SUPPORTING CONTINUOUS MEDIA APPLICATIONS IN A MICRO-KERNEL ENVIRONMENT

Geoff Coulson, Gordon S. Blair, Philippe Robin and Doug Shepherd

Department of Computing,
Lancaster University,
Lancaster LA1 4YR, U.K.
e.mail: mpg@comp.lancs.ac.uk

ABSTRACT

Currently, popular operating systems are unable to support the end-to-end real-time requirements of distributed continuous media. Furthermore, the integration of continuous media communications software into such systems poses significant challenges. This paper describes a design for distributed multimedia support in the Chorus micro-kernel operating system environment which provides the necessary soft real-time support while simultaneously running conventional applications. Our approach is to extend existing Chorus abstractions to include QoS configurability, connection oriented communications and real-time threads. The design uses the following key concepts: the notion of a flow to represent QoS controlled communication between two application threads, a close integration of communications and thread scheduling and the use of a split level scheduling architecture with kernel and user level threads. The paper shows how our design qualitatively improves performance over existing micro-kernel facilities by reducing the number of protection domain crossings and context switches incurred.

1 INTRODUCTION

A considerable amount of research has already been carried out in communications support for continuous media over high speed networks. However, much less work has been done in the area of general purpose operating system support for continuous media. Typically, end-system implementations have either been embedded in non real-time operating systems such as UNIX and suffered from poor performance, or have been implemented in specialised hardware/ software environments unable to support general purpose applications.

The SUMO Project at Lancaster [Coulson,93] is addressing this deficiency in the state of the art by extending a commercial micro-kernel (i.e. Chorus [Hermann,88]) to support continuous media applications alongside standard UNIX applications (Chorus already supports UNIX applications through the provision of a UNIX subsystem or 'personality'). Chorus is a useful starting point for continuous media support as it includes a number of desirable real-time features. However, in common with other micro-kernels, it fails to adequately support continuous media in a number of key areas. First, communication in Chorus is *message based* whereas continuous media requires *stream-oriented* communications. Second, Chorus offers no *quality of service* (QoS) control over communications and only coarse grained relative priority based scheduling to control the QoS of processing activities. Finally, Chorus does not offer *end-to-end* real-time support spanning both the communications and scheduling components.

To overcome these deficiencies we introduce the concept of a 'flow'. A flow characterises the production, transmission and eventual consumption of a single media stream as an integrated activity governed by a single statement of QoS. Realisation of the flow concept demands tight integration between communications, thread scheduling and device management components of the operating system. It also requires careful optimisation of control and data transfer paths within the system.

The rest of this paper is structured as follows. Section 2 provides the background on the Chorus micro-kernel necessary to understand the rest of the paper. Section 3 describes the programming interface to our multimedia facilities and section 4 presents some examples of its use. Following this, section 5 discusses the implementation of the multimedia support, concentrating on communications and scheduling issues. The examples of section 4 are revisited in section 6 to illustrate the qualitative efficiency gains produced by our design over standard micro-kernel facilities. Finally, section 7 discusses related work in the

field and section 8 presents our conclusions and indicates our plans for future work.

2 BACKGROUND ON CHORUS

Chorus, conceived at INRIA, France, is a micro-kernel based operating system which supports the implementation of conventional operating system environments through the provision of 'sub-systems' or 'personalities' (for example a sub-system is available for UNIX SVR4). The micro-kernel is implemented using modern techniques such as multithreaded 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 unique identifiers. Actors are address spaces and containers of resources and may exist in either user or system 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. They can be aggregated into *port groups* to support multicast messaging and may be migrated between actors. Inter-process communication is datagram based and supports both request/reply messages (via the ipcCall() and ipcReply() system calls and one shot messages (via ipcSend() and ipcReceive()).

Chorus has several desirable real-time features and has been widely used for embedded real-time applications. Its real-time features include pre-emptive scheduling, page locking, system call timeouts, and efficient interrupt handling. Chorus also incorporates a framework, called *scheduling classes*, which allows system implementers to add new scheduling policy modules to the system. These modules are upcalled each time a scheduling event occurs. Modules impose their scheduling decisions by manipulating a global table of thread priorities.

Unfortunately, Chorus' real-time support is not sufficient for the requirements of distributed multimedia applications, principally because there is no support for QoS control 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.

In addition, Chorus suffers from a lack of communications/ scheduling integration. This means that there is no way to provide timely scheduling in concert with communications events as required by end-to-end continuous media communications. 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 PROGRAMMING INTERFACE AND ABSTRACTIONS

To remedy its current deficiencies for real-time continuous media support and real-time control, we have extended the Chorus API with new low level calls and abstractions (provided in a user level library called *libflow*).

3.1 Primitive Abstractions

The primitive abstractions are as follows:-

- *rtports* - communication end points for real-time communications,
- *devices* - hardware and/or software producers, consumers and filters of real-time data,
- *rthandlers* - user defined procedures which manipulate real-time data coming from or going to an rtport,
- *QoS controlled connections* - communication channels with a specific QoS, and
- *QoS handlers* - user defined procedures which are upcalled when QoS commitments degrade.
- *threads* - a set of real-time lightweight thread management and synchronisation primitives.

These abstractions are described fully in the following sub-sections.

Rtports and Devices

Rtports are an extension of standard Chorus ports and serve as end-points for both continuous media communications and real-time

messages with bounded latency. As such, they participate in the implementation of end-to-end flows (in conjunction with rthandlers and QoS controlled connections). Like Chorus ports, rtports are named globally and can be accessed in a location independent fashion from anywhere in the distributed system.

There are, however, the following differences between Chorus ports and rtports:-

- rtports have an associated QoS,
- the internal buffers of rtports can be directly accessed by the application programmer,
- rtports may not migrate because the QoS commitments offered by the rtport assume a fixed association between the underlying device and the actor performing the create operation.

A *device* is a producer, consumer or filter of real-time data which supports the creation of rtports. Devices may be either software drivers for physical devices or independent software objects that generate, process or consume continuous media data. One important type of device is the *null* device. This 'device' enables pieces of user code to act as data sources and sinks. An rtport associated with such a device is similar to a conventional Chorus port except that ipcSend() and ipcReceive() calls made on such a port may be latency bounded if the rtport is the end-point of a QoS controlled connection.

