# Implementing a QoS Controlled ATM Based Communications System in Chorus

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

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Lancaster,
LA1 4YR,
UK.
(Internal Report MPG-94-05)

telephone: +44 (0)524 65201
e-mail: [pr, geoff, campbell, gordon, michael]@comp.lancs.ac.uk

***ABSTRACT***

*In this paper we describe the design of a QoS driven communications stack in a micro-kernel operating system environment. The paper focuses on resource management aspects of the design and in particular we deal with CPU scheduling, network resource management and memory management issues. We describe an API at which QoS parameters are supplied by users and present an architecture able to guarantee these QoS requirements with varying degrees of commitment. The architecture includes modules to translate user level QoS parameters into representations usable by the scheduling, network and memory management subsystem. It also incorporates admission tests which determine whether or not a new connection can be accepted given a particular QoS requirement and resource availability. In addition to predictable resource allocation the architecture also incorporates dynamic QoS management which ensures that ongoing commitments continue to be met in the face of variable loading.*

## 1. Introduction

In the recent past, significant research has been carried out into communications systems for distributed real-time and multimedia applications. A surprisingly small amount of this work, however, has considered the issues that arise when high-speed communications systems are interfaced to *conventional* workstations running standard multiprogrammed operating systems such as UNIX. Rather, the research has tended to focus on network issues [Kurose,93] or has made specific assumptions about the end points of multimedia and real-time communication. Some researchers have considered abstract transport protocol specifications or service interfaces [Baguette,92]. Others have assumed specialised end-systems such as CODECS or multimedia enhancement units [Hayter,91], [Scott,92]. Still others have considered computers running specialised real-time operating systems unable to support conventional applications or conventional modes of operation (e.g. such systems typically preclude dynamic process creation) [Jeffay,91].

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

- 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 micro-kernel operating system, specifically Chorus [Bricker,91], as a system software support layer for 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 area of distributed real-time and multimedia application support has concentrated on API issues [Coulson,93a], CPU scheduling issues [Coulson,93b], transport issues [Campbell,93] and network architecture [Campbell,94]. 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*. Note that 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 in more depth in [Campbell,94].

The paper begins by providing, in section 2, some necessary background material on Chorus. Next we present, in section 3, 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 which minimises 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 per-machine connection and resource management.

We then investigate the management of CPU, communications and memory resources in this architecture. Section 4 deals with static aspects of resource management such as the derivation of resource quantities from QoS parameters (QoS translation), and admission testing. Section 5 deals with dynamic aspects of resource management such as scheduling, ensuring bounded latency access to memory, and time constrained protocol operation. Finally, we discuss some related work in section 6 and offer concluding remarks in section 7.

## 2. Background on Chorus

Chorus is a commercial micro-kernel based operating system which supports the implementation of conventional operating system environments through the provision of 'sub-systems' (for example a sub-system is available for UNIX SVR4 as mentioned above). The micro-kernel is implemented using modern techniques such as multi-threaded address spaces and inter-process communication with copy-on-write semantics. The basic Chorus abstractions are *actors*, *threads* and *ports*, all of which are named by globally unique and globally accessible identifiers. Actors are address spaces and containers of resources which may exist in either user or supervisor space. Threads are units of execution which run code in the context of an actor. By default, they are scheduled according to either a pre-emptive priority based scheme or round robin timeslicing. Ports are message queues used to hold incoming and outgoing messages. Inter-process communication is datagram based and supports both request/reply and single shot messages.

Chorus has several desirable real-time features and has been widely used for embedded real-time applications. Real-time features include pre-emptive scheduling, page locking, timeouts on system calls, and efficient 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,
- the exclusive use of connectionless communications makes it impossible to pre-specify communications resource allocation,
- the above two points are particular instances of a more general point: that there is no possibility of *guaranteeing* the QoS of activities.

Note, however, that the above limitations are not unique to Chorus: they are shared by most of the other micro-kernels in current use (e.g. [Accetta,86], [Tanenbaum,88]).

# 3. Architecture

## 3.1 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 figure 1 and described below.

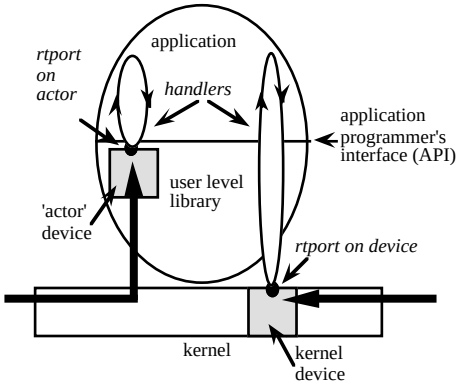



*Figure 1: Devices, Rtports and Handlers*

- *rtports*: these are extensions of standard Chorus ports and serve as access points for real-time communications. Rtports have an associated QoS which defines constraints on communication. They also provide direct access to buffers by the application thus minimising copy operations.
- *devices*: these are producers, consumers and filters of real-time data which support the creation of rtports and provide the memory for their buffers. One special type of device is the *null* device which is 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 which 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. They encourage an event-driven style of programming which is appropriate for real-time applications and also avoids the additional context switches associated with a traditional send()/ recv() based interface.
- *QoS controlled connections*: these are abstract communication channels with a specific QoS. A connection is established between a source and a sink rtport according to a given QoS specification. There are two types of connection: 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 facilities for dynamically re-negotiating the QoS of an open connection. It also allows pipelines of 'software signal processing' modules to be configured for local continuous media processing. Full details of the continuous media API are specified in [Coulson,93a].

### 3.2 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 which, in this context, we refer to as a *virtual processor* (VP). The scheduling architecture is a *split level* structure [Govindan,91] consisting of a single kernel scheduler (KLS) to schedule VPs, and per-actor user level schedulers (ULSs) to schedule lightweight threads on those VPs (see figure 2).

