3

Implementing a QoS Controlled ATM Based Communications System in Chorus

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

Distributed Multimedia Research Group,
Department of Computing, Lancaster University,
Lancaster LA1 4YR, United Kingdom

phone: +44 (0)524 65201
e-mail: mpg@comp.lancs.ac.uk

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 micro-kernel/ PC environment and supported by an ATM based, QoS driven communications stack. We focus in particular on resource management aspects of the design and deal with CPU scheduling, network resource management and memory management issues. An architecture is presented which 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.

Keyword Codes: C.2.4, C.2.2, H.5.1.
Keywords: Distributed Systems, Network Protocols, Multimedia Information Systems.

1. Introduction

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

- to support real-time and multimedia applications in 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 [1], to underpin both UNIX and real-time applications. Real-time and multimedia applications are supported by the extensions described in this paper. Alongside these, a standard UNIX SVR4 'personality' included with Chorus is used to support UNIX applications.

Our previous work in the field of distributed real-time and multimedia application support has concentrated on API issues [2], CPU scheduling issues [3], transport issues [4] and

G. Neufield et al. (eds.), *Protocols for High Speed Networks IV*

network architecture [5]. 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 inter networking issues are discussed more fully in [5].

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 inter networking,
- a framework for *QoS driven memory management*, and
- a framework for *flow** *management* which integrates the management of resources in both end-systems and the network.

We then discuss the management of CPU, communications and memory resources in this architecture. The various resource management functions are categorised as either *static* or *dynamic* as suggested in [5]. In essence, static QoS management (treated in section 4) 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). Dynamic resource management, which is concerned with run time issues, is not treated in detail in this paper. We offer concluding remarks in section 5.

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 'personalities' (for example a personality is available for UNIX SVR4 as mentioned above). The micro-kernel is implemented using modern techniques such as multi-threaded 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 which run code in the context of an actor. They are scheduled according to either a pre-emptive priority based or round robin time slicing scheme. Ports are message queues used to hold incoming and outgoing messages. The inter-process communication sub-system supports both request/reply and single shot messages.

Chorus has several desirable real-time features and has been fairly widely used for embedded real-time applications. Real-time features include pre-emptive scheduling, page locking, time-outs 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

---

* 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.

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 sub-system, the exclusive use of connectionless datagrams makes it impossible to pre-specify 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 of wiring pages.

Note, however, that such limitations are not unique to Chorus: they are shared by most of the other micro-kernels in current use (e.g. [6], [7]).

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 the following:

- *rtports*: these are extensions of standard Chorus ports and serve as access points for real-time communications. Rtports have an associated QoS which defines timeliness constraints on communication. They also provide direct application access to buffers 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 when data is available/ required. They encourage an event-driven style of programming which is 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*. 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 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 3.3. In the local case, they are implemented in terms of optimised memory mapping mechanisms.

QoS of open connections and for building pipelines of 'software signal processing' modules for local continuous media processing. It also has synchronisation primitives based on eventcounters and sequencers which incorporate the notion of *deadline inheritance* [8] 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 [2] and [8].

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. In this context, we refer to Chorus kernel threads as *virtual processors* (VPs). The scheduling architecture is a *split level* structure [9] consisting of a single kernel scheduler (KLS) to schedule VPs, and per-actor user level schedulers (ULSs) to schedule lightweight threads on those VPs.

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:-

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

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

The KLS/ ULS information exchange is accomplished via a combination of shared KLS/ ULS memory and kernel-to-VP upcalls [9]. The shared memory 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. Upcalls, implemented as *software interrupts*, are used by the KLS to inform VPs of the occurrence of real-time events in a timely fashion. Such events include timer expirations, and data arrivals from local kernel devices or from the network device. Software interrupts are always targeted at VPs but can be initiated either by kernel components (e.g. the KLS) or by library code in application actors.

The design also uses the concept of *non blocking system calls* [10] to ensure that VPs are always available to run light weight threads [8].

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 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 IPC stack. However, we do intend in the future to integrate the functionality of the two stacks in a unified architecture.

3.3.1. Abstract Layering

The communications architecture enforces a strong distinction between communication for signalling purposes (i.e. connection establishment, network resource management and connection tear-down), and user data transfer purposes. The transport, AAL5 and ATM layers are common to both the signalling and the user data stack and are described below. The signalling stack specific layers comprise an upper network sub-layer for resource management

---

* 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.

in IP routers and a lower network sub-layer for resource management in ATM switches. The IP layer is a subset of an existing network resource reservation protocol called RSVP [11] which we encapsulate in the IP Internet Connection Management Protocol (ICMP). The ATM signalling protocol, called ATMSig, is a subset of the ATM Forum's UNI 3.0 [12].