Rtports are created with the following call:-

```
typedef enum {
    s_message, s_stream
} flow_type;
status rtportCreate(DEV *d; flow_type service; rtport *p);
```

The user specifies to the system a device on the local site with which the new rtport should be associated, the required QoS of the rtport and the type of service required: either a QoS controlled message service or a stream service. The device argument is a standard Chorus unique identifier which refers to a hardware or software device. Note that rtportCreate() will only succeed when the referenced device resides on the local site.

The QoS specification is used to denote the potential level of service of the rtport and to deduce future resource allocation needs. However, resources are not actually committed until the rtport is involved in a connection (see later sub-section). QoS parameters are specified to the system by means of a data structure called a QoSVector. Different definitions of this data structure are used depending on the service type required. QoSVector definitions for the QoS controlled message and the stream service are given below :-

```
typedef enum {best_effort, guaranteed} com;
typedef enum {isochronous,workahead} del;

typedef struct {
    com commitment;
    int buffsize;
    int priority;
    int latency;
    int error;
    int buffrate;
    int jitter;
    del delivery;
    int error_interval;
    } StreamQoS;
```

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

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

The first four parameters are common to both service types. The *commitment* parameter allows the programmer to express a required degree of certainty that the QoS levels requested will actually be honoured. If the commitment is *guaranteed*, resources are permanently dedicated to support the requested QoS levels. Otherwise, if the commitment is *best effort*, resources are not permanently dedicated and may be preempted for use by other activities. A later sub-section describes special reporting facilities provided in the case that requested QoS levels are violated. *Buffsize* specifies the required size of an internal buffer to be associated with the rtport. *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 (see later in this section).

The *error* parameter has a different interpretation depending on the type of service requested. For stream connections, error, which is used in conjunction with *error_interval*, refers to the maximum permissible number of buffer losses and corruptions over the given interval. In the case of message connections, error 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 conections. Buffrate refers to the required delivery rate fo buffers at the sink end of the connection. Jitter, measured in milliseconds, refers to the permissible tolerance in buffer delivery time from the periodic delivery time implied by buffrate. The delivery parameter also refines the meaning of buffrate. If delivery is isochronous, the stream service delivers precisely at the rate specified by buffrate; otherwise, it attempts to ‘work ahead’ (ignoring the jitter parameter) at rates temporarily faster

than buffrate. One use of the workahead delivery mode is to support applications such as real-time file transfer. Its primary use, however, is for pipelines of processing stages.

Rthandlers

Rthandlers are user supplied C functions which may (optionally) be attached to rtports. They may be attached to both sending and receiving rtports. Rthandlers are upcalled from their associated rtport whenever data is required at a source rtport or has been delivered, by a connection, to a sink rtport. The thread which upcalls the rthandler runs in user mode and thus allows the user the freedom to provide arbitrary code for rthandlers. The upcalling of an rthandler performs two logically distinct functions:-

i) *event notification* - it is indicated that data is required or has been delivered, and
ii) *data transfer* - it is made possible for the rthandler to access the rtport's buffer to insert or extract data as appropriate.

Applications can use rthandlers either for the notification of events alone, or for both event notification and data transfer. We feel that this separation of notification and delivery is important for continuous media applications. It permits applications to choose whether they want to actively process continuous media data in user space, or merely to track the passage of continuous media generated and consumed in supervisor space. This latter case arises when the device under consideration is, for example, a kernel managed video device with associated frame buffer which is receiving data directly from the network card. Here, efficiency can be maximised as continuous media data need not cross protection domains.

The call to attach an rthandler to a rtport is as follows:-

```
typedef int(Rthandler)(Buffer **b; int *size; int *event;
                       time_t *send_timestamp, *recv_timestamp;
                       bool admission);
status rtportAttachRthandler(rtport *p; Rthandler f,
                             int eventmask; short priority);
```

The first two arguments to rthandler functions inform the application's code of the size and address of the rtport's internal buffer. In cases where the buffer is mapped into user space, this permits user code to directly supply/extract data from the buffer while rthandlers are

executing[1]. Rthandlers can assume that they have exclusive access to the buffer for as long as their call is extant. In the case of kernel supported devices, direct user access to buffers may or may not be possible depending on the protection attributes imposed by the device and/or the kernel. If access to buffers is disallowed (as indicated by a NULL value for the b parameter), the rthandler performs an event notification but not a data transfer role.

The third parameter, *event*, allows application code to associate multiple logical message types with a single rthandler. Source rthandlers provide an integer event identifier and this is passed on, unchanged and uninterpreted, with the corresponding buffer, to the destination rthandler. The fourth and fifth arguments supply timestamps for the benefit of the receiver. These relate to the times at which the buffer was obtained at the source and delivered at the sink. Finally, the *admission* parameter is used to allow the infrastructure to perform a scheduling admission test by determining the execution time of the rthandler(s). When it sees a *true* admission value, application code in rthandlers is expected to direct the calling thread on a dummy run through a 'typical' path so that the resource manager can derive an estimate of the execution time of the (set of) rthandlers in normal circumstances. This execution time is added to the a priori known time for protocol processing to help derive the deadline of the rtport's associated thread[2]. Note that admission is only given as true under these circumstances; at all other times it is given as false.

The first two parameters to the rtportAttachRthandler() call itself specify the rtport to which the rthandler is to be attached and the rthandler function. The third and fourth parameters are used to control the behaviour of rthandlers when multiple rthandlers have been attached to the same rtport. *Eventmask* is a bitmap used to determine for which set of events (see above) this particular rthandler will be called, and *priority* is used to determine the order in which multiple rthandlers are called when more than one share a common event[3]. Finally, the rtportAttachRthandler() call will fail if the given rtport was not created

---

1 The indirection of the buffer and size parameters allows application code to change to a new buffer before returning control to the infrastructure. Note also that rthandlers with overlapping events and the same priority are executed in non-deterministic order.

2 The execution time of rthandlers may involve waiting for the completion of other threads which run as a result of actions taken by rthandler code (see the section on threads below). Note also that the execution time of the rthandler is monitored by the system and dynamically refined at run-time over multiple calls. Details of our scheduling scheme are given in section 5.1.

3 The only factor used to determine the ordering of rthandler calls at *source* rtports is the priority parameter because, of course, event identifiers are not known until the rthandler has returned.

in the current address space. This is because the virtual address of the rthandler must have an interpretation in the rtport's supporting address space.