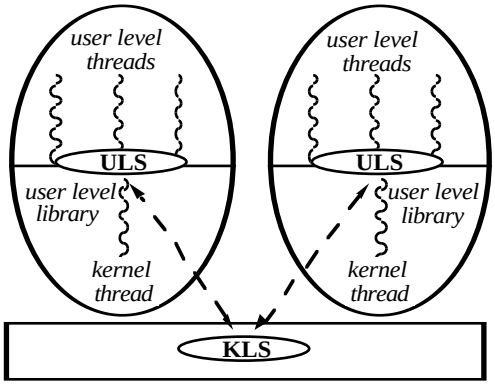



*Figure 2: Split level scheduling architecture*

The advantage of lightweight threads and user level scheduling is that context switch overhead is minimal. 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:-

i) each ULS always runs its most urgent lightweight thread,

ii) the KLS always runs the VP supporting the *globally* most urgent lightweight thread.

Note that 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 4.3.1.

The necessary information exchange between the KLS and the ULSs is accomplished via a combination of shared memory and upcalls from the kernel [Govindan,91]. The per-VP shared memory areas contains the urgency of the most urgent lightweight thread known to this VP. This is read by the KLS on each kernel level rescheduling operation to determine the next VP to schedule.

A passive shared memory area between the KLS and the ULSs is not sufficient. It is also necessary for the KLS to be able to actively interrupt VPs to inform them of the occurrence of real-time events in a timely fashion. Such events include timer expirations used to implement pre-emption in user level scheduling, and data arrivals from local kernel devices or from the network device. We use a *software interrupt* mechanism for the notification of such events. Software interrupts are always targeted at application actors but can be initiated either by kernel components (e.g. the KLS) or by library code in other application actors.

The last major component of the scheduling architecture is the use of *non-blocking system calls* [Marsh,91] to avoid potential violations of the scheduling invariants. With traditional

blocking system calls such violations can occur when a lightweight thread performs a blocking system call which blocks its underlying VP. This disallows another lightweight thread in the same actor from executing while the blocking call is extant - even if it is the globally most urgent thread. Non blocking system calls avoid this problem by returning immediately and thus allow the calling VP to run another lightweight thread while the system call is being serviced. The result of the call is eventually notified by a software interrupt as discussed above.

### 3.3 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 figure 3) is intended to support connection oriented communications over ATM networks with QoS support and configurable error control. Because of these distinct design goals, we have initially designed our stack to operate entirely separately from the existing Chorus facilities although we intend in the future to integrate the functionality of the two stacks in a unified architecture.

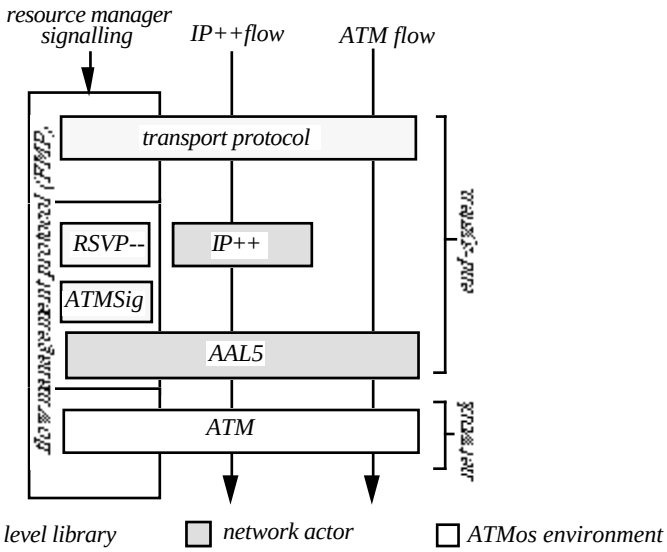



*Figure 3: Communications architecture*

#### 3.3.1 Abstract Layering

The central component in the communications architecture is the *flow management protocol* (FMP) [Campbell,94]. This provides a framework for the management of network resources in both local and wide-area networks using out-of-band signalling. The FMP has two main functions: i) to request the allocation of CPU and memory resources on remote end-systems, and ii) to manage resources in the network itself. The FMP uses a specialised signalling protocol stack consisting of a subset of RSVP [Zhang,93] which we encapsulate in ICPM, and an ATM signalling protocol, called ATMSig, which is based on a subset of the ATM Forum's UNI 3.0 [ATM,93].

The user data stack is positioned alongside the FMP's signalling stack. The upper architectural layer consists of a connection oriented, rate based, transport protocol [Campbell,92a]. This provides facilities for QoS configuration, performance monitoring, notification of QoS degradations, and in service QoS re-negotiation. Admission control, resource reservation, and QoS maintenance for the protocol are supported by the scheduling and memory management architectures described in this paper (see sections 4 and 5).

The 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) [Deering,93]. In particular, IP++ uses a packet header field called a *flow-id* to identify IP packets as belonging to a particular *flow* or connection, and a *flow-spec* (see section 4.3.1) to define the QoS associated with each flow.

Flow-specs are held by IP++ routers[1] 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 initialised at connection set up time by the FMP. Below the IP layer we use an ATM Adaptation Layer service to perform segmentation and reassembly of IP packets into/from 53 byte ATM cells. We use a minimal AAL5 service for this purpose.

The lowest layer of our architecture is based on the Lancaster Campus ATM network. This delivers ATM to a mix of workstations, PCs, and multimedia devices designed at Lancaster [Scott,92]. It also interconnects a number of Ethernets and interfaces to the rest of the UK via the SuperJANET 100 Mpbs Joint Academic Network. The PCs which run the system described in this paper are directly connected to ATM switches manufactured by Olivetti Research Limited (ORL) via 4x4 ORL ATM interface cards. The ORL ATM switches are implemented using 'soft' switching and run a small micro-kernel called *ATMos* as identified in figure 3.

