would ensure that the server was always ready to service a message but would take all the CPU resource allocated to the class!) while offering a reasonable probability that the server will be ready when a message arrives.

[10]There is no thread implicitly associated with the source side of message connections. Dedicated threads are only applicable when rthandlers are used, and it is not useful to attach rthandlers to the source of message connections as message connections are not *active* in the sense of stream connections.

2) Admission Testing: The semantics of thread scheduling for each of the three resource classes are as follows:

- G_I: threads for these connections are (preemptively) scheduled to run such that the completion of a quantum is guaranteed to be completed by the logical arrival time + quantum + j (where j is the jitter QoS parameter and logical arrival time is the start of the requisite period). An extended earliest deadline first [23] (EDF) algorithm and admission test is used to ensure this behavior.
- G_W: these are scheduled according to the standard preemptible EDF policy. The jitter QoS parameter is ignored and quanta may be scheduled ahead of their logical arrival time to permit workahead. Again, an admission test is performed.
- B: these are scheduled according to the preemptible earliest deadline first policy but no admission test is used.

Each of the G and B resource classes is allocated a fixed portion of the CPU resource. Note, however, that the "firewall" that this separation implies is used only to limit the number of threads in each class—not to restrict the use of CPU cycles at run-time. If there are unused resources in one class, these resources are automatically exploited by the other class at run-time (see Section IV-C3).

The firewalls can be dynamically altered at run-time by the programmer, but a typical configuration will allow a relatively small allocation for G threads. This is to encourage users to choose best effort threads wherever possible. Best effort threads should be perfectly adequate for many "soft" real-time needs so long as the system loading is relatively low. The guaranteed classes should only be used when absolutely necessary—for example, when threads are delivering data to a end device intended for human perception such as a video frame buffer.

The admission tests for G_I threads are

$$\sum_{i=1}^{N_G} \frac{\mathit{quantum}_i}{\mathit{period}_i} \le R_G, \quad 0 \le R_G \le 1$$

$$\sum_{i=1}^{N_G} \frac{\mathit{quantum}_i}{\mathit{quantum}_i + \mathit{jitter}_i} \le 1.$$

The admission test for this class is a two-stage process, and each of the two tests are modifications of the well-known Liu/Layland test [23] (this guarantees that each quantum in the given set of tasks can be completed *at least* by the end of its period as long as it is runnable at the start of its period). The first of our tests ensures that the overall resource used by all G threads is not greater than the allocated portion. N_G refers to the total number of G threads in the system and R_G refers to the portion of CPU resources dedicated to this class of threads (such that $R_G + R_B = 1$ where R_B represents the portion of the CPU resource dedicated to B threads).

The second test imposes the additional constraint that each quantum must complete by the end of its user-stated jitter bound rather than simply by the end of the requisite period.

Note that this second test is rather conservative (e.g., if a thread with zero jitter is requested the test will pass only this one thread!). However, we relax this overconservative property by also taking into account the notion of *harmonic sets* (i.e., sets of threads all of whose periods are divisible by the period of the member with the smallest period). It can be shown that harmonic sets can be scheduled without clashes as long as, in each period of the thread with the smallest period, it is possible to fit the quanta of all the threads in the set that fall within this period. This remains true even where the threads involved have a requirement for zero jitter. We are also working on an approach that allows us to optimally exploit the degrees of freedom allowed by the threads with relaxed jitter constraints for use by those with tight constraints [34].

For G_W threads the admission test is simply

$$\sum_{i=1}^{N_G} \frac{quantum_i}{period_i} \leq R_G.$$

For B threads there is no admission test and the test for G_W sporadic servers is identical to that for G_W periodic threads. Each time a new message connection is created that alters the period or quantum of its server, a new admission test must be performed to ensure that the modified sporadic server can still be accommodated in the appropriate resource class.

2) Dynamic QoS Maintenance: At run-time, the dynamic operation of the scheduling scheme uses a combination of *priorities*[11], *deadlines*, and *scheduling times* to capture the abstract notion of "urgency." The scheduler uses three distinct priority bands into which the four classes of thread are mapped. The semantics of priority are that at any given time there is no runnable thread in the system that has a priority greater than the currently running thread. Within each priority band, all threads are made runnable when their scheduling time is reached and actually run when their deadline is earlier than the deadline of all other runnable threads in the band.