Rthandlers are central to our design for two major reasons:-

- *programmability*
  Because rthandlers are executed under QoS constraints imposed by their associated rtport, they provide a way to execute application code without compromising the system provided QoS support required for the realisation of flows. We also contend that structuring the API with rthandlers is a natural and effective model for real-time programming. Real-time programming is considerably simplified when programmers can structure applications to react to events and delegate to the system the responsibility for initiating events. Of course the programmer is still ultimately in control of event initiation but this control is expressed declaratively through the provision of a QoSVector parameter and need not be explicitly programmed in a procedural style.
- *efficiency*
  An efficiency gain results from the use of a single thread (originating in the communication system) for both protocol and application processing. In conventional systems, applications interface with communications by performing system calls which block and reschedule if the protocol is not ready to send or if data has not yet arrived. With the rthandler implementation, on the other hand, no context switch is incurred and it is not necessary for the application and protocol to wait for each other as the protocol always initiates the exchange and the application code should always be ready to run.

Figure 1 illustrates devices, rtports and rthandlers. It also shows incoming QoS controlled connections (shown as heavy black lines), one for each of the two rtports, and rthandlers attached to the rtports and upcalling into user space. One of the rtports is associated with a *null* device (where the rtport is implemented in the libflow user level library), and the other with a physical device (implying an rtport implementation in supervisor address space[4]).

In the case of the actor rtport, the rthandler is performing the role of both event notification and data transfer. In the case of the kernel device rtport, the rthandler plays a similar event notification role (i.e. data is

---

4 As rtports can be implemented in either supervisor space or user libraries, the stubs of the Chorus ipcSend() and ipcReceive() are modified to distinguish between the two different implementations and perform either a trap or a library procedure call as appropriate.

being sent/ received), but application code does *not* directly participate in the data transfer. Instead, data is directly obtained from/ delivered to the device by the connection associated with the device, and need not cross into user space.

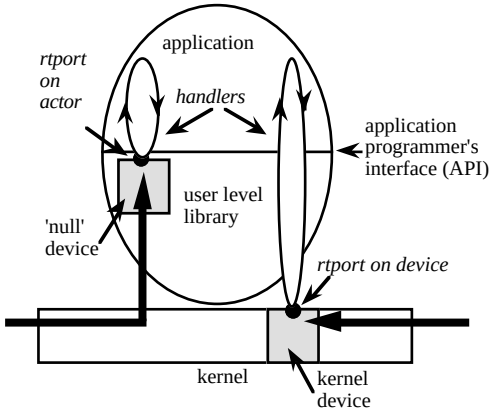

Figure 1 Devices, Rtports and Rthandlers

QoS Controlled Connections

All communication in standard Chorus is connectionless and datagram based. However, as noted in section 2, flow services require resource reservation commitments both in the end-system and in the network. Because of this, we have added simplex connection oriented communications called QoS controlled connections to abstract over the necessary resource allocation. The call to set up such a connection is as follows:-

```
status rtportConnect(rtport *source, *sink; QoSVector *qos;
rtport *ctl);
```

The rtportConnect() call takes two rtports, a source and a sink, as its primary parameters. Notice that because of the location independent nature of rtports, it is possible to call rtportConnect() from a site entirely separate from the sites on which the source and sink rtports reside. This is a convenient facility for distributed multimedia applications which are often structured as a centralised master process supervising and controlling a number of physically distributed sources and sinks [Anderson,91a]. The internal protocol for this facility, which we call remote connection, is fully described in [Campbell,92a].

The QoS parameters are the same as those used at rtport creation time except that the values specified at connect time are *actual* rather

than *potential*. The values specified must be less than or equal to those specified in the QoS of both rtports involved, or rtportConnect() will fail. The two rtports must also have been created with the same flow service type: either message or stream types. The *latency* QoS parameter subsumes rthandler execution time if rthandlers are attached and thus specifies the full latency of an application-to-application flow. If rthandlers are not attached, latency is interpreted as rtport to rtport. In the latter case, latency is measured from the time a thread calls ipcSend() to the time the buffer is received (but not necessarily delivered to the application) at the sink rtport.

There are two categories of connection corresponding to the two types of flow supported by rtports:-

- *message connections*
  Message connections wait passively until activated by ipcSend() calls in the conventional manner. It is possible to attach rthandlers at the sink end of message connections, but source rthandlers are inapplicable because there is no active entity in the connection to call them.
- *stream connections*
  Each end of a stream connection tries to *actively* obtain/ deliver data at the rate determined by buffrate. If rthandlers are attached, this results in the calling of the rthandlers, otherwise the connection blocks until the application calls ipcSend() or ipcReceive().

The final argument to rtportConnect() is a result parameter which returns a new rtport used to dynamically control the behaviour of connections. The ‘operations’ available on the control rtport are the following:-

- *renegotiate* - this allows the user to dynamically change the QoS of the connection by supplying a new QoSVector argument,
- *disconnect* - to destroy a connection,
- *start, stop* - respectively activate and de-activate the connection, and
- *prime* - ready the end-to-end connection by filling the receive buffers so that a subsequent start will take immediate effect.

The last three operations are only applicable to stream connections. Note that start, stop and prime can also be used for cross-stream synchronisation purposes; their use in this context is described in [Campbell,92a].

In implementation, connection establishment is provided by a per-site connection manager which is realised as a user level actor. The connection manager maintains a list of actors and their associated rtports for that site. The manager accepts incoming connection requests

and dispatches them (via standard Chorus IPC) to a listening lightweight thread in the appropriate rtport implementation.

QoS Handlers

QoS handlers are upcalled by the system in a similar way to the rthandlers described above. However, whereas the above handlers notify communication events and allow access to communication buffers, QoS handlers are used to notify the application layer when the QoS commitments provided by connections have been violated. The intention is that QoS handlers will usually be attached and supported by application level QoS manager objects. QoS managers embody a particular policy for coping with QoS degradations. For example, they may attempt to request renegotiation of QoS or choose a connection to close down on the basis of application defined criteria. The call to attach a QoS handler is:-

```
typedef int (QOSHandler)(QoSVector *current, *new);
status rtportAttachQOSHandler(rtport *p; QOSHandler f);
```

Threads

Although lightweight threads (see section 5.1) are implicitly created on behalf of applications when they establish QoS controlled connections, we also allow applications to *explicitly* create lightweight threads, both real-time and non real-time. Our thread types and thread creation primitives are similar to those described in [Tokuda,90] and hence will not be described here. However, our thread synchronisation primitives are novel in that they incorporate the concept of *deadline inheritance*[5].