### 3.3.2 Realisation in 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*[2]. The transport layer of the FMS is implemented in the connection manager actor described in section 3.5. The transport layer of user applications is implemented in the same user level library[3] that supports the API abstraction discussed in section 3.1. This is so that its service interface can be provided by the library level rtport and rthandler abstractions defined in that section. The transport protocol communicates with the network actor via *system calls* for send side communications, and *software interrupts* for receive side communications.

Below the transport protocol, the rest of the communications architecture, including the ATM card device driver, is implemented in the network actor. 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 or the sink of a flow but is merely routing from one network to another. In this case, CPU and memory resources are dedicated to flows on the basis of a flow-spec supplied by the FMP (see section 4.4). Otherwise, the function of the IP++ layer is effectively null.

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 [Tennenhouse,90]. Currently, the maximum service data unit size for the AAL5, IP++ and transport layers alike is restricted to 64Kbytes. This means that no further segmentation/ reassembly is required above the AAL5 layer[4]. The ATM cards generate an interrupt every time a cell is received, and every time they are ready to transmit. Communication between the interrupt service routines and the per-flow AAL5 threads is via Chorus mini-ports which are specially optimised for this purpose.

## 3.4 Memory Management Architecture

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

---

1 Flow specs are also used by the FMP to control resource reservation at the ATM level in ATM switches.

2 Note that this is distinct from the standard Chorus 'network actor' which is also called the 'Network Device Manager'.

3 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.

4 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.

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 which are policed by the VM layer.
- mappers are supervisor actors which 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 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 non predictable memory access latency due to arbitrary page faulting[1]. 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 applications with memory regions offering latency bounded access,
- determining whether or not requests for QoS controlled memory resources should succeed or fail,
- pre-empting QoS controlled memory from ‘low urgency’ threads on behalf of ‘high urgency’ threads, and,
- re-mapping QoS controlled memory regions from one actor to another.

In addition to servicing requests from the kernel VM layer, the QoS mapper is accessed from the user level libraries implementing the connection abstraction. In particular, user level code invokes 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.

### 3.5 Connection Management Architecture

Connection management in a QoS driven system is more than the establishment of communication between remote machines. It is also the task of connection management to arrange for the allocation of suitable CPU, network and memory resources according to the specified QoS of the requested connection. This includes partitioning the responsibility for QoS support among individual resource managers. For example, the connection manager partitions the API level *latency* QoS parameter (see section 4.1) between the network and the CPU resource managers on each end system.

The architecture of our connection management scheme 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 4) in the user level library determines the resource requirements of new connection requests. The output of the QoS translator is then directed to a per-machine *connection manager* actor which performs admission testing and resource allocation for all required resources. The connection manager must communicate with each of the CPU, network and memory resource managers independently, and only if all are able to provide the required resource commitment can the connection request be granted.

---

1 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.

The connection manager has to deal with both local and remote CPU and memory resources. The necessary signalling and communication functionality for the remote case is provided by the FMP described above. Figure 4 illustrates the relationship between the connection manager and the FMP (which is situated in the network actor). The figure shows two separate sites, each with its own connection manager.

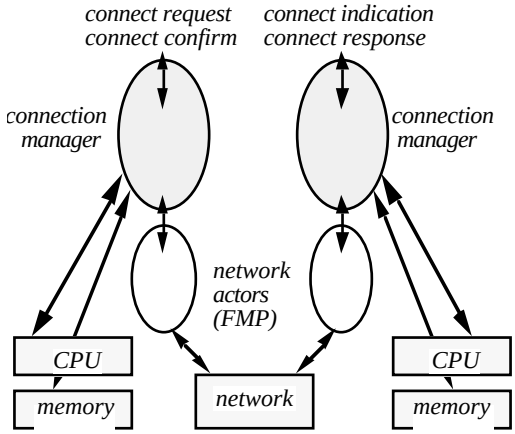



*Figure 4: Connection management architecture*

## 4. Static 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 to make appropriate reservations.

In general, 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 fulfil those requirements.

### 4.1 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 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 honoured 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 to control resource pre-emption for connections. All things being equal, a connection with a low

priority will have its resources pre-empted 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 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 before/ after the rthandler is called. To determine the quantum value, the infrastructure performs a 'dummy' upcall of the handler and measures the time taken for it to return (a boolean flag is used to let the application code in the rthander 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 (error_interval is not applicable to message connections).

For the stream service, 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 10ms implies that buffers may be delivered up to 5ms either side of the nominal buffer delivery time. *Delivery* also refines the meaning of buffrate. If *isochronous* delivery is specified, the stream service delivers 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.

## 4.2 The CPU Resource

### 4.2.1 QoS Translation

We distinguish three major classes of QoS controlled connection for resource management purposes which are determined by the *commitment* and *delivery* QoS parameters. The three classes are:-

- $G_I$: commitment = guaranteed; delivery = isochronous,
- $G_W$: commitment = guaranteed; delivery = workahead, and
- B: commitment = best effort.

The semantics of thread scheduling for each of the three classes are as follows:-

- $G_I$: threads for these connections are scheduled to run *non-preemptively* for each quantum and have their execution quanta statically allocated along the future time line. An admission test is performed to ensure that this static allocation is feasible.
- $G_W$: these are scheduled according to the preemptible earliest deadline first (EDF) policy [Liu,73]. Again, an admission test is performed.
- B: these are scheduled according to the preemptible earliest deadline first policy but no admission test is used. This class is preemptible by both $G_I$ and $G_W$ threads.

