The Design and Implementation of an Operating System to Support Distributed Multimedia Applications

Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden

Abstract

Support for multimedia applications by general purpose computing platforms has been the subject of considerable research. Much of this work is based on an evolutionary strategy in which small changes to existing systems are made. The approach adopted here is to start *ab initio* with no backward compatibility constraints. This leads to a novel structure for an operating system. The structure aims to decouple applications from one another and to provide multiplexing of all resources, not just the CPU, at a low level. The motivation for this structure, a design based on the structure, and its implementation on a number of hardware platforms is described.

I. Introduction

GENERAL purpose multimedia computing platforms should endow text, images, audio and video with equal status: interpreting an audio or video stream should not be a privileged task of special functions provided by the operating system, but one of ordinary user programs. Support for such processing on a platform on which other user applications are running, some of which may also be processing continuous media, cannot be achieved using existing operating systems — it requires mechanisms that will consistently share out resources in a manner determined by both application requirements and user preferences.

Continuous media streams have two important properties. The first property is that their fidelity is often dependent upon the timeliness with which they are presented. This *temporal property* of continuous media imposes the requirement that code which manipulates the media data may need to be scheduled within suitable windows of time. The second property is that they are often tolerant of the loss of some of their information content, particularly if it is known how the data is to be used (e.g. many compression schemes rely on human factors to achieve high compression rates). This *informational property*, without regards to its exact nature, may be exploited by systems which handle continuous media.

The properties of the streams can be extended to the applications which process them; we have temporal requirements for such applications which are stronger than traditional data processing applications, and informational requirements which are weaker.

In order for an operating system to support both traditional and multimedia applications, a wider range of facilities than is found in current operating systems needs to be provided. This paper describes an operating system, called Nemesis, whose goal is to provide this range of facilities. This work was carried out as part of the Pegasus project [1], an ESPRIT Basic Research Activity.

The underlying assumptions made in the Pegasus project are:

1. General purpose computing platforms will *process* continuous media as well as simply capture, render and store them.
2. Users will run many applications which manipulate continuous media simultaneously.
3. An application manipulating continuous media will make varying demands on resources during its execution.
4. The application mix and load will be dynamic.

In multimedia systems which capture, display, and store continuous media the range of applications is constrained (although by no means currently exhausted). The introduction of the processing of continuous media in real time adds an important new dimension. If the current situation is typified as using processors to control continuous media, future systems will be typified as systems in which the data types operated on have been extended to include continuous media streams [2]. What we are concerned with here is to provide an environment in which such applications can be developed and run.

Traditional general purpose operating systems support the notion of a virtual processor interface in which each application sees its own virtual processor – this provides a method for sharing the real processor. However, each virtual

Ian Leslie, Richard Black, Paul Barham and Robin Fairbairns are with the University of Cambridge Computer Laboratory, Cambridge UK.
Derek McAuley is with the Department of Computer Science, University of Glasgow, UK.
Timothy Roscoe is with Persimmon IT Inc, Durham, NC.
David Evers is with Nemesys Research Limited, Cambridge UK.
Eoin Hyden is with AT&T Bell Labs, NJ.

processor sees a performance which is influenced by the load on the other virtual processors, and mechanisms to control this interference are generally not available. Multimedia applications require such mechanisms.

One way of controlling this interference is by providing multiple real processors. For example, many multimedia applications (or parts thereof), run on processors on peripheral cards so that the main processor is not involved. Moreover, the code running on the peripheral is likely to be embedded and there is no danger of competing applications using the peripheral at the same time. The same approach is also used in mainframes where the use of channel processors reduces the I/O demands on the central processors, in particular ensuring that the central processors do not get overloaded by I/O interrupts.

Our aim in Nemesis is to allow a general purpose processor to be used to provide the functions one would find in a specialised DSP peripheral while providing the same control of interference across virtual processors as can be achieved with distinct hardware. We wish to retain the flexibility of the virtual processor system so that resources can be used more efficiently than in a dedicated-peripheral approach.

In approaching the design of an operating system with these goals, the immediate question of revolution versus evolution arises. Should one attempt to migrate a current operating system (or indeed use a current operating system) in order to meet these goals, or should one start afresh? The reasons why current general purpose operating systems are not appropriate are well established. Similarly, hard real time solutions which require static analysis are not appropriate in a situation where the application mix is dynamic.