The most important of our thread synchronisation calls, which are based on *eventcounters* and *sequencers* [Reed,79], are as follows (note that the boolean result of await() is used to distinguish whether the call returned due to timeout expiry or the evencounter target being reached):-

```
void advance(eventcounter *e, bool inherit);
```

---

5 Note that deadline inheritance solves a slightly different problem to the various solutions to the well known *priority inversion* problem (e.g. see [Tokuda,90]). The latter problem occurs when a thread with a late deadline holds a multual exclusion lock for which a thread with an early deadline is waiting. Deadline inheritance as described here is related to *condition synchronisation* rather than mutual exclusion.

```
bool await(eventcounter *e; u_long target; time_t timeout);
```

The semantics of deadline inheritance are as follows:-

- Each thread maintains a list of deadlines sorted earliest deadline first. The entry at the front of the list is the *effective deadline* used by the scheduler. When a thread starts to run it has only one entry in its list.
- Whenever a thread T executes ecs_advance(e), any threads that were blocked on an earlier call of ecs_await(e), and freed as a result of the ecs_advance(e) call, *inherit* the effective deadline of T *iff* T has an earlier deadline than their current effective deadline. The mechanism of inheritance is to place the inherited deadline at the front of the deadline list of the inheriting thread. When multiple threads share the same deadline value, the one with the shortest deadline list takes preference.
- Whenever a thread which has passed on a deadline to other threads decides to terminate, all inherited entries associated with the terminating thread are removed from the deadline lists of all the inheriting threads. This may cause a change in the effective deadline of one or more of the inheriting threads.
- The programmer can explicitly enable or disable inheritance by appropriately setting the second parameter of ecs_advance().

Deadline inheritance releases application programmers from the need to structure their event handling code (expressed in rthandlers) in a single sequential thread. With deadline inheritance, code can be structured in terms of an arbitrarily complex system of concurrent worker threads without compromising the predictable performance of rthandlers. To do this, an rthandler thread would typically call ecs_advance() to release one or more blocked worker threads (which themselves could, of course, unblock further threads). Each time a worker thread was released from a blocked ecs_await() call it would inherit the deadline of the original rthandler thread and, in this way, the original deadline would propagate through the system. Finally, before returning, the original rthandler thread would perform an ecs_await() call to wait for the system of worker threads to complete their task.

3.2 Compound Abstractions

The compound abstractions, described in the following sub-sections, are as follows:-

- *invocation* - a request/ reply service composed of multiple message flows,

- *pipeline* - concatenations of QoS controlled stream flows containing intermediary processing stages.

Invocation

The invocation service is a compound service realised in terms of two message connections arranged in a back to back, request/ reply, configuration.

To provide a convenient interface to the programmer, the libflow library exports abstractions which relieve the programmer of the necessity of explicitly manipulating the two message connections. These abstractions comprise an operation to create an invocation service instance (invocationConnect()) and operations to send and receive data on an invocation service instance (ipcCall() and ipcReply()). The operation to establish an invocation service instance is as follows:-

```
status
invocationConnect(rtport *clientPrt, *serverPrt;
                        InvocationQoS *qos; rtport *ctl);
```

InvocationConnect() uses the supplied rtports (clientPrt and serverPrt) as the endpoints of the request connection. The underlying implementation transparently allocates the two rtports required for the reply connection. These two rtports do not need to be visible to the user because of the semantics of the ipcCall() and ipcReply() calls (see below). Note, however, that it is the programmer's responsibility to attach an rthandler to serverPrt before using the service. The InvocationQoS parameter denotes the required end-to-end QoS of the invocation and includes fields such as round-trip latency and an option field which specifies at-most-once or at-least-once semantics. This information is used to determine the QoSVector parameters in the two underlying connections.

In use, a client thread at the initiating end calls ipcCall(). Then, at the server end, the incoming call is executed on the sink rtport's rthandler. When the server wishes to reply, it calls ipcReply() without explicitly specifying an rtport.

Pipelines

Our design supports the requirement for pipelines through the concatenation of stream connections. Processing stages in pipelines are realised as rthandlers attached to intermediate *filter* devices. Some filter

devices, e.g. compression functions, are implemented in hardware and managed by the kernel. Other filtering requirements, however, can be met by application level software processing. To permit application writers to implement such processing, we provide a generic software filter device known as a *connector*. Connectors are implemented in the libflow user level library in a similar way to the invocation abstractions described above. Figure 2 illustrates a three stage pipeline with two intermediate connector devices.

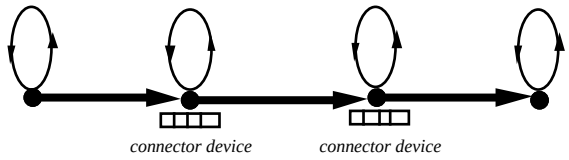

Figure 2 Pipeline

The following call is used to create a connector:-

```
status rtconnectorCreate(DEV *d);
```

A connector device is an encapsulated bounded buffer on to which programmers can create rtports. Programmers create a single rtport on the connector to serve as both the sink of the upstream connection and the source of the downstream connection. An rthandler is then attached to the rtport to encapsulate the required application processing. The rthandler is invoked from below when a buffer is available on the upstream connection and the downstream connection waits until the rthandler returns. As the connector's buffer is shared between the two connections, no buffer copy overhead is incurred between pipeline stages.

To enable pipelines to be created with a single statement of end-to-end QoS (and thus fall within the definition of flow), the following call is used:-

```
status
pipelineConnect(rtportList rtports;
                        StreamQoS *qos;
                        rtport *ctl);
```

PipelineConnect() takes a list of rtports, all of which are assumed to have an rthandler already attached. An ordinary StreamQoS structure specifies the end-to-end QoS. If the *delivery* flag in the StreeamQoS structure is set to isochronous, only the last connection in the pipeline is actually set up as an isochronous binding. This is because it is desirable to permit as much asynchronicity as possible so that the pipeline can run with minimal constraints and maximal elasticity. As

long as the last pipeline stage runs with the required isochronicity the users end-to-end QoS specification is sufficiently upheld.