The G_I class is given a single high-priority band (only critical Chorus server threads such as the pager daemon are allocated a higher band). B threads are given the next highest band and G_W threads are initially assigned to the lowest priority band. G_I threads are made runnable whenever their logical arrival time is reached (i.e., the start of the period pertaining to their current quantum). As mentioned above, G_W threads are initially assigned to the lowest priority band but they are "promoted" to the G_I band when their logical arrival time is reached. This means that they can enjoy workahead when resources allow, but not at the expense of G_I and B threads. B_W threads are also runnable before their logical arrival time but are not similarly promoted. Finally, B_I threads only become schedulable at a time indicated by the deadline minus the quantum time. This approximates isochronicity to the extent that it removes the possibility of jitter causing threads to complete *before* time although it still leaves the possibility of them completing after *time*. This overall scheme, in conjunction with the admission tests, ensures that G_I threads always meet their jitter constraints, G_W threads always *at least* meet their rate requirement, and B threads optimally share the resources left to them.

Nonreal-time threads in the B_C class (e.g., those from conventional UNIX applications) are assigned appropriate priorities so that they receive reasonable service according to their role. Their deadline and scheduling time are always set to *now* so that they are effective scheduled solely on the basis of their priority. As an example, B_C threads fulfilling an interactive role would have relatively high priority, which may be greater than that of B threads. Other B_C threads, such as compute-bound applications and nontime-critical daemons, will have accordingly lower priorities.

[11]Note that the "priority" in this discussion is different from the priority API level QoS parameter. In this section, priority is an internal thread scheduling attribute, not visible or directly manipulable from the API level.

D. The Network Resource

1) QoS Translation: The network subsystem offers guarantees on *bandwidth, delay bounds*, and *packet loss*. To enable it to do this, the QoS translation function maps the API-level QoS parameters onto a *flow spec*, which is a representation of QoS appropriate to the IP++ and ATM levels:

```
typedef struct {
   int          flow_id;
   com          commitment;
   int          mtu_size;
   int          rate;
   int          delay;
   int          loss;
} flow_spec_t;
```

Flow_id uniquely identifies the network-level flow. It is the virtual circuit identifier for flow specs used at the AAL5/ATM level and the flow id in the IP++ packet header for flow specs used at the IP level. *Mtu_size*[12] refers to the maximum transmission unit size and *rate* refers to the rate at which these units are transmitted. These are directly derived from the *buffsize* and *buffrate* API-level QoS parameters. *Delay* comprises that portion of the API-level latency parameter that has been allocated, by the FMP, to the network. It subsumes both propagation and queuing delays in the network. Finally, *loss* is an upper bound probability of *mtu* loss due to buffer overflow at switches and routers. Loss is trivially derived from the *error* and *error_interval* API-level QoS parameters.

2) Admission Testing: In the network, only two traffic classes are recognized: *guaranteed* and *best effort* as denoted by the *commitment* API-level QoS parameter. Admission testing and resource allocation are only performed for the former; best effort flows use whatever resource is leftover.

For guaranteed flows, three admission tests are performed at each switch along the chosen path: a bandwidth test, a delay-bound test and a buffer-availability test. If, at the

[12]Although the discussion and admission tests in this section apply generically to both the IP++ and ATM layers, the admission tests are described here, for clarity, in an ATM context only. Mtu_size in the case of ATM cells is 53 b and in the case of IP++ packets is 64 Kb. One restriction of the admission tests is that they are only applicable to switches/routers with a *single* CPU. As we use single CPU ATM switches, this assumption is justified in our implementation environment.

current switch, the admission-control tests are successful, the necessary resources are allocated. Then the switch appends details of the cumulative delay incurred so far and forwards the flow spec to the next switch. Eventually, the remote end-system performs the final tests and determines whether or not the QoS specified in the flow spec can be realized.

If the required QoS is realizable, the remote end-system returns a confirmation message to the initiating end-system. As it traverses the same route in reverse, the admission test protocol *relaxes* any overallocated resources at intermediate switches [3].

3) Bandwidth Test: The bandwidth test consists in verifying that enough processing (switching) power is available at each traversed switch to accommodate an additional flow without impairing the guarantees given to other flows. The admission test must satisfy worst case throughput conditions; this happens when all flows send packets back to back at the peak rate. As in Section IV-C2 the admission control test is based on [23]

$$\sum_{i=1}^{N} t_i \cdot rate_i \leq R.$$

Here, t_i refers to the service time (cf. mtu quantum) of flow i in the current switch, where there are N flows and $rate_i$ is the rate of the ith flow (i.e., 1/mtu period). $R, 0 \leq R \leq 1$, represents the portion of resource dedicated to guaranteed flows.

4) Delay-Bound Test: The delay-bound test determines the minimum acceptable delay bound that does not cause scheduler saturation. There are two phases in the delay bound test. First, each switch on the data path computes a local delay bound. Second, it is checked that the sum of all the local delay bounds do not exceed the flow spec's *delay* parameter.