B threads are further subdivided into three subclasses: i) those requiring isochronous

delivery, ii) those requiring workahead delivery and iii) all other non real-time threads in the system (e.g. Chorus server threads or non real-time application threads). The differences between the treatment of these subclasses is minimal and is described in section 5.1.

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 4.1. Figure 5 illustrates the notions of period and quantum together with the related scheduling concepts of *scheduling time*, *deadline* and *jitter*.

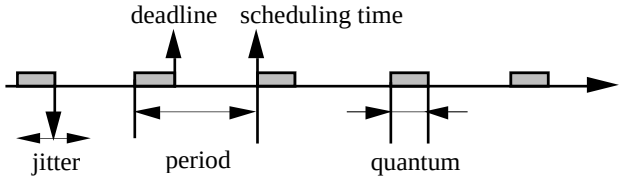



*Figure 5: Periodic thread scheduling terminology*

For message connections, periodic *sporadic server* threads are used at the receive side[1]. One sporadic server is provided for each of the two commitment classes 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. The period of each server is derived as follows:-

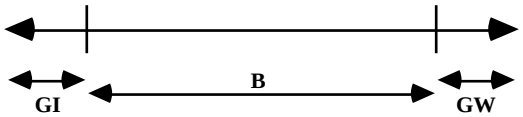
$$period = \min(recv\_latency_0, \ ..., \ recv\_latency_n)$$

where $recv\_latency_i$ is the proportion of the total end-to-end latency allocated by the connection manager to the receive end-system for message connection *i*. These period and quantum values ensure that whenever a buffer arrives on a message connection there will always be sufficient CPU resource available to service the connection within the permissible latency bound. Of course, if two messages arrive simultaneously then one must wait, but this is inevitable on a uniprocessor. On a multiprocessor machine, per processor sporadic servers could be provided.

### 4.2.2 Admission Testing

Each of the three classes of thread is allocated a proportion of the total CPU resource available as illustrated in figure 6. The vertical lines represent ‘firewalls’ between the classes. Note that firewalls are used 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 classes at run time (see section 5.1).

The firewalls can be dynamically altered at run time by the programmer, but a typical configuration will allow a relatively small allocation for $G_I$ and $G_W$ 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 - in particular, guaranteed isochronous threads should only be used for connections which are delivering data to an end device such as a frame buffer or audio chip.




[1] There is no thread implicitly associated with the source side of message connections. As pointed out in section 3.1, 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.

*Figure 6: Partitioning of CPU resource*

The admission tests for $G_I$ threads are:-

i) $$\sum_{i=1}^{N_{G_I}} \frac{quantum_i}{period_i} \leq R_{G_I}, \quad 0 \leq R_{G_I} \leq 1$$

ii)
```
pos := 0, q := 0, dropped := 0;
length := LCM(periods of all GI threads)
while(pos < length) {
      adjust := 0;
      while (try to place quantum q on the timeline
         at position pos + adjust or position pos - adjust
         without a clash == FAIL) {
            adjust := adjust + system clock tick;
            if (adjust > jitter_QoS) {
                  dropped := dropped + 1;
                  break;
            }
      }
      pos := pos + period_QoS;
      q := q + 1;
}
if (dropped > error_QoS * (length / error_interval_QoS))
      return FAIL;
else
      return SUCCEED;
```

$G_I$ threads have their CPU resources 'pre-allocated' along the future time line and always run their quanta to completion without being pre-empted. The admission test for this class, as illustrated above, is a two stage process. Firstly, it is established whether or not sum of the candidate thread's quanta can be accommodated within the allocated portion of the total CPU resource. This test is directly taken from [Liu,73]. $N_{G_I}$ refers to the total number of $G_I$ threads in the system and $R_{G_I}$ refers to the proportion of CPU resources dedicated to this class of threads (such that $R_{G_I} + R_{G_W} + R_B = 1$ where $R_{G_W}$ and $R_B$ respectively represent the portion of the CPU resource dedicated to $G_W$ and B threads ).

The second part of the test establishes whether or not it is possible to fit the quanta onto the time line without clashing with the previously positioned quanta of previously allocated $G_I$ threads. To ease this latter task, we exploit the degrees of freedom implied by the *error* and *jitter* QoS parameters so that if we are attempting to place a quantum on the time line and there is a clash with a quantum from a previously allocated connection, we attempt to fit the quantum by sliding it within the bounds permitted by the jitter parameter (note that the granularity of 'sliding' is determined by the granularity of the system clock tick). If it is impossible to accommodate a particular quantum, then that quantum can even be dropped as long as this does not violate the error QoS parameter over any single error interval. The algorithm embodied in the above pseudocode illustrates this process. LCM() is the lowest common multiple which is used to determine the shortest time line without repetitions. Note that, in principle, the above algorithm could be refined to attempt to move quanta from existing threads when fitting in new quanta. However, any gains in this direction would be offset by the increased time and CPU effort required as the strategy begins to approach exhaustive search.

For $G_W$ threads the admission test is:-

$$\sum_{i=1}^{N_{G_W}} \frac{quantum_i}{period_i} \leq R_{G_W} - S, \quad 0 \leq S \leq R_{G_W}$$

This is again a standard EDF admission test with the addition of a 'safety factor' S which ensures that the $G_W$ CPU share is not fully allocated. This is used for two reasons: i) a certain amount of resource surplus is required so that workahead can, in fact, take place, and ii) the EDF algorithm is known to degrade badly as the CPU becomes saturated [Liu,73]. The value of S can either be fixed statically or arrived at empirically at run time.

For B threads there is no admission test. Admission tests for the sporadic servers are identical to those for $G_W$ and B periodic threads. Each time a new message connection is created which 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.