In implementation, the end-to-end QoS specification supplied to pipelineConnect() is partitioned across the various component connections using a resource reservation protocol described in [Campbell,93]. The result parameter *ctl* is exported by libflow as a convenient handle to control the end-to-end pipeline flow. When programmers invoke control operations on this rtport, libflow must perform a (non trivial) mapping to the control rtports of the full set of constituent connections.

4 EXAMPLES OF USE

Consider the two applications illustrated in figure 3. The first application (left) is transferring real-time audio from a kernel managed audio device on one machine to a kernel managed speaker device on another machine. The second application is using the same source and sink, but also pipes the data through a real-time software device implemented in user space.

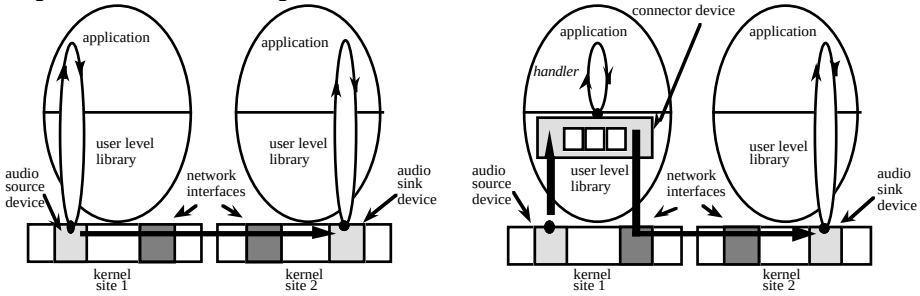

Figure 3 Two Application Scenarios

The first application is implemented by creating rtports on the source and sink devices and connecting them with a QoS controlled connection. As both devices are in kernel space and no application specific processing is required, data never crosses the kernel/ user boundary at either the source or sink machines. Instead, all processing and copying is performed in kernel space. Nevertheless, the programmer can synchronise with the flow of data by attaching rthandlers which are executed on a lightweight thread each time a buffer is sent or received. The user cannot gain access to the rtport's buffer because of kernel protection constraints, but he/she knows the precise instant at which data is sent or delivered.

The second application uses a two stage pipeline with the data piped through user space via a connector device. The first pipeline stage connects an rtport on the source device and an rtport on the connector. The second stage connects a second rtport on the connector and an rtport on the remote sink device. The user's application specific code is written into rthandler functions attached to the connector's rtport(s). This arrangement is simple and intuitive for the programmer and yet is still susceptible to a highly efficient implementation as will be explained in the following section.

5 IMPLEMENTATION

In this section, we discuss scheduling and communication implementation issues arising from the abstractions already described.

5.1 Scheduling

The scheduling implementation exploits the concept of *lightweight threads* to minimise the overhead due to context switches. However, the design also uses kernel threads running in both user and supervisor modes. To qualify the use of the term 'thread' in the following sub-sections, we introduce the following definitions:-

- *kernel threads* - kernel supported threads which run in either user or supervisor mode,
- *lightweight threads* - implemented in the libflow library and multiplexed on top of kernel threads.

For real-time operation, both classes of thread are non time-sliced. When used to implement stream connections, threads run *periodically* at a rate determined by the QoS of the connection. In our current scheme, threads belonging to connections with guaranteed commitment and isochronous delivery are scheduled to run non-preemptively for each period and have their execution periods pre-allocated along the future time line [Robin,94] (an admission test is used at connect time to ensure that sufficient resources are available). Threads with other combinations of commitment and delivery use preemptible earliest deadline first (EDF) scheduling [Liu,73] (with an appropriate admission test).

Non real-time threads are scheduled according to standard Chorus policies (e.g. round robin timesliced) and share whatever processor time is left after real-time threads have taken their requirements. The

real-time thread extensions co-exist with the existing Chorus facilities offered by the scheduling classes mechanism already described in section 2. More details of resource allocation issues in scheduling are given in [Robin,94].

The implementation architecture for real-time thread scheduling is a split level scheme [Govindan,91] consisting of a single kernel scheduler (KLS) and multiple co-operating user level thread schedulers (ULS), one in each actor. Each actor multiplexes lightweight threads on a small number of kernel threads dedicated to the actor (ideally only one kernel thread for uni-processors); this is depicted in figure 4.

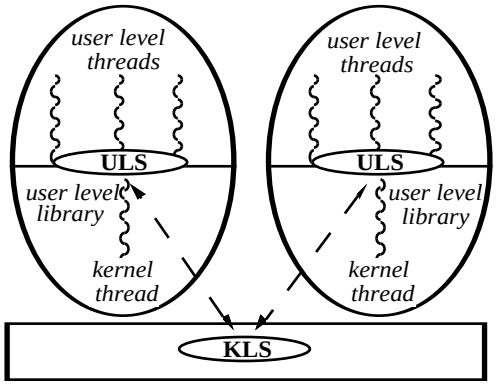

Figure 4 Split Level Scheduling Architecture

For EDF threads, the scheme maintains the following invariants with respect to the two types of scheduler:-

i) each ULS runs the lightweight thread in its actor with the earliest deadline,
ii) the KLS runs the virtual processor of the actor with the globally earliest lightweight thread deadline.

The necessary information exchange between the kernel scheduler and the user level schedulers is accomplished via a combination of shared memory and upcalls from the kernel [Govindan,91]. In the implementation of the shared memory area, each kernel thread (referred to as a *virtual processor)* has an associated context structure [Marsh,91] which contains information such as current lightweight thread context, next runnable thread and earliest deadline of a lightweight thread supported by this virtual processor. Each context is mapped into kernel space and used to exchange/share information with the kernel, thus avoiding unnecessary system calls. For example, the system clock is mapped to each virtual processor (read only) avoiding the cost of a system call to get the time value. The globally earliest deadline of each virtual processor is read on each kernel level rescheduling operation by the KLS to compute the next virtual processor to schedule.

The attraction of the split level scheme is that many context switches at the user level can take place without the need for expensive kernel level context switches. For example, in an intra-actor pipeline, if a number of the rthandler threads have deadlines earlier than any lightweight thread in any other actor, then these threads can be switched freely at the user level while their supporting virtual processor runs uninterrupted. This can result in considerable time savings as context switches at the user level are an order of magnitude more efficient than kernel context switches.