The first phase calculation is taken from [16]

$$d = \sum_{i=1}^{N_U} t_i + T.$$

Here, d is the local delay bound incurred at the current switch by the current flow. As before, t_i refers to the service time of flow i in the current switch, but here the index variable i ranges over the members of a set U. The set U contains those flows supported by the current switch whose local delay bound is lower than the sum of the service times of *all* flows supported by the current switch. N_U represents the cardinality of U. T represents the largest service time of all flows in a set V where V is the complement of set U. A full proof of the theorem underlying this formula can be found in [16]

The second phase calculation is

$$\sum_{i=1}^{N_s} d_n \leq delay.$$

This merely requires that sum of the delays at each switch is less than the delay parameter in the flow spec. N_s refers to the number of switches on the path and d_n refers to the nth value of d obtained from the first phase calculations.

5) Buffer Availability Test: The amount of per-switch memory allocated to a new flow must be sufficient to buffer the flow for a period that is greater than the combined queuing delay and service time of its packets. The calculation for buffer space is

$$buffersize = mtu\_size \lceil d \cdot rate \cdot loss \rceil .$$

Here, *buffersize* represents the amount of memory that must be allocated at the current switch for the current flow. The combination of the queuing delay and service time is bounded by d as derived from the first phase delay formula above.

6) Dynamic QoS Maintenance: Much of the dynamic QoS maintenance for the execution of communications protocols in the end-system is encapsulated either in the protocols themselves (e.g., error control), in the scheduling subsystem (e.g., rate control, maintaining latency, and jitter bounds) or in the memory management subsystem (i.e., buffer management). However, one interesting QoS-maintenance issue not in this category is the interleaving of ATM cells at the network interface from different connections on the basis of their QoS [11]. On many ATM interface cards with on-board AAL's this is taken care of in hardware but in our case we have been able to investigate this issue in software due to the fact that our interface cards only deal with the ATM level.

The receive side cell processing is simple. The receiver interrupt service routine[13] reads the VCI of the current cell while it is still on the ATM interface card. The interrupt-service routine then dispatches a receiver thread to copy the cell payload into the appropriate partially assembled AAL5 packet, and when the receiver thread sees the last cell of an AAL5 packet, it raises a software interrupt to the appropriate VP. Unfortunately, we are not able to perform any QoS-driven scheduling on the receive side as it has proved imperative to get each cell off the board as quickly as possible to avoid excessive cell loss due to FIFO overrun. Thus, the receive thread is given a scheduling priority higher than even G_I threads.

On the transmit side, though, we are able to schedule cells more intelligently and have designed an EDF-based *cell-level scheduler*. Application actors running send-side user-level transport-protocol code deliver buffers to the network actor via a system call. This informs the network actor of 1) the location in its address space into which the buffer has been mapped and 2) the deadline of the buffer (the end of the quantum of the transport protocol thread).

The cell-level scheduler runs in the context of the transmit-interrupt service routine, which is periodically activated by the ATM card to signal that cells can be copied to the card for transmission. The scheduler chooses to run one of a number of per-connection *transmit threads* by sending a message to a miniport on which the transmit thread is waiting (see Fig. 7). The choice of thread to activate is made on the basis of priority, deadline, and scheduling time as described in Section IV-C3. Each transmit thread is given the same priority band as its associated user-level lightweight thread, and the deadline

[13]Although the card interrupts on each cell arrival, the receive thread "greedily" consumes any cells waiting in the card's FIFO each time it runs, thereby avoiding an interrupt for each cell.

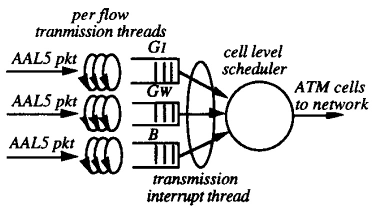


Fig. 7. Cell-level scheduler.

of each thread is derived from the deadline of the next cell in the thread's associated buffer. Cell deadlines themselves are derived by giving each cell in the buffer a specific temporal offset from the deadline of the entire buffer. The scheduling time of each thread becomes *now* whenever the thread has a buffer to send.

The transmit threads are allocated at connection establishment time and are taken into account in the scheduling admission tests. This is done by adding a time t_{tx} to the *quantum* parameter of the connection's transmit side lightweight thread (see Section IV-A); t_{tx} is calculated as *cells* $\times$ t_{cell} where *cells* is the number of ATM cells in a buffer of size *buffsize* and t_{cell} is the average time taken to transfer an ATM cell to the interface card.