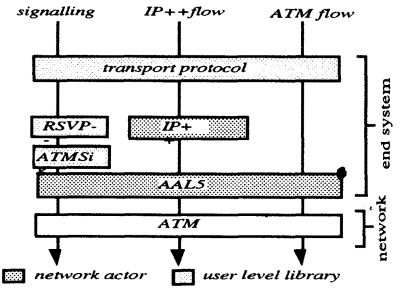

Figure 3: Communications architecture

The user data stack is positioned alongside the signalling stack. The upper architectural layer is a connection oriented transport protocol [13] which provides for QoS specification at connection time (including configurable error control), in service QoS re-negotiation, and end-to-end flow control (via a rate based mechanism). Other traditional 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 the remainder of this section.

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) [14]. 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 4.4.1) to define the QoS associated with each flow. Flow-specs are held by IP++ routers* 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 RSVP signalling 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, PCs, and multimedia devices designed at Lancaster [15]. It also interconnects a number of Ethernets and interfaces to the rest of the UK via an SMDS connection to the SuperJANET 100 Mpbs Joint Academic Network. The PCs which run the system described in this paper are connected to 4x4 ATM switches manufactured by Olivetti Research Limited (ORL) via ISA bus interface cards. The ORL ATM switches are implemented using 'soft' switching and run a small micro-kernel called *ATMos*.

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

* Flow specs are also used by the FMP (see section 3.5) to control resource reservation at the ATM level in ATM switches.

*network actor**. The transport layer of the signalling stack is implemented in the FMP actor described in section 3.5. The transport layer of user applications is implemented in the same user level library† that supports the API abstractions discussed in section 3.1. 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 *asynchronous system calls* [8] 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 two signalling 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 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 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 [16]. 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+. 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' (see section 4.4.3).

3.4. Memory Management Architecture

The purpose of the extended memory management architecture, which is built on top of the existing Chorus abstractions [17], 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† . Our design encapsulates most of the QoS driven memory management functionality inside a system actor 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,
- pre-empting QoS controlled memory from 'low urgency' threads on behalf of 'high urgency' threads when necessary, and,
- efficiently re-mapping QoS controlled memory regions from one actor to another.

---

* Note that this is distinct from the existing Chorus 'network actor' which is called the 'Network Device Manager'.

\+ 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.

† 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.

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 the intra-machine connection case. 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. 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) [5], to realise 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. In end-systems, the FMP co-operates with the CPU and memory management subsystems and in the network it runs on IP++ routers and ATM switches, and coordinates itself by means of the RSVP and ATMsig protocols described in section 3.3. A central role of the FMP is to partition 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 4.1) between the network and the CPU resource managers on each end system.

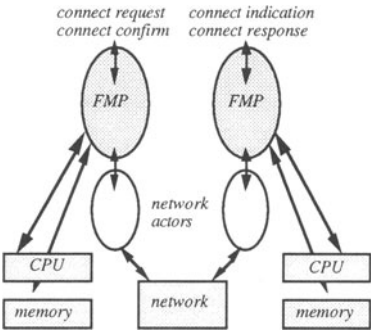

Figure 4: Flow management architecture

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 4) in the user level library determines the resource requirements of the request. The output of the QoS translator is then directed to the FMP which runs in a per-machine FMP actor (see figure 4).

4. Resource Management Strategies

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. This paper concentrates on the reservation of *specific* resources (i.e. CPU, memory and network resources) rather than treating resource resevation 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 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 for fine grained control over 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 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 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 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

---

* 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 4.4.3).

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 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 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, 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 [2].

4.2. 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 G_W are selected on the basis of the *commitment* and *delivery* QoS parameters described in section 4.1:

Best effort { - isochronous (BI) / - workahead (BW) } Guaranteed { - isochronous (GI) / - workahead (GW) }

In addition to the two best effort classes a third best effort class, B_C, is distinguished which refers to non real-time Chorus and UNIX threads out of the scope of the real-time extenstions. Additionally, all three best effort classes are often grouped together and referred to by the shorthand name B. Similarly, the guaranteed classes are collectively referred to as G.

4.3. The CPU Resource

4.3.1. 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 4.1. Figure 6 illustrates the notions of period and quantum together with the related scheduling concepts of *scheduling time*, *deadline* and *jitter*.

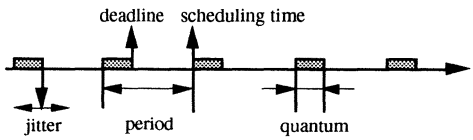

Figure 6: Periodic thread scheduling terminology

For message connections, periodic *sporadic server* threads are used at the receive side. One sporadic server per application actor is provided for each of the two applicable commitment classes (viz. G_W and B; 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:-

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

$Recv\_latency_i$ 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 which requires less resource than a optimal period (i.e. the optimal period, 1 / *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.