Each virtual processor must ensure that it responds in a timely fashion, not only to the deadlines of its own lightweight threads, but also to externally generated events which demand service from new threads. The most important such events are:-

i) timer events used to implement pre-emption in user level scheduling,
ii) buffer arrivals from local kernel devices,
iii) network packet arrivals from the network device, and
iv) indications that a buffer is being delivered to/ sent from a kernel device where the connection is such that the buffer need not pass into user space (see section 4).

It is essential that such events are notified to virtual processors in as efficient a manner as possible. We have already rejected the standard Chorus strategy of having a thread waiting on a port because of the associated overhead of a synchronisation and context switch. Our favoured solution is to employ *software interrupts* whereby the occurrence of an event causes the virtual processor to jump to an entry point in its user level scheduler. When this happens, the user level scheduler saves the current lightweight thread context and schedules the appropriate lightweight thread to perform the required action.

Other researchers have proposed the use of ‘scheduler activations’ [Anderson,91b] for event notification purposes; these are effectively kernel threads which upcall into the scheduler when scheduling events occur. The difference between scheduler activations and software interrupts is that scheduler activations provide a new kernel supported virtual processor in addition to an event notification. In contrast, software interrupts handle the event on the stack of the actor’s single virtual processor. In our environment, however, scheduler activations suffer from the problem of increased kernel level concurrency (with the associated overhead of kernel managed context switches). While this increase in concurrency can be beneficial in a multiprocessor architecture, its appeal in a uniprocessor design is less clear. The advantage of the software interrupt mechanism is that it allows an invariant to be maintained of only one virtual processor per actor. As previously explained, it is optimal for the split level scheduling scheme

to have the same number of virtual processors per actor as there are real CPUs. If the number of virtual processors exceed this number, the split level scheduling scheme is compromised as it is not clear on what basis to schedule multiple virtual processors per actor which are running on a single CPU.

A further issue is the problem of 'priority inversions' where a thread with a later deadline executes at the expense of a thread with an earlier deadline. This situation can arise in our design when a lightweight thread performs a blocking system call and thus blocks its underlying virtual processor. As we prefer only one virtual processor per actor, other lightweight threads in the same actor will be unable to execute while the blocking call is extant - even if they have the globally earliest deadline. Scheduler activations would solve this problem by injecting a new execution context whenever the actor's kernel thread blocks (but at the expense of an undesirable increase in kernel level concurrency as discussed above). Our favoured solution is to employ *non blocking system calls* [Marsh,91]. These calls return immediately and thus allow the calling kernel thread to resume acting as a virtual processor for lightweight threads (the result of the call will eventually be notified by a software interrupt as discussed above). This strategy enables the scheduling invariants to be maintained whilst avoiding extra kernel level threads per actor.

5.2 Communications

General Case

Connections between devices on different machines are implemented via a connection oriented transport protocol specifically designed to support QoS controlled communications. The protocol was designed and implemented as part of an earlier project at Lancaster [Shepherd,91]. It supports QoS parameterisation at connection set-up time and also monitors QoS and reports on degradations. It is possible to dynamically renegotiate QoS levels on the basis of these reports. The protocol uses a rate based scheme [Clark,87] for flow control whereby sources and sinks negotiate a mutually acceptable transfer rate at connection set-up time. This allows data to flow at a smooth rate, which is important for continuous media, and also permits responsive back pressure to be applied when the sink runs out of buffers. Rate based flow control has further advantages. First, is lightweight and permits higher throughput than schemes based on windowing. Second, it decouples flow control from error control so that connections which do not require error control do not need to pay for it. Third, it fits in

nicely with the notion of periodically schedulable threads. A final point is that, as recommended by researchers in the area (e.g. [Tennenhouse,90]), our communications design uses no multiplexing in the protocol stack above the link layer.

The transport protocol can run in either supervisor mode or user mode. Supervisor mode is appropriate for connections involving kernel supported physical devices as both data transfers and execution for these connections are confined to supervisor memory space. User mode is appropriate for connections terminated by user supplied rthandlers attached to null devices. In the user mode implementation, the transport protocol runs in the context of lightweight threads in the address space of a user virtual processor, the network card is accessed through kernel calls to a device driver which provides an interface at the link-layer, and the protocol itself is implemented in the libflow user level library. It is essential for efficiency that the user mode protocol implementation minimises the number of kernel calls per user level buffer. To achieve this, the implementation batches data to reduce the number of transfers to the network card and also minimises memory allocation calls by maintaining its own buffer cache. Experience will tell if these techniques are sufficient but results reported in [Forin,90] and [Thekkath,93] are encouraging.

Note that the user library transport implementation and rthandler mechanism simplify the task of scheduling in two respects. Firstly, the deadline and required CPU time for the processing of each continuous media buffer is known in advance. The former is obtained implicitly from the QoS specification of the connection; the latter is deduced by adding the transport and rthandler execution times. Secondly, the rthandler scheme eliminates any need for synchronisation between the application and a distinct transport entity: a single seamless thread of execution subsumes both these activities.

To realise the active semantics of stream connections, connections have dedicated lightweight threads at each end (in the case of connections whose end rtports are kernel managed, these lightweight threads are supported by virtual processors running in supervisor mode in supervisor actors). The source thread is responsible for continually executing the user's rthandler(s), obtaining data (either from the rthandlers or directly from a physical device as appropriate), and executing the transport protocol. The sink thread operates analogously; it is awakened when the full set of link-layer packets making up a user level buffer have been received, and then executes the transport

protocol and delivers the data either by calling the user's rthandler(s) or by delivering data directly to a device[6].

Optimisations

A number of important optimisations are possible if communication is between devices in the same address space. In this case, connections between rtports in the same actor are simply implemented as a single lightweight thread which repeatedly calls the source rthandler(s) with a particular buffer address and then calls the sink rthandler(s) with the same address. This single mechanism serves to implement both the data transfer and the delivery notification aspects of the communication. It also implicitly ensures mutual exclusion to the buffer by the source and sink rthandlers. Connections between rtports in the same actor are often used in the context of intra-actor pipelines. To minimise data copying in such pipelines, the address of a single buffer is passed from stage to stage as the various stages of the pipeline are executed. Finally, when the last pipeline stage in the actor has disposed of the data, the buffer is released. If new data arrives at the actor while the previous data is being passed along the pipeline, processing of this data proceeds concurrently using a separate buffer. In this way, it is possible to efficiently implement arbitrarily long intra-actor pipelines without incurring data copying overheads.

