EXTENDING THE CHORUS MICRO-KERNEL TO SUPPORT CONTINUOUS MEDIA APPLICATIONS

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 [1] is addressing this deficiency in the state of the art by extending a commercial micro-kernel (i.e. Chorus [2]) to support continuous media applications alongside standard UNIX applications (Chorus already supports UNIX applications through the provision of a UNIX subsystem). 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.

Presented at the 4th International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, UK, November 1993

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 also revisited 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’ (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 which 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. 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. [3], [4]).

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*). The new abstractions are illustrated in figure 1 and described below.

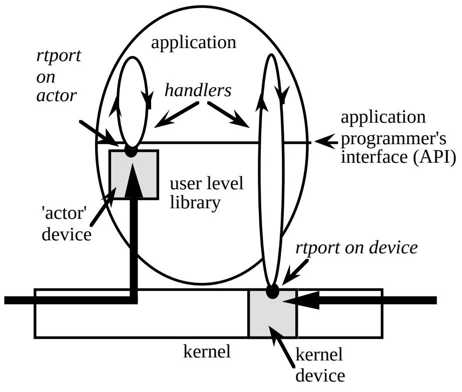

Fig. 1. Devices, Rtports and Handlers

rtports: these are extensions of standard Chorus ports which serve as access points for continuous media communications. Rtports have an associated QoS which defines constraints on communication. They also provide direct access to buffers by the application thus minimising copy operations.

devices: these are producers, consumers and filters of real-time data which support the creation of rtports and provide the memory for their buffers. Devices may be either drivers for physical devices or Chorus actors containing application code. Devices can be implemented in either user space or system space.

handlers: these are user defined C functions which manipulate real-time data coming from or going to an rtport. Handlers are (optionally) attached to rtports and are upcalled on real-time threads associated with the rtport to notify the application that data is required/available, and to obtain/ deliver continuous media data from/to the rtport's associated buffer.

QoS controlled connections: these are end-to-end communication channels with a specific QoS. A connection is established between a source and a sink rtport according to a given QoS specification. Flows are then realised as the combination of the QoS controlled connection and its associated handlers. QoS controlled connections are *active* in the sense that they (rather than the application itself) initiate data collection and data delivery from/to the source/sink application respectively through the invocation of handlers.

In addition to these features, our design includes facilities for bounded latency messaging, exception handling to deal with QoS degradations and means for dynamically re-negotiating the QoS of an open connection. It also allows pipelines of 'software signal processing' modules to be configured for local continuous media processing. Full details of the continuous media API are specified in [1].

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.

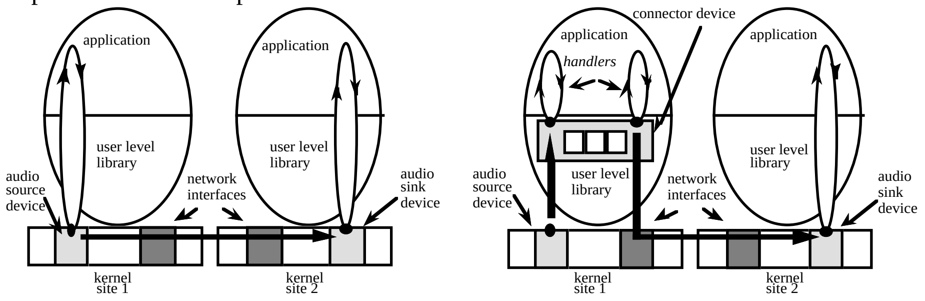

Fig. 3. Two Application Scenarios

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 handlers which are executed on a user mode 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 handler 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 Issues

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 user level threads wherever possible 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:-

- System threads are kernel supported threads which run in supervisor mode,
- Kernel threads are kernel supported threads which run in user mode,
- User threads are implemented in the libflow library and multiplexed on top of kernel threads.

All three classes of thread are non time-sliced but pre-emptive.