## 4.3 The Network Resource

### 4.3.1 QoS Translation

The network sub-system offers guarantees on *bandwidth*, *packet loss* and *delay bounds*. 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;
   int         mtu_size;
   int         rate;
   int         delay;
   int         loss;
} flow_spec_t;
```

*Flow_id* uniquely identifies the network level flow. It corresponds to the virtual circuit identifier at the AAL5/ATM level and the flow id in the IP++ packet header. *Mtu_size*[1] 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 which has been allocated, by the connection manager, to the network. It subsumes both propagation and queuing delays in the network. Finally, *loss* is an upper bound probability of cell loss due to buffer overflow at switches. Loss is calculated from the *error* and *error_interval* API level QoS parameters and is equal to 1 - *error*/ *error_interval*.

### 4.3.2 Admission Testing

In the network, only two traffic classes are recognised: *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 left over.

For guaranteed flows, three admission tests are performed by the FMP at each switch along the chosen path: a banwidth test, a delay bound test and a buffer availability test. If, at the current switch, the admission control tests are successful, the necessary resources are allocated. Then the FMP protocol entity in 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 realised.

If the required QoS is realisable, the FMP at the remote end-system returns a confirmation message to the initiating end-system. As it traverses the same route in reverse, the FMP *relaxes*

---

[1] 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 bytes and in the case of IP++ packets is 64Kbytes. 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.

any over-allocated resources at intermediate switches [Anderson,91].

**Bandwidth Test** The bandwidth test consists in verifying that enough processing 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 4.2.2 the admission control test is based on [Lui,73]:-

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

Here, $t_i$ refers to the service time of flow $i$ in the current switch, where there are $N$ flows and $rate_i$ is the rate of the $i$'th flow. $R$, $0 \leq R \leq 1$, represents the portion of resource dedicated to guaranteed flows.

**Delay Bound Test** The delay bound test determines the minimum acceptable delay bound which 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 [Ferrari,90]:-

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

Here, $d$ is the delay incurred at the current switch. $N$ represents the number of flows in a set $U$ where $U$ contains those flows whose local delay bound is lower than the service times of all flows supported by the current switch. $T$ represents the largest service time of all flows in a set $V$ where $V$ is the complement of set $U$. A full proof of this theorem can be found in [Ferrari,90]

The second phase calculation is:-

$$\sum_{n=1}^{N} d_n \leq delay$$

This merely requires that sum of the delays at each router is less than the delay parameter of the flow spec.

**Buffer Availability Test** The amount of memory allocated to a new flow traversing a router must be sufficient to buffer the flow for a period which is greater than the combined queuing delay and service time of its packets. The calculation for buffer space is:-

$$buffersize = mtu\_size\lceil d \;\; rate \;\; loss \rceil$$

Here, *buffersize* represents the amount of memory that may 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.

## 4.4 The Memory Resource

### 4.4.1 QoS Translation

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

**Number of buffers** To calculate the buffer requirement, the *buffsize*, *buffrate* and *jitter* QoS parameters are used. It is also necessary to take into account the delay bound offered by

the flow management protocol. The latter will typically permit a larger degree of jitter than the API level jitter bound and any discrepancy will 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:-

$$buffers = buffrate(delay + quantum + \frac{jitter}{2})$$

In this formula, *quantum*, *jitter* and *buffrate* are API level QoS parameters. *Delay* is the delay bound specified in the network level flow spec.

Only one buffer is required at the sender due to the nature of the rate based transport protocol: each buffer is assumed to be 'on the wire' before the start of the next period.

**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 i) the delay due to the RPC communication between the VM layer and the mapper, and ii) the delay associated with the external swap device[1]. 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 commitment and delivery QoS parameters as follows:-

- $G_I$: buffer regions allocated to these connections are locked and non-preemptible.
- $G_W$: buffer regions for these connections are locked but are 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 pre-emption 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 which are initially passed to the *rgnAllocate()* system call and retained to validate future requests on regions.

### 4.4.2 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 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 non real-time applications as well as best effort connections.

If no physical memory is available to fulfil 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 lowest priority connection 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.

---

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

# 5. Dynamic Resource Management

Dynamic resource management is concerned with QoS *maintenance*, *monitoring*, *policing* and *re-negotiation* [Campbell,92b]. The role of the *maintenance* function is to actually achieve the requested levels of QoS given the resources dedicated at resource allocation time - e.g. by providing access to allocated memory or by the application of a particular scheduling mechanism. The role of *monitoring* is to observe the maintenance function in a continuous observe-compare-adjust feedback loop and make any fine grained corrections necessary to keep performance within the tolerances permitted by the QoS specification. It is also the responsibility of the monitoring function to report to the higher layers when QoS maintenance breaks down. The *policing* function is intended to ensure that users do not attempt to use more resource than they have been allocated. This is largely a network issue and is concerned with preventing users injecting data into the network at a higher rate than that agreed at resource reservation time. Policing is a simple matter with CPU and memory resources as these are always protected by the kernel and cannot be 'stolen' by applications. The final function, *re-negotiation*, is initiated by the higher layers to dynamically modify the QoS specification of a live connection. This function is often initiated when performance has irretrievably broken down and must be re-established at a lower QoS.

Dynamic resource management in its full generality is a large area which we have only begun to explore. In this section we restrict ourselves mainly to a discussion of QoS maintenance issues in the domain of each of the three major resource classes.

## 5.1 Scheduling

Although the admission testing procedures described in section 4.3.2 only refer to the EDF scheduling policy, QoS maintenance in the CPU resource domain actually uses a combination of schedule times, deadlines and priorities[1] to capture the abstract notion of 'urgency'. The $G_I$ class is given a single highest priority band (only critical Chorus server threads such as the pager daemon are higher), the $G_W$ class the next highest priority band and the B class a range of priority bands all of which are less than those of the guaranteed classes. 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. This means that a $G_W$ thread runs when it is the most urgent runnable $G_W$ thread and there are no runnable $G_I$ threads, and B threads can only run when there are no runnable $G_I$, $G_W$ or higher priority B threads. 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.

In addition to enjoying high priority as a *class*, each *individual* $G_I$ thread will necessarily be able to run, and not collide with other $G_I$ threads, immediately it reaches its scheduling time (as determined by the static time line built up by the admission test of section 4.3.2). The scheduling time information required by ULSs for $G_I$ threads is stored by the KLS as a vector of numbers representing positions along a segment of the time line as calculated by the admission test. Of course, for $G_I$ threads to be actually runnable at their schedule time it is necessary that the other two resource classes - memory and network - are also able to meet their commitments[2].

The above scheme seems to give B threads a very poor chance of obtaining CPU service. However, the admission control scheme for the $G_I$ and $G_W$ classes has already assured that ample CPU time will be left for the B class. As explained in section 4.3.1, class B is subdivided into three subclasses. Class B threads of the isochronous variety (class B(i)) only become schedulable at a time indicated by the deadline minus the quantum time. This

---

1 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 which is not visible or directly manipulable from the API level.

2 It is also possible for threads - even $G_I$ threads - to be pre-empted and delayed by device interrupts which run as non schedulable activities. This is probably unavoidable in a general workstation environment.

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. Conversely, the scheduling time for workahead threads (class B(ii)) is always *now* (of course, such threads will block if the data they require is not yet available and so will not be permanently runnable).

Class B(iii) includes non real-time threads - e.g. those from conventional UNIX applications. These threads 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 class B(iii) threads which are fulfilling an interactive role would have relatively high priority which may be greater than that of B(i) and B(ii) threads (both of which will be in the same priority band). Other B(iii) threads, such as compute bound applications and non time critical daemons, will have priorities lower than the B(i) and B(ii) classes.

### 5.2 Dynamic Communications Management

A major aspect of dynamic resource management in the communications stack is the interleaving of ATM cells at the network interface from different connections on the basis of their QoS [Campbell,94]. Fortunately, because of the low level nature of our ATM interface cards we have been able to investigate this issue in software.

The receive side cell processing is simple. The receiver interrupt service routine[1] reads the VCI of the current cell while it is still on the card. The interrupt service routine then dispatches a receiver thread to copy the cell payload into a partially assembled AAL5 packet, and when the receiver thread sees the last cell of an AAL5 packet, it raises a software interrupt to the destination 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 i) the location in its address space into which the buffer has been mapped and ii) the deadline of the buffer (which is the end of the quantum of the transport protocol thread.

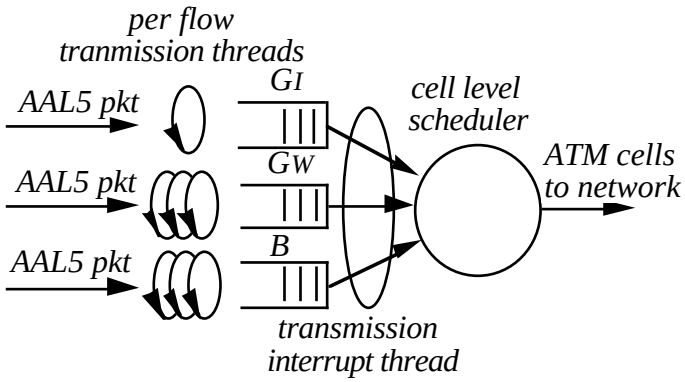



*Figure 7: Cell level scheduler*

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 another cell can be copied to the card for transmission. The scheduler chooses to run one of a number of per-connection[2] transmit

---

1 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.

2 Actually, only *one* thread is required for *all* threads in the $G_I$ class because the $G_I$ admission algorithm has

threads by sending a message to a mini-port on which the thread is waiting (see figure 7). The choice of thread to activate is made on the basis of the connection classes. Thus a $G_W$ thread will only run if there is no $G_I$ thread ready, and a B thread will only run if there are no ready threads from either of the guaranteed classes. Within classes, threads are scheduled on the basis of the deadline of the next cell in the thread's associated buffer. Cell deadlines are derived by giving each cell in the buffer a fixed temporal offset from the deadline of the entire buffer.

### 5.3 Dynamic Memory Management

The only dynamic QoS mapper function we have yet considered in detail is the region re-mapping function. This is used when buffers are mapped from one actor to another in a QoS controlled connection between local rtports. Region re-mapping is particularly important in the context of *pipelines* which arise when applications are structures as chains of modules which sequentially process a stream of real-time data. Pipelines can be implemented either within single actors or across multiple actors (or, indeed, across multiple machines although it is only the intra-machine 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; VmAddr address; VmSize size);
```