Further optimisations are possible in communication between kernel and user space. Chorus currently incurs (at best) one copy and one virtual memory remap for a data transfer from the network driver to user space. We can reduce this to zero overhead per transfer and one single virtual memory remap incurred at connection establishment time. Our strategy is to dedicate shared, per connection, physical memory

---

6 In our current, ATM based, implementation we run AAL5 in software in the ATM card's device driver. On the send side, a per-connection thread in the driver is responsible for fragmenting AAL5 packets into ATM cells and feeding them to the network. On the receive side, another per-connection thread in the driver builds up AAL5 packets from incoming cells. The destination virtual processor is informed of the completion of each packet by a software interrupt. It then schedules the connection's lightweight thread to perform transport processing (e.g. transport packet re-assembly and checksumming). The lightweight thread blocks after it has processed each AAL5 packet and waits for the next. Then, having completed processing of the last packet, it upcalls the user's rthandler as described above. An implementation that used on-board AAL processing would clearly be more efficient than our current solution but the software interrupt, scheduling and transport protocol processing design would be unaffected.

buffers between the network driver and user space which are mapped in the address spaces of both. Note that, by a simple extension, this scheme can also be applied to the case of actor to actor communications on the same machine. The strategy here is for each actor to unmap a buffer from its own address space, map it into the address space of the next actor in the pipeline, and then pass a software interrupt to the next actor to implement the event notification aspect of the connection. We are currently looking into the possibility of combining these three operations into a single optimised system call. This topic, together with other issues concerning the allocation and preemption of physical buffers to/ from user programs are discussed in [Robin,94].

6 EXAMPLE REVISITED

To illustrate how the communications and scheduling subsystems work together, we now return to the second example of section 4. This example involved a two stage pipeline with its intermediate device in user space and its source and sink devices in the kernels of separate machines.

On the source machine, the data and control flow pattern is as follows. First, data is copied (or DMA transferred) by the first connection from the audio device to the connector device's user level buffer. This connection is purely local and is implemented as a lightweight thread running in supervisor mode in the supervisor actor encapsulating the audio device driver. Having performed the copy (or supervised the DMA), this thread delivers a software interrupt to the sink actor's user level scheduler. The ULS, on receiving the interrupt, schedules a lightweight thread to execute the connector device's rthandler. When it runs (as determined by its deadline) this lightweight thread executes the code in the user's rthandler. When the rthandler returns, a context switch takes place between the current thread and the lightweight thread of the downstream connection (this thread will have been previously blocked waiting for data on a user level synchronisation primitive). Note that this context switch involves no kernel overhead whatsoever (assuming that no other actor has a globally earlier deadline); the original kernel thread simply changes from executing the first lightweight thread to the second. Having become unblocked, the new lightweight thread starts to run the send side transport protocol for the second connection and eventually issues a system call to the network device driver to ask it to copy data from the connector's buffer to the network card.

In total, the above processing on the source machine has cost two domain crossings (for the system call), two kernel context switches and two copies (or one copy and a DMA transfer). Note also that, if a longer pipeline had been involved, with multiple connectors in the same actor, these costs would have remained identical. If, however, the equivalent processing had been carried out using conventional Chorus mechanisms, the expense would have been four domain crossings (two for a 'read' operation and two for a 'write'), a context switch and four copies (two in optimal conditions). Furthermore, if a pipeline had been involved, a standard Chorus implementation would have incurred considerable extra overhead in terms of domain crossings, context switches and copies.

At the sink machine, data is received at the network interface card and the protocol processing is performed by a lightweight thread in supervisor space before the buffer is transferred to the sink device. The data does not need to cross into user address space at the sink machine. Thus the cost incurred is zero domain crossings, one context switch (to allow the user's rthandler to run) and two copies (or one copy and a DMA). Using standard Chorus mechanisms, the cost here would be same as at the source: i.e. four domain crossings, a context switch and four copies.

7 RELATED WORK

The split level scheduling scheme which has significantly influenced our design is described in [Govindan,91]. However, in Govindan's scheme, there is no end-to-end QoS control and, although threads are correctly 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.

In Govindan's scheme, real-time threads alternate between two states: workahead (scheduled with a time sliced round robin policy) and critical (scheduled with an earliest deadline first policy). There is also a class of non real-time interactive threads which take precedence over real-time threads in the workahead state but not the critical state. Users explicitly notify the system, in anticipation of message arrivals, the times at which workahead threads should become critical. Our scheme, on the other hand, does not require an explicit, user visible, distinction between workahead and critical threads as the deadlines of real-time threads are implicitly derived from QoS statements supplied at connect

time. The computation performed in Govindan's scheme by workahead and interactive threads is, in our system, performed by non-real-time threads scheduled according to the standard Chorus priority/ round robin schemes.

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 and connector abstractions perform a similar role but are applicable to remote as well as local communications. Furthermore, MMSs use a passive read/write based I/O interface which results in more thread synchronisations than our rthandler approach.

Work on real-time extensions to Mach consisting of real-time threads, real-time synchronisation primitives and time driven scheduling is described in [Tokuda,90]. These extensions are intended for real-time computing in general rather than multimedia support in particular. The thread model allows the creation of both periodic and aperiodic threads. The scheduling mechanism is derived from the ARTS kernel [Tokuda,89] and permits hard real-time scheduling based on classic techniques [Liu,73]. Again, for end-to-end continuous media support, the main limitation of this work is the lack of declarative QoS 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 [Tokuda,92] in the ARTS kernel has addressed QoS issues and continuous media but it is still not clear how scheduling and communications interact.

8 CONCLUSIONS

We have presented a low level API and implementation scheme for distributed multimedia support in a Chorus based micro-kernel environment. As the basic abstractions of Chorus are comparable to those of other current micro-kernels such as Mach and Amoeba we expect that it should be possible to extend other micro-kernels in a similar way.