E. The Memory Resource

1) QoS Translation: We can deduce two memory-related quantities from the user-supplied QoS parameters at connection establishment time: 1) the number of buffers required per connection and 2) the required access latency associated with those buffers. Buffers are implemented as Chorus memory regions.

2) Number of Buffers: To calculate the end-system buffer requirement, the *buffsize, buffrate*, and *jitter* QoS parameters are used. It is also necessary to take into account the network delay bound, *delay*, offered by the FMP. The network delay bound will typically permit a larger degree of jitter than the API-level jitter bound and any discrepancy must be made good through the use of additional jitter smoothing buffers. Given these input parameters, the expression for the number of buffers required at the receiver is

$$\mathit{buffers} = \mathit{buffrate}\left(\mathit{delay} + \mathit{quantum} + \frac{\mathit{jitter}}{2}\right).$$

In this formula, the expression in the brackets represents the maximum time for which any single buffer must be held. *Delay* is the delay bound specified in the network-level flow spec while *quantum, jitter*, and *buffrate* are API-level QoS parameters. *Jitter* is divided by two because the jitter parameter expresses both lateness and earlyness and it is only the lateness component that need be taken into consideration.

Only one buffer is required at the sender due to the structure of the send-side communications architecture: each buffer is assumed to be "on the wire" before the start of the next period.

3) Region Access Latency: There are basically two qualities of memory access available in the standard Chorus system. These relate to the access latency of swappable pages and the access latency of locked pages. The latency bound of the former is a function of 1) the delay due to the RPC communication between the VM layer and the mapper and 2) the delay associated with the external swap device.[14] The latency bound of the latter is much smaller and is a function of the system bus and clock speed.

We assign either swappable or locked regions to connections on the basis of their resource class as follows:

- G_I: buffer regions allocated to these connections are locked and nonpreemptible.
- G_W: buffer regions for these connections are locked but are potentially preemptible by memory requests from G_I connections if memory resources run low.
- B: buffer regions for these connections are assigned from standard swappable virtual memory. These regions may be explicitly locked by the API library code but are subject to preemption from by both G_I and G_W connections. The decision as to whether the library code should lock buffers or not is determined by the *priority* API-level QoS parameter.

The QoS mapper can deduce the class of each memory request on the basis of the *commitment, delivery*, and *priority* QoS parameters initially passed to the extended *rgnAllocate*() system call and retained to validate future operations on regions.

4) Admission Testing: In its admission testing role, the QoS mapper maintains tables of all the physical memory resources in the system. In a similar way to the KLS, it also maintains firewalls and high- and low-water marks between resource quantities dedicated to the different connection classes. The B section is used by all standard and nonreal-time applications as well as best effort connections.

If no physical memory is available to fulfill a request from a G_I connection, the QoS mapper can *preempt* a locked memory region from an existing B or G_W connection. Similarly, G_W connections can preempt locked regions from B connections. The QoS mapper chooses for preemption the buffer associated with the connection with the lowest *priority* in the lowest class available. The effect of preemption is simply to transform locked memory into standard swappable memory. This, of course, may result in a failure of the preempted connection's QoS commitment. However, a software interrupt is delivered to the ULS of a thread whose memory has been preempted so that if QoS commitments are violated, the connection concerned can deduce the likely reason.

5) Dynamic QoS Maintenance: The only dynamic QoS mapper function we have yet considered in detail is the region remapping function. This is used when buffers are mapped from one actor to another in a QoS-controlled connection between local rtports. Region remapping is particularly important in the context of *pipelines*, which arises when applications are structured as chains of modules that sequentially process a stream of real-time data. Pipelines can be implemented either within single actors or across multiple actors (or, indeed,

[14] We intend in the future to look at the possibility of bounding the access latency to swappable pages (e.g., through specialized page replacement policies and disk layout strategies), but our present design simply considers the access latency of swappable pages to be unbounded.

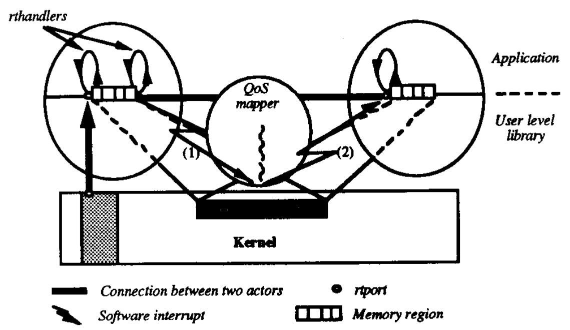


Fig. 8. Pipeline example.

across multiple machines although it is only the intramachine cases that concern us here).