4.3.2. Admission Testing

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

- G_I: threads for these connections are scheduled to run such that the completion of a quantum is guaranteed to be completed by the logical arrival time + *j* (where *j* is the jitter QoS parameter and logical arrival time is the start of the requisite period). An extended earliest deadline first [18] (EDF) algorithm and admission test is used to ensure this behaviour.
- G_W: these are scheduled according to the 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 4.4.3).

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 - in particular, guaranteed isochronous threads should only be used for connections which are delivering data to an end device intended for human perception such as a frame buffer or audio chip.

The admission tests for G_I threads are:-

$$\sum_{i=1}^{N_G} \frac{quantum_i}{period_i} \leq R_G, \quad 0 \leq R_G \leq 1 \qquad \text{i)}$$

$$\sum_{i=1}^{N_G} \frac{quantum_i}{jitter_i} \leq 1 \qquad \text{ii)}$$

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 [18] (the latter guarantees that each quantum in the given set of tasks can be completed by the end of its period as long as it is runnable at the

start of its period). The first test ensures that the overall resources used by all G threads are 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.

For G_W threads the admission test is simply:-

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

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.

4.3.3. Dynamic Scheduling Management

At run time, the scheduling scheme uses a combination of *priorities**, *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 highest priority band (only critical Chorus server threads such as the pager daemon). 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 the current quantum). As mentioned above, G_W threads are initially assigned to the lowest priority band but they are 'promoted' to the highest 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.

Non real-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 non time critical daemons, will have accordingly lower priorities.

---

* 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.

4.4. The Network Resource

4.4.1. QoS Translation

The network sub-system 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;
    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 at the IP level. *Mtu_size** 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 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 calculated from the *error* and *error_interval* API level QoS parameters and is equal to 1 - *error / error_interval.*

4.4.2. 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 bandwidth 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 entity 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* any over-allocated resources at intermediate switches [19].

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 4.3.2 the admission control test is based on [18]:-

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

---

* 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.

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 [20]:-

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

Here, d is the delay incurred at the current switch. As before, t_i refers to the service time of flow i in 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 the theorem underlying this formula can be found in [20]

The second phase calculation is:-

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

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 n'th value of d obtained from the first phase calculations.

Buffer Availability Test The amount of per-switch memory allocated to a new flow 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 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.

4.4.3. Cell Scheduling

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

---

* Actually, only *one* thread is required for *all* threads in the G_I class because the G_I admission algorithm has ensured that the quanta of these threads do not overlap and can thus be processed sequentially (see section 4.3.2).

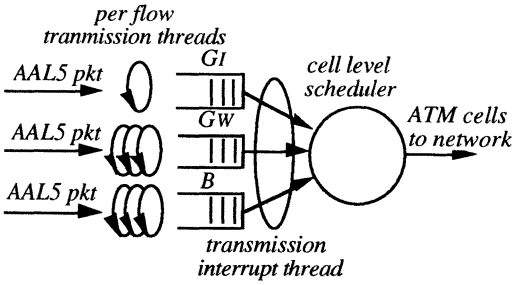

Figure 7: Cell level scheduler

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 4.1). t_{tx} is calculated as *cells* x 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.

4.5. The Memory Resource

4.5.1. 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 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:-

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

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 earliness 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.

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*. 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 non-preemptible.
- 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 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 operations on regions.

4.5.2. 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 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 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.

5. 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 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

---

* 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.

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 severe performance hit, 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. A related drawback is that the receive side AAL5 kernel thread in the network actor is 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 [21]. 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.

The support of the Swiss FNRS for Michael Papathomas through grant no. 8220-037225 is gratefully acknowledged.

References

1. 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.

2. 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, 1994; also available as Internal Report MPG-93-04, Computing Department, Lancaster University, Bailrigg, Lancaster, U.K. . February 1993.

3. 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.

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

5. Campbell, A., Coulson, G. and Hutchison, D., "A Quality of Service Architecture", ACM Computer Communications Review, April 1994.

6. 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.

7. 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.

8. Coulson, G., 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. Editor: Otto Spaniol. Kluwer Academic Publishers, 1994.

9. Govindan, R., and D.P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media", Thirteenth ACM Symposium on Operating Systems Principles, Asilomar Conf. Center, Pacific Grove, California, USA, SIGOPS, Vol 25, pp 68-80, 1991.

10. 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.

11. Zhang, L., Deering, S., Estrin, D., Shenker, S and D. Zappala, "RSVP: A New Resource ReSerVation Protocol", *IEEE Network*, September 1993.

12. ATM User Network Interface Specification Version 2.4, August 5th, 1993.

13. 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.

14. Deering, S., "Simple Internet Protocol Plus (SIPP) Specification", Internet Draft, <draft-ietf-sipp-spec-00.txt>, February 1994.

15. 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.

16. Tennenhouse, D.L., "Layered Multiplexing Considered Harmful", *Protocols for High-Speed Networks*, Elsevier Science Publishers B.V. (North-Holland), 1990.

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

18. 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.

19. 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.

20. 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.

21 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.