At the present time, we have established an experimental infrastructure consisting of two 80386 based PCs running Chorus. The PCs are equipped with VideoLogic audio/ video/ JPEG compression

boards. The machines are now equipped with ATM interface cards and connect to a local ATM network. This is enabling us to extend our investigation of QoS issues and resource reservation to the network and work on an overall architecture for QoS. Our plans for establishing an ATM based infrastructure and end-to-end QoS architecture are detailed in [Campbell,92b].

We are currently working on the implementation of both the transport and scheduling aspects of the design presented in this paper. The scheduling implementation is based around the Chorus scheduling classes facility mentioned in section 2. This provides an ideal implementation basis for our kernel level scheduler. The user level scheduling and thread support aspects of the design are based on a simple non-timesliced package which we are modifying to accept software interrupts and a shared memory interface to the new kernel scheduling class. As mentioned above, our transport implementation is based on a pre-existing protocol. We are currently porting this to Chorus from the transputer based platform for which it was initially designed.

There remain a number of important issues which we have not yet addressed. One relates to the provision of an admission algorithm for flows. Another involves the extension of connections and rtports to operate in the context of port groups as supported by standard Chorus. This latter extension is non trivial due to the inherent problems of connection oriented multicast [Cramer,92]. A third issue is the provision of a higher level distributed programming platform which we are currently investigating in co-operation with researchers at CNET, France. The platform will be based on the ISO's emerging standards for Open Distributed Processing with extensions for real-time synchronisation, continuous media and QoS [Coulson,94].

A final issue we intend to address is the applicability of our extensions to hardware architectures beyond the standard uni-processor bus-based systems we are currently using. This is an important issue as it is well acknowledged that multimedia workstations require additional hardware support and that bus-based interconnects do not scale well [Scott,93]. The extension to a shared memory multiprocessor architecture should be relatively straightforward. Chorus already incorporates multiprocessor support and our lightweight threads package can also take advantage of this. To investigate the applicability of our scheme to message passing multiprocessors, we intend to port our system to a star configured switch-based multimedia workstation currently being built at Lancaster. The intention here is to retain our programming abstractions but to extend the low level implementation to accommodate non-local devices realised as specialised media specific processing nodes connected via the system switch.

ACKNOWLEDGEMENT

The research reported in this paper was funded 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

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,91a Anderson, D.P., and P. Chan, "Toolkit Support for Multiuser Audio/Video Applications", Proc. Second International Workshop on Network and Operating System Support for Digital Audio and Video, IBM ENC, Heidelberg, Germany, 1991.

Anderson,91b Anderson, T.E., Bershad, B.N., Lazowska, E.D. and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism", *Proc. Thirteenth ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, CA, USA, pp 95-109, October 1991.

Campbell,92a Campbell, A., Coulson G., Garcia F., and D. Hutchison, “A Continuous Media Transport and Orchestration Service”, Proc. ACM SIGCOMM ‘92, Baltimore, Maryland, USA, August 1992; also ACM Computer Communication Review, Vol 22, No 4, pp 99-110, October 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. 4th International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, UK, November 1993; also available as MPG-93-22, Computing Department, Lancaster University, Lancaster LA1 4YR, UK, 1993.

Clark,87 Clark, D.D., Lambert, M.L., and L. Zhang, "NETBLT: A High Throughput Transport Protocol", Computer Communication Review, Vol 17, No 5, pp 353-359, 1987.

Coulson,93 Coulson, G., and Blair, G.S., “Micro-kernel Support for Continuous Media in Distributed Systems”, To appear in computer Networks and ISDN Systems, 1994; also available as Internal Report No. MPG-93-04 Department of Computing, Lancaster University, Lancaster LA1 4YR, UK, 1993.

Coulson,94 Coulson, G., Blair, G.S., Stefani, J.B., Horn, F. and Hazard, L., "Supporting the Real-time Requirements of Continuous Media in Open Distributed Processing", Internal Report, MPG-93-10, Lancaster University, Jan 93; to be published in Computer Networks and ISDN Systems Special Issue on Open Distributed Processing, 1994.

Cramer,92 Cramer, A., Farber, M., McKellar, B. and Steinmetz, R., "Experiences with the Heidelberg Multimedia Communication System: Multicast, Rate Enforcement and Performance", Proc. IFIP Conference on High Speed Networks, Liege, Belgium, 1992.

Forin,90 Forin, A., Golub, D. and Bershad, B., "An I/O System for Mach 3.0", Internal Report, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA, 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.

Hermann,88 Herrmann, F., Armand, F., Rozier, M., Gien, M., Abrossimov, V., Boule, I., Guillemont, M., Leonard, P., Langlois, S. and W. Neuhauser, “CHORUS, A New Technology for Building UNIX Systems”, Proc. EUUG Autumn Conference, Cascais, Portugal, pp 1-18, October 3-7 1988.

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.

Reed,79 Reed, D.P. and Kanodia, R.K., "Synchronisation with Eventcounts and Sequences", CACM, Vol 22, No 2, pp 115-123, February 1979.

Robin,94 Robin, P., Coulson, G., Campbell, A., Blair, G. and Papathomas, M., "Implementing a QoS Controlled ATM Based Communications System in Chorus", Internal Report, MPG-94-05, Lancaster University, 1994.

Scott,93 Scott, A.C., Shepherd, W.D. and Lunn, A.S., "The LANC - Bringing ATM to the workstation", 4th IEE Conference on Telecommunications 1993 (ICT'93), Manchester, April 1993.

Shepherd,91 Shepherd, W.D., Coulson, G., García, F., and D. Hutchison, "Protocol Support for Distributed Multimedia Applications", Proc. Second International Workshop on Network and Operating Systems Support for Digital Audio and Video, Heidelberg, Germany, 1991.

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 (North-Holland), 1990.

Thekkath,93 Thekkath, C.A., Nguyen, T.D., Moy, E. and Lazowska, E., "Implementing Network Protocols at User Level", IEEE Transactions on Networking, Vol 1, No 5, pp 554-565, October 1993.

Tokuda,89 Tokuda, H. and Mercer, C.W., "ARTS: A Distributed Real-time Kernel", ACM Operating Systems Review, Vol 23, No 3, July 1989.

Tokuda,90 Tokuda, H., Nakajima, T. and Rao, P., "Real-time Mach: Towards a Predictable Real-time System", Proc. Usenix 1990 Mach Workshop, Usenix, October 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.