Pipelines across multiple actors are implemented using software interrupts as the control transfer mechanism. When a pipeline stage wants to send a buffer of data to a subsequent stage in another actor, the user-level library implementing the QoS-controlled connection performs a software interrupt

```
int raiseEvent(VP * dest; int event; bool unmap;
        VmAddr addr; VmSize size);.
```

The *raiseEvent*() system call specifies a destination actor and details of the memory region to be remapped. When the kernel receives this call, it invokes the QoS mapper, which maps the specified region into the destination address space (the Boolean argument is used to control whether or not the region is also unmapped from the caller's address space). The QoS mapper then forwards a software interrupt to the target VP, passing as an argument the virtual address at which the region is mapped (see Fig. 8). Note that, in many cases, it is only necessary to perform this mapping the first time data is passed along the pipeline. Subsequent transfers can be accomplished using the existing shared region that *raiseEvent*() has already established and simply passing control with a call of raiseEvent with null *addr* and *size* parameters.

The extra cost due to the QoS mapper invocation is minimal. It incurs no protection boundary crossing and no virtual memory context switch as the QoS mapper is executed as a supervisor actor and thus shares the kernel's address space.

The QoS mapper is also currently used as a repository of QoS related statistics of relevance to user-level library code when it detects QoS degradations. The primary statistic is the number of page faults incurred by a region associated with a B connection. This information is used to better inform the choice of which B regions to lock and which to leave unlocked.

V. Related Work

A large amount of work has been carried out on QoS support in networks but significantly less work has been done on *integrated* QoS support over all layers including the end-system. Several different ways of categorizing QoS guarantees have been identified in the network-level work. For example, in [12] a distinction is made between three different service commitments: 1) guaranteed service for real-time applications; 2) predicted service, which utilizes the measured performance of delays and is targeted towards continuous media applications; and 3) best effort service, where no QoS guarantees are provided. In our design, commitment is supported both in the network and in the end-system.

There have been a number of reported efforts in the area of resource reservation in network nodes. In particular, ST-II [30] was designed as a source initiated resource allocation framework for packetised audio and video communications across the Internet. RSVP [32] is a similar design, which offers receiver initiated reservation and multipoint-to-multipoint support. SRP [3], also designed for the Internet, supports both network and end-system resource allocation.

In the area of QoS-configurable transport systems, [31] describes a protocol intended to run over a network layer offering comprehensive QoS guarantees. The protocol offers QoS configurability and includes an algorithm for bounding buffer allocation given throughput and jitter bounds. The design uses a shared memory interface between user and protocol threads. However, scheduling issues were not addressed in this work. Another prominent QoS-driven transport protocol is TPX, which was designed under the Esprit OSI 95 project [5].

The HeiTS project [20] has investigated end-system issues in the integration of transport QoS and CPU scheduling. HeiTS puts considerable emphasis on an optimized buffer pool that minimizes copying and also allows efficient data transfer between local devices. The scheduling policy used is a rate monotonic scheme whereby the priority of the thread is proportional to the message rate accepted. The implementation environment of HeiTS, AIX, and OS/2, differs from our microkernel based environment.

A major influence on our work in the scheduling area is the split-level scheduling scheme described in [17]. However, in Govindan's scheme, there is no end-to-end QoS control and, although threads are appropriately scheduled once an application-level message has been received, the scheduling of protocol processing is controlled by a standard nonreal-time policy. Our scheme integrates the scheduling of protocol and application processing through the mechanisms of rthandlers and QoS-controlled connections. Govindan also describes a framework for interaddress-space communication known as memory mapped streams (MMS). MMS's are integrated with the scheduling system and work with a range of data transfer implementations such as copying, shared memory, or remapping. However, the abstraction is only applicable for intramachine communication. Our QoS-controlled connection abstraction performs a similar role but is applicable to remote as well as local communications. Our design as a whole also differs in that it incorporates guaranteed as well as best effort commitment.

Work on real-time extensions to the Mach microkernel, consisting of real-time threads, real-time synchronization primitives and time-driven scheduling, is described in [28]. The scheduling mechanism is derived from the ARTS kernel and permits hard real-time scheduling based on EDF. The main limitations of this work are the lack of API-level QoS specification and the lack of integration with the communications subsystem. As an example of the latter, the API provides means to create periodically executable threads, but there is no way to associate this periodicity with the arrival of messages on a Mach port. More recent work by the same group has addressed QoS issues, including QoS monitoring through the concept of *deadline handlers*, which are invoked when deadlines are missed [29].