Our real-time scheduling system uses the earliest deadline first policy [5]. However, we do not attempt to provide absolute guarantees that deadlines will be met; the guarantee QoS parameter is treated as a soft rather than an absolute requirement. This means that QoS commitments can be revoked when the system becomes overloaded. It is, however, possible to bound overloads by introducing an admission algorithm and this would be a simple and natural extension to our current design.

Non real-time threads are scheduled according to standard Chorus policies (e.g. 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 thread facilities though the scheduling classes mechanism already described in section 2.

The implementation architecture for real-time thread scheduling is a split level scheme [6] consisting of a single kernel scheduler (KLS) and multiple co-operating user level thread schedulers (ULS), one in each actor. Each actor multiplexes user 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.

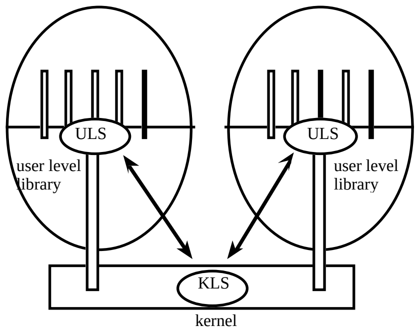

Fig. 4. Split Level Scheduling Architecture

The scheme maintains the following invariants with respect to the two types of scheduler:-

i) each user scheduler runs the user thread in its actor with the earliest deadline,
ii) the kernel scheduler runs a kernel thread which is executing in the actor with the globally earliest user 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 [6]. In the implementation of the shared memory area, each kernel thread (referred to as a *virtual processor)* has an associated context structure [7] which contains information such as current user thread context, next runnable thread, global deadline and next asynchronous event time value. Each context is mapped into kernel space and used to exchange/share information with the kernel, thus avoiding unnecessary system calls. For example, the global time value is mapped in each virtual processor (read only) avoiding the cost of a system call to get the time value. The global deadline value of each virtual processor and the value of the next asynchronous event are read on rescheduling operation by the kernel to compute the next thread 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 handler threads have deadlines earlier than any other user thread in any other actor, then these threads can be switched freely at the user level while their supporting kernel thread 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.

Connections involving physical, kernel managed, devices use system threads to obtain data from devices. To ensure fair processor allocation, it is important that these threads are scheduled consistently with user threads in the various user level actors. To achieve this, the kernel itself is treated similarly to a user actor for scheduling purposes and the system threads within it are scheduled analogously to user threads in real actors. Thus, to schedule system threads, the kernel operates a module equivalent to a user level scheduler which interacts with the global kernel scheduler in exactly the same way as real user level schedulers.

Each actor must ensure that it responds in a timely fashion, not only to the deadlines of its own user 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) buffer or buffer fragment 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.

It is essential that such events are notified to actor 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 actor's virtual processor to jump to an entry point in its user level scheduler. When this happens, the user level scheduler saves the current user thread context and schedules the appropriate user thread to perform the required action. The software interrupt implementation uses an asynchronous event list managed by each virtual processor. This list is scanned on each kernel and user level rescheduling operation and any pending interrupt handlers are executed.

Other researchers have proposed the use of 'scheduler activations' [12] 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 execution context 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 kernel threads 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 user thread performs a blocking system call and thus blocks its underlying kernel thread. As we prefer only one kernel thread per actor (our scheduling scheme is optimal with this constraint), other user 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* [7]. These calls return immediately and thus allow the calling kernel thread to resume acting as a virtual processor for user level 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 [8]. 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 [9] 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 two 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. A final point is that, as recommended by researchers in the area (e.g. [10]), the 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. However, user mode operation is preferred for connections involving actor devices as this results in a simplification of the scheduling scheme. In user mode operation, 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 the number of kernel calls per user level buffer. To achieve this, the user mode implementation batches data to minimise 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 [11] are encouraging.