The raiseEvent() system call specifies a destination actor and details of the memory region to be remapped. When the kernel receives this system call, it invokes the QoS mapper which unmaps the region from the source address space and maps it into the destination address space. The QoS mapper then forwards the software interrupt to the target VP, passing as an argument the virtual address at which the region is mapped (see figure 8).

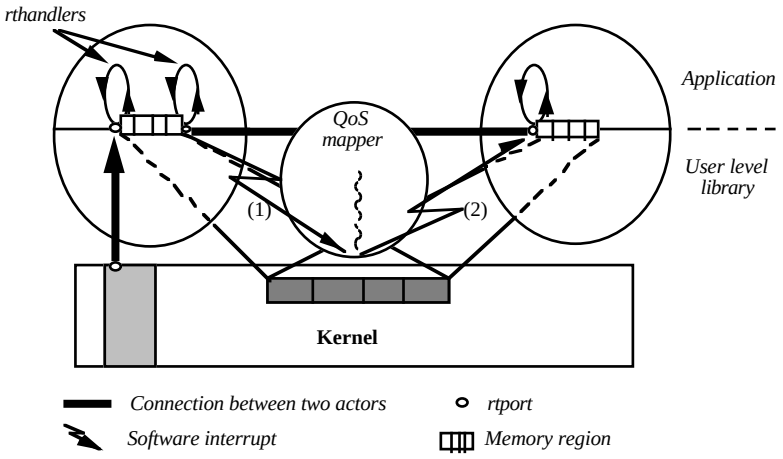



*Figure 8: Pipeline example*

The extra cost due to the QoS mapper invocation is minimal as the QoS mapper is executed as a supervisor actor (i.e. context switches are cheap as no protection boundary is crossed between the kernel and the QoS mapper).