VI. Conclusion and Future Work

We have described the design of a QoS-driven communications stack in a microkernel operating system environment. The discussion has focused on resource management aspects of the design and in particular we have dealt with CPU scheduling, network resource-management and memory-management issues. The architecture minimizes kernel-level context switches and exploits early demultiplexing so that incoming data can always be treated according to the QoS of its associated API-level connection. It also eliminates data copying on both send and receive (except for unavoidable copies to/from the ATM interface card). On send, the user's buffer is mapped to the lower layers, which process it *in situ*, and, on receive, the lower layers allocate a buffer and map it to the transport layer, which subsequently passes it to the application by passing the address of the buffer as an argument to an rthandler.

At the present time we are experimenting with an infrastructure consisting of three 486 PC's running Chorus and connected to an Olivetti ATM switch via ISA bus ATM interface cards. The PC's contain VideoLogic audio/video/JPEG compression boards as real-time media sources/sinks. The Olivetti switch is also connected to a wider ATM network consisting of Fore ASX100 switches. The current state of the implementation is that the API, split-level scheduling infrastructure, transport protocol and ATM card drivers are in place. In the next implementation phase we will refine the QoS-driven memory-management scheme and add heterogeneous networking with IP++ support.

There remain a number of important issues that we have yet to tackle. One is the need to synchronize real-time data delivery on separate application-related connections (e.g., for lip sync over audio and video connections). Along with our collaborators at CNET, Paris, we are currently investigating the use of real-time controllers written in the Esterel real-time language for this purpose [19]. Another issue, which is being addressed in a related project at Lancaster, is the requirement for QoS-controlled multicast connections. We already know how we can support multicast at the API level, but our ideas on engineering multicast support in the microkernel environment are still immature. A further issue is the incompleteness of the dynamic QoS management design. In particular, we would like to extend our design to include access latency bounds on swappable memory regions and also to accommodate comprehensive QoS monitoring and automated reconfiguration of resources in the event of QoS degradations.

Finally, we briefly report on our experiences with our ATM hardware, which is proprietary equipment supplied by Olivetti Research Labs, Cambridge, U.K. The hardware consists of PC ISA bus ATM interface cards and a "soft" switch, which runs a microkernel called ATMos and is fully programmable. While the link speed of the ATM equipment is 100 Mb/s, and the switch is capable of throughput of this order, the speed of the system as a whole is restricted by the fact that the ATM interface cards only support host/card data transfer at the granularity of ATM cells. Although this is a drawback in performance terms, the advantage is that it permits us to experiment with cell-level scheduling in the end-systems (see Section IV-D3) as well as in the switch [6]. Unfortunately, the card has architectural as well as performance implications for the rest of our design in that it dictates that SAR be carried out in kernel space to avoid the crippling overhead of a software interrupt/asynchronous system call per cell. We are thus forced to compromise our ideal strategy of a single, nonmultiplexed, user-level, per-connection thread operating all the way up/down the stack. Another problem is that the receive side AAL5 kernel thread in the network actor is impossible to schedule correctly as it must take "top" priority in order to ensure that cells are copied off the card as soon as possible. In light of these considerations, we intend to experiment in the future with an interface card that has on-board AAL5 and DMA for cell movement in order to realistically evaluate the performance potential of our design.

Acknowledgment

The authors would like to thank their colleagues at CNET, particularly J.-B. Stefani, F. Horn, and L. Hazard, for their close cooperation in this work. They would also like to acknowledge A. Mauthe for fruitful discussions on CPU scheduling and F. Ball for his contributions in the ATM networking area.

References

[1] V. Abrossimov, M. Rozier, and M. Shapiro, "Generic virtual memory management for operating system kernels," in *Proc. SOSP '89* (Litchfield Park, AZ), Dec. 1989.
[2] M. Accetta *et al.*, "Mach: A new kernel foundation for UNIX development," Tech. Rep., Dept. of Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, Aug. 1986.
[3] D. P. Anderson, R. G. Herrtwich, and C. Schaefer, "SRP: A resource reservation protocol for guaranteed performance communication in the Internet," Intern. Rep., Univ. of Calif. at Berkeley, 1991.
[4] *ATM User Network Interface Specification* Version 3.0, Sept. 1993.
[5] Y. Baguette, *et al.*, "TPX specification," in *OSI 95 Rep.* (Univ. of Leige, Belgium), ULg-5/R/V1, Oct. 92.
[6] F. Ball and D. Hutchison, "Traffic control in an ATM LAN," in *Proc. 2nd IFIP Workshop Performance Modelling, Evaluation of ATM Networks* (Bradford, U.K.), July 4–7, 1994.
[7] A. Bricker *et al.*, "Architectural issues in microkernel-based operating systems: The CHORUS experience," *Comput. Commun.*, vol. 14, no. 6, pp. 347–357, July 1991.
[8] A. Campbell, G. Coulson, F. Garcia, and D. Hutchison, "A continuous media transport and orchestration service," in *Proc. ACM SIGCOMM '92* (Baltimore, MD), Aug. 1992.
[9] A. Campbell, G. Coulson, and D. Hutchison, "A multimedia enhanced transport service in a quality of service architecture," in *Proc. Fourth*