To realise the active semantics of connections, connections have dedicated threads at each end. These are user threads for connections whose end rtports are implemented in libflow, and system threads for those whose rtports are kernel managed. The source thread is responsible for continually executing the user handler, obtaining data (either from the handler or directly from a physical device), 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 a user handler or by delivering data directly to a device. Both threads are scheduled to run periodically so that the connection's rate guarantees can be upheld.

Note that the user library transport implementation and handler 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 handler execution times. Secondly, the handler scheme eliminates any need for synchronisation between the application and a distinct transport entity: a single seamless thread of execution subsumes both these activities.

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 user thread which repeatedly calls the source handler with a particular buffer address and then calls the sink handler with the same address. This single mechanism serves to implement both the data transfer and the delivery notification aspects of the communication. 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 copies and one virtual memory remap at best. Our strategy is to temporarily 'loan' (via a virtual memory remap) network buffers to user actors when data arrives. Eventually, of course, the network driver will need to reclaim its buffer but in many cases the user will have processed and forwarded the data before this becomes necessary (otherwise the user actor can choose to copy the data somewhere safe). We are currently working on a driver-to-user protocol designed to effectively compromise between the network driver's need to always have available buffers while eliminating the need to copy in the majority of cases.

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 to simply pass the address of the re-mapped source buffer along with the event notification.

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 system thread as described above. Having performed the copy (or supervised the DMA), the system thread delivers a software interrupt to the sink actor’s user level scheduler. The scheduler, on receiving the interrupt, schedules a user thread to execute the sink handler. When it runs (as determined by its deadline) the user thread executes the user’s code and eventually blocks on a user level synchronisation primitive. Execution of this primitive causes the user thread of the second connection’s source rtport to unblock and a context switch to take place (this thread will have been previously blocked in the source rtport’s handler waiting for data). 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 user thread to the second. At this point the user thread executing the source handler starts to run the transport protocol and eventually issues a system call to the network device driver t 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 pipeline had been involved, 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 system thread in kernel 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 [6]. 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 handlers 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 repeated user notification of critical times as the system has prior knowledge of message arrival patterns from QoS statements supplied at connect time. We are also able to manage with just two classes of thread: real-time threads (scheduled earliest deadline first) and non real-time threads. The workahead/critical distinction is not required because the deadlines of real-time threads are known at all times. 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 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 handler approach.

Work on real-time extensions to Mach consisting of real-time threads, real-time synchronisation primitives and time driven scheduling is described in [13]. 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 [14] and permits hard real-time scheduling based on classic techniques [5]. 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 [15] in the ARTS kernel has addressed QoS issues and continuous media but it is still not clear how scheduling and communications interact.

8 Conclusions and Future Work

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 also have Ethernet cards and are connected by a dedicated Ethernet. Due to the bottleneck presented by the PC’s ISA bus, Ethernet serves as an adequate MAC layer technology for our current research which is focusing on end-system rather than network QoS. However, in the future we intend to use 80486 machines with higher bandwidth EISA busses and ATM interface cards. This will enable 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 [16].

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 [17]. 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 [18].

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 [19]. The extension to a shared memory multiprocessor architecture should be relatively straightforward. Chorus already incorporates multiprocessor support and our user level 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

1. Coulson, G., and Blair, G.S., “Micro-kernel Support for Continuous Media in Distributed Systems”, Internal Report No. MPG-93-04 Department of Computing, Lancaster University, Lancaster LA1 4YR, UK., 1993.

2. 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.

3. 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.

4. 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.

5. Liu, C.L. and Layland, J.W., “Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment”, Journal of the Association for Computing Machinery, pp 46-61, February 1973.

6. 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.

7. 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.

8. 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.

9. 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.

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

11. 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.

12. 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.

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

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

15. 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.

16. 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.

17. 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.

18. Coulson, G., Blair, G.S., Davies, N. and N. Williams, “Extensions to ANSA for Multimedia Computing”, Computer Networks and ISDN Systems, 25, pp 305-323, 1992.

19. 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.