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.

---

ensured that the quanta of these threads do not overlap and can thus be processed sequentially (see section 4.3.2).

## 6. 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 categorising QoS guarantees have been identified in the network level work. For example, in [Clark,92] a distinction is made between three different service commitments: i) guaranteed service for real-time applications; ii) predicted service, which utilises the measured performance of delays and is targeted towards continuous media applications; and iii) 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 [Topolcic,90] was designed as a source initiated resource allocation framework for packetised audio and video communications across the Internet. RSVP [Zhang,93] is a similar design which also offers receiver initiated reservation and multipoint-to-multipoint support. SRP [Anderson,91], also designed for the Internet, supports both networks *and* end-system resource allocation.

In the area of QoS configurable transport systems, [Wolfinger,91] 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 [Baguette,92].

The HeiTS project [Hehmann,91] has investigated end-system issues in the integration of transport QoS and CPU scheduling. HeiTS also puts considerable emphasis on an optimised buffer pool which minimises 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 role of QoS monitoring, maintenance and commitment are not addressed by either of the above mentioned pieces of work.

The major influence on our work in the end-system area is the split level scheduling scheme described in [Govindan,91]. 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 non real-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 inter-address-space communication known as memory mapped streams (MMS). MMSs are integrated with the scheduling system and work with a range of data transfer implementations such as copying, shared memory or re-mapping. However, the abstraction is only applicable for intra-machine communication. Our QoS controlled connection abstraction performs a similar role but is applicable to remote as well as local communications.

Work on real-time extensions to the Mach micro-kernel, consisting of real-time threads, real-time synchronisation primitives and time driven scheduling, is described in [Tokuda,92]. The scheduling mechanism is derived from the ARTS kernel and permits hard real-time scheduling based on EDF. Again, for end-to-end continuous media support, the main limitation of this work is the lack of API level QoS specification and integration with the communications sub-system. 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 [Tokuda,93].

## 7. Conclusions

We have described the design of a QoS driven communications stack in a micro-kernel 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 minimises kernel level context switches and exploits early demultiplexing so that incoming data, even at the cell level, 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 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 experimental infrastructure consisting of three 486 PC's running Chorus and connected to an Olivetti Research Labs ATM switch via ISA bus ORL ATM interface cards. The PCs also contain VideoLogic audio/ video/ JPEG compression boards as real-time media sources/ sinks. 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.

We would also like to experiment with an ATM interface card with on-board AAL5 support. This would limit the flexibility of our current design and would not allow us to experiment with ATM cell-level scheduling, but we could better evaluate the performance potential of the system if SAR functions did not have to be carried out in software.

Apart from the performance issue, an architectural limitation of our current cell-level card it that it obstructs the ideal strategy of a single, non-multiplexed, per-connection thread operating all the way up/ down the stack. This is because SAR must be carried out asynchronously with higher level protocol processing and thus more than one thread is required. Even worse, we are obliged to use *kernel* threads in the network actor to perform SAR functions and thus incur kernel level context switch overheads. A related drawback is that the receive side AAL5 kernel threads in the network actor are impossible to schedule correctly due to the need to copy cells off the card as soon as possible. With a card featuring on-board AAL5 and DMA for data movement these drawbacks would be eliminated.

There remain a number of important issues which we have yet to tackle. One point that remains to be addressed is the need to synchronise 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 [Hazard,91]. 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 micro-kernel environment are still immature. A final 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.

## Acknowledgement

The research reported in this paper was funded partly by CNET, France Telecom as part of the SUMO project, and partly under UK Science and Educational Research Council grant number GR/J16541. We would also like to thank our colleagues at CNET, particularly Jean-Bernard Stefani, Francois Horn and Laurent Hazard, for their close co-operation in this work.

## References

**[Abrossimov,89]** Abrossimov, V., Rozier M. and Shapiro M., "Generic Virtual Memory Management for Operating System Kernels", SOSP'89, Litchfield Park, Arizona, December 1989.

**[Accetta,86]** Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and M. Young, “Mach: A New Kernel Foundation for UNIX Development”, *Technical Report* Department of Computer Science, Carnegie Mellon University, August 1986.

**[Anderson,91]** Anderson, D.P., Herrtwich, R.G. and C. Schaefer. “SRP: A Resource Reservation Protocol for Guaranteed Performance Communication in the Internet”, *Internal Report*, University of California at Berkeley, 1991.

**[ATM,93]** ATM User Network Interface Specification Version 2.4, August 5th, 1993.

**[Baguette,92]** Baguette, Y., "TPX Specification", OSI 95 Report, ULg-5/R/V1, University of Leige, Belgium, October 92.

**[Bricker,91]** Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and M. Rozier, “Architectural Issues in Microkernel-based Operating Systems: the CHORUS Experience”, *Computer Communications*, Vol 14, No 6, pp 347-357, July 1991.

**[Campbell,92a]** Campbell, A., Coulson G., García F., and D. Hutchison, “A Continuous Media Transport and Orchestration Service”, *Proc. ACM SIGCOMM ‘92*, Baltimore, Maryland, USA, August 1992.