General purpose operating systems with "real time threads" in which the real time behaviour is provided by static priority are also inappropriate, unless one is running a single multimedia application or can afford to perform an analysis of the complete system in order to assign priorities. A better solution might be to take an existing operating system and modify its scheduling system to support multimedia applications – perhaps one reason for the difficulty in performing such a scheduler transplant is that knowledge of the characteristics of the scheduler often migrates to other components making the effect of replacement unpredictable.

This, together with our view that processor scheduling is not the only important aspect of operating system support for multimedia applications has lead us to start from scratch. As we describe below, providing a realisation of a virtual processor that has the properties that we require has profound implications on the complete structure of the operating system.

The main theme guiding the design of Nemesis is multiplexing system resources at the lowest level – in the case of the processor, this multiplexing system is the scheduling algorithm. However, it is the multiplexing of all resources, real or virtual, which has determined the fundamental structure of Nemesis.

This has given rise to a system in which as much functionality as possible executes in the domain of the application[1]. This includes code that in a traditional microkernel would execute in a shared server. It should be emphasised that this need not change the interface seen by application programmers. The API seen by a programmer is often a thin layer of library code supplying a veneer over a set of kernel traps and messages to server processes – whereas in Nemesis the majority of the functionality would be provided by a shared library. As an example, a POSIX API in a Nemesis domain can be provided over a POSIX emulation which mostly runs within the applications domain.

In Nemesis a service is provided as far as possible by shared library code and the design of a service will aim to minimise the number of changes in protection domain. To aid in the construction of such services, references between the various parts of the code and data are simplified by the use of a *single address space* with protection between domains provided by the access control fields of address translations.

After a discussion of quality of service management and application crosstalk in section II, the structure of the Nemesis kernel, the virtual processor model and the event mechanism, is described in detail in section III. Events are a native concept in Nemesis. Events can be used to support an implementation of event counts and sequencers, and in practice all domains currently use this mapping. Other synchronisation primitives can be built on top of event counts and sequencers when required.

Scheduling amongst and within domains is described in section IV. Two domain scheduling algorithms are presented, one in detail, one briefly. Although scheduling is an important aspect of supporting multimedia applications, Nemesis does not take the view that there is a correct scheduling algorithm; indeed the structure of Nemesis is designed to make the use of alternative scheduling algorithms straightforward.

Two aspects of the system are only briefly described: the linkage model for the single address space and the interdomain communication mechanisms. A system in which code implementing operating system services executes within the application domain gives rise to problems of linking the pieces of code and data required and of providing safe and efficient sharing of this code and data. These problems, as well as those directly attributable to the use of a single address space, are discussed in section V.

Higher layer inter-domain communication (IDC) systems can be built over events. Section VI presents the system used for interdomain invocations based on an RPC model. This section also presents the bulk transfer mechanism used in Nemesis, in the context of support for networking I/O.

[1]The concept of domains in Nemesis will be explained in section III, for the moment they can be thought of as analogous to Unix processes.

The current state of the implementation and systems built over it are described along with some early conclusions in section VII.

II. The Model of Quality of Service (QoS) Management

Managing quality of service (QoS) in an operating system can be done in a number of ways. At one extreme one can make hard real time guarantees to the applications, refusing to run them if the hard real time guarantees cannot be made. At the other extreme one can hope for the best by providing more resource than one expects to be used.

In between are a range of options which are more appropriate for multimedia systems. In general the approach is to provide probabilistic guarantees and to expect applications to monitor their performance and adapt their behaviour when resource allocation changes.

Some QoS architectures, for example [3] assume a context in which applications specify their QoS requirements to a layer below them which then determines how that requirement is to be met and in turn specifies derived QoS requirements to the next layer below. This is a particularly bad approach when the layers are performing multiplexing (e.g. a single thread operating on behalf of a number of applications) since great care must be taken to prevent QoS crosstalk. Even when the processing is not multiplexed we cannot escape the need to have a recursive mapping of QoS requirements down the service stack. This is not a practical approach; providing this mapping is problematic, particularly when the application itself is unlikely to understand its QoS requirements, and when they change in time.

A. Feedback for QoS Control

Our approach is to introduce a notion of feedback control. This is an adaptive approach, in which a controller adjusts application QoS demands in the light of the observed performance. This should be distinguished from the more usual type of feedback where applications degrade gracefully when resources are over committed. This is shown schematically in figure 1.