Int. Workshop Network, Operating Syst. Support Digital, Audio, Video (Lancaster, U.K.), Oct. 93.
[10] A. Campbell, G. Coulson, F. Garcia, D. Hutchison, and H. Leopold, "Integrated quality of service for multimedia communications," in *Proc. IEEE Infocom '93* (San Francisco, CA), Mar. 1993.
[11] A. Campbell, G. Coulson, and D. Hutchison, "A quality of service architecture," *ACM Comput. Commun. Rev.*, Apr. 1994.
[12] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *Proc. ACM SIGCOMM '92* (Baltimore, MD), Aug. 1992, pp. 14–26.
[13] G. Coulson, G. S. Blair, P. Robin, and D. Shepherd, "Extending the Chorus micro-kernel to support continuous media applications," in *Proc. Fourth Int. Workshop Network, Operating Syst. Support Digital, Audio, Video* (Lancaster, U.K.), Oct. 93.
[14] G. Coulson and G. S. Blair, "Micro-kernel support for continuous media in distributed systems," *Comput. Networks, ISDN Syst.*, vol. 26, pp. 1323–1341, 1994.
[15] G. Coulson, G. S. Blair, P. Robin, and D. Shepherd, "Supporting continuous media applications in a micro-kernel environment," in *Architecture and Protocols for High-Speed Networks*, Otto Spaniol, Ed. Norwell, MA: Kluwer, 1994.
[16] D. Ferrari and D. Verma, "A scheme for real-time channel establishment in Wide Area Networks," *IEEE J. Select. Areas Commun.*, vol. 8, no. 3, Apr. 1990.
[17] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proc. Thirteenth ACM Symp. Operating Syst. Principles* (Pacific Grove, CA), SIGOPS, vol. 25, 1991, pp. 68–80.
[18] M. Hayter and D. McAuley, "The desk area network," *ACM Operating Syst. Rev.* , vol. 25, no 4, pp. 14–21, Oct. 1991.
[19] L. Hazard, F. Horn, and J.B. Stefani, "Notes on architectural support for distributed multimedia applications," Rep. CNET/RC.W01.LHFH.001, Centre National d'Etudes des Telecommunications, Paris, France, Mar. 1991.
[20] D. B. Hehmann, R. G. Herrtwich, W. Schulz, T. Schuett, and R. Steinmetz, "Implementing HeiTS: Architecture and implementation strategy of the Heidelberg high speed transport system," in *Proc. Second Int. Workshop Network, Operating Sys. Support Digital, Audio, Video* (Heidelberg, Germany), 1991.
[21] K. Jeffay, D. Stone, and F. Donelson Smith, "Kernel support for live digital audio and video," in *Proc. Second Int. Workshop Network, Operating Syst. Support Digital, Audio, Video* (Heidelberg, Germany), 1991.
[22] J. F. Kurose, "Open issues and challenges in providing quality of service guarantees in high-speed networks," *ACM Comput. Commun. Rev.*, vol. 23, no. 1, pp. 6–15, Jan. 1993.
[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. Assoc. Computing Mach.*, vol. 20, no. 1, pp. 46–61, Feb. 1973.
[24] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First class user-level threads," in *Proc. Symp. Operating Syst. Principles (SOSP)* (Asilomar, CA), Oct. 1991, pp. 110–121.
[25] A. C. Scott, W. D. Shepherd, and A. Lunn, "The LANC—Bringing local ATM to the workstation," in *Proc. Fourth IEE Telecommun. Conf.* (Manchester, U.K.), Aug. 1992.
[26] A. S. Tanenbaum, R. van Renesse, H. van Staveren, and S. J. Mullender, "A retrospective and evaluation of the Amoeba distributed operating system," Tech. Rep., Vrije Universiteit, CWI, Amsterdam, The Netherlands, 1988.
[27] D. L. Tennenhouse, "Layered multiplexing considered harmful," in *Protocols for High-Speed Networks*. Amsterdam, The Netherlands: Elsevier, 1990.
[28] H. Tokuda, Y. Tobe, S. T. C. Chou, and J. M. F. Moura, "Continuous media communication with dynamic QOS control using ARTS with an FDDI network," *ACM Comput. Commun. Rev.*, 1992.
[29] H. Tokuda and T. Kitayama, "Dynamic QOS control based on real-time threads," in *Proc. Fourth Int. Workshop Network, Operating Syst. Support Digital, Audio, Video* (Lancaster, U.K.), Oct. 93.
[30] C. Topolcic, "Experimental Internet stream protocol, version 2 (ST-II)," *Internet Request for Comments No. 1190*, Rep. RFC-1190, Oct. 1990.
[31] B. Wolfinger and M. Moran, "A continuous media data transport service and protocol for real-time communication in high speed networks," in *Proc. Second Int. Workshop Network, Operating Syst. Support Digital, Audio, and Video* (Heidelberg, Germany), 1991.
[32] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation protocol," *IEEE Network*, Sept. 1993.
[33] S. Deering, "Simple Internet protocol plus (SIPP) specification," Internet Draft ⟨draft-ietf-sipp-spec-01.txt⟩, Aug. 1994.
[34] A. Mauthe, "Scheduling considering jitter requirements of continuous media tasks," Intern. Rep. MPG-94-19, Computing Dept., Lancaster Univ., Bailrigg, Lancaster, U.K., May 1994.