**[Campbell,92b]** Campbell, A., Coulson, G., García, F., Hutchison, D., and H. Leopold, “Integrated Quality of Service for Multimedia Communications”, Proc. IEEE Infocom’93, also available as MPG-92-34, Computing Department, Lancaster University, Lancaster LA1 4YR, UK, August 1992.

**[Campbell,93]** Campbell, A., Coulson, G. and Hutchison, D., "A Multimedia Enhanced Transport Service in a Quality of Service Architure", *Proc. Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster University, Lancaster LA1 4YR, UK, October 93.

**[Campbell,94]** Campbell, A., Coulson, G. and Hutchison, D., “A Quality of Service Architecture”, ACM Computer Communications Review, April 1994.

**[Clark,92]** Clark, D.D., Shenker S., and L. Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism" *Proc. ACM SIGCOMM’92*, pp. 14-26, Baltimore, USA, August, 1992.

**[Coulson,93a]** Coulson, G., and G.S. Blair. "Micro-kernel Support for Continuous Media in Distributed Systems", To appear in Computer Networks and ISDN Systems, Special Issue on Multimedia, 1993; also available as Internal Report MPG-93-04, Computing Department, Lancaster University, Bailrigg, Lancaster, U.K. . February 1993.

**[Coulson,93b]** Coulson, G., Blair, G.S., Robin, P. and Shepherd, D., "Extending the Chorus Micro-kernel to Support Continuous Media Applications", *Proc. Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster University, Lancaster LA1 4YR, UK, October 93.

**[Deering,94]** Deering, S., “Simple Internet Protocol Plus (SIPP) Specification”, Internet Draft, <draft-ietf-sipp-spec-00.txt>, February 1994.

**[Ferrari,90]** Ferrari, D. and D. Verma, “A Scheme for Real-Time Channel Establishment in Wide Area Networks”, *IEEE J. Selected Areas in Comm.*, Vol 8 No 3, April 1990.

**[Govindan,91]** Govindan, R., and D.P. Anderson, “Scheduling and IPC Mechanisms for Continuous Media”, *Thirteenth ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, California, USA, SIGOPS, Vol 25, pp 68-80, 1991.

**[Hayter,91]** Hayter, M and D. McAuley, “The Desk Area Network”, ACM Operating Systems Review, Vol 25, No 4, pp14-21, October 1991.

**[Hazard,91]** Hazard, L., Horn, F., and J.B. Stefani, “Notes on Architectural Support for Distributed Multimedia Applications”, *CNET/RC.W01.LHFH.001*, Centre National d’Etudes des Telecommunications, Paris, France, March 91.

**[Hehmann,91]** Hehmann, D.B., Herrtwich, R.G., Schulz, W., Schuett, T. and R. Steinmetz, “Implementing HeiTS: Architecture and Implementation Strategy of the Heidelberg High Speed Transport System”, *Proc. Second International Workshop on Network and Operating System Support for Digital Audio and Video*, IBM ENC, Heidelberg, Germany, 1991.

**[Jeffay,91]** Jeffay, K., Stone, D. and F. Donelson Smith, “Kernel Support for Live Digital Audio and Video”, *Proc. Second International Workshop on Network and Operating System Support for Digital Audio and Video*, IBM ENC, Heidelberg, Germany, Springer Verlag, 1991.

**[Kurose,93]** Kurose, J.F., “Open Issues and challenges in Providing Quality of Service Guarantees in High-Speed Networks”, *ACM Computer Communications Review*, Vol 23, No 1, pp 6-15, January 1993.

**[Liu,73]** Liu, C.L. and Layland, J.W., “Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment”, Journal of the Association for Computing Machinery, Vol. 20, No. 1, pp 46-61, February 1973.

**[Marsh,91]** Marsh, B.D., Scott, M.L., LeBlanc, T.J. and Markatos, E.P., "First class user-level threads", *Proc. Symposium on Operating Systems Principles (SOSP)*, Asilomar Conference Center, ACM, pp 110-121, October 1991.

**[Scott,92]** Scott, A.C., Shepherd W.D. and A. Lunn, “The LANC - Bringing Local ATM to the Workstation”, *4th IEE Telecommunications Conference*, Manchester, UK, 1993, also available as Internal Report ref. MPG-92-33, Computing Department, Lancaster University, Lancaster LA1 4YR, UK, August 1992.

**[Tanenbaum,88]** Tanenbaum, A.S., van Renesse, R., van Staveren, H. and S.J. Mullender, “A Retrospective and Evaluation of the Amoeba Distributed Operating System”, *Technical Report*, Vrije Universiteit, CWI, Amsterdam, 1988.

**[Tennenhouse,90]** Tennenhouse, D.L., “Layered Multiplexing Considered Harmful”, *Protocols for High-Speed Networks*, Elsevier Science Publishers B.V. (North-Holland), 1990.

**[Tokuda,92]** Tokuda, H., Tobe, Y., Chou, S.T.C. and Moura, J.M.F., "Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network", *ACM Computer Communications Review*, 1992.

**[Tokuda,93]** Tokuda, H., Kitayama, T., “Dynamic QOS Control based on Real-Time Threads”, *Proc. Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, Lancaster University, Lancaster LA1 4YR, UK, October 93.

**[Topolcic,90]** Topolcic, C., “Experimental Internet Stream Protocol, Version 2 (ST-II)”, *Internet Request for Comments No. 1190* RFC-1190, October 1990.

**[Wolfinger,91]** Wolfinger, B., and M. Moran. "A Continuous Media Data Transport Service and Protocol for Real-time Communication in High Speed Networks." *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, IBM ENC, Heidelberg, Germany, 1991.

**[Zhang,93]** Zhang, L., Deering, S., Estrin, D., Shenker, S and D. Zappala, “RSVP: A New Resource ReSerVation Protocol”, *IEEE Network*, September 1993.