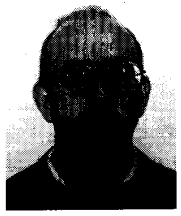
Geoff Coulson received the first class honors degree in computer science and the Ph.D. degree in systems support for multimedia applications from the University of Lancaster, Lancaster, U.K.

He has recently been appointed to a Lectureship at the University of Lancaster and is currently working on the SUMO project, investigating operating system support for continuous-media communications and application support. His research interests are distributed-systems architectures, multimedia communications, and operating-system support for continuous media.

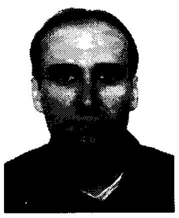
Andrew Campbell was a Consultant with Raytheon and GTE in the United States, developing solutions for mobile, packet-switched, and tactical communication systems, before joining the Distributed Multimedia Research Group, Lancaster University, Lancaster, U.K. He was recently appointed to a British Telecom Research Lectureship in the Department of Computing, Lancaster University, and is currently a Visiting Scholar at the Center for Telecommunications Research, Columbia University, New York, NY. His research interests include networking, protocol design, operating systems, and quality-of-service architecture.

Philippe Robin received the D.E.S.S. degree in computer sciences from PARIS-XI (ORSAY) University.

From 1989 to 1992, he was with Chorus Systemes, Paris, France, involved in the development of the Chorus/MiX operating system. He was also in charge of the Chorus sales and support for Europe. Later he worked on the development of an i486 fault-tolerant platform using Chorus/MiX. He is now with the Distributed Multimedia Research Group, Lancaster University, Lancaster, U.K., working on operating-system support for continuous media and quality-of-service management on Chorus.

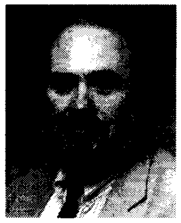


Gordon Blair is currently a Senior Lecturer in the Computing Department, Lancaster University, Lancaster, U.K., and has been actively involved in research in distributed systems. Following the completion of the Ph.D. degree, he was an SERC Research Fellow at Lancaster University, before being appointed to a lectureship post in 1986. He has been responsible for a number of research projects there in the areas of distributed systems and multimedia support and has published over 70 papers in his field. His current research interests include distributed-multimedia computing, operating-system support for continuous media, the impact of mobility on distributed systems, and the use of formal methods in distributed-system development.



Michael Papathomas received the M.Sc. ("diplo^me d'informatitien") and Ph.D. ("Doctorat e`s Sciences mention informatique") degrees in computer science from the University of Geneva, Switzerland, in 1985 and 1992, respectively.

He is currently a Visiting Research Fellow at Lancaster University, Lancaster, U.K., supported by a grant from the Swiss FNRS ("Fonds National de Recherche Scientifique"). His research interests include pragmatic and formal aspects of concurrent and distributed systems, object-oriented programming, and multimedia.



Doug Shepherd has been Professor of Computing and Director of Information Systems Policy at Lancaster University, Lancaster, U.K. His current research interests include distributed operating systems, multimedia storage systems, and high-performance multimedia protocols.

Dr. Shepherd was Chairman of the 4th Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster University, November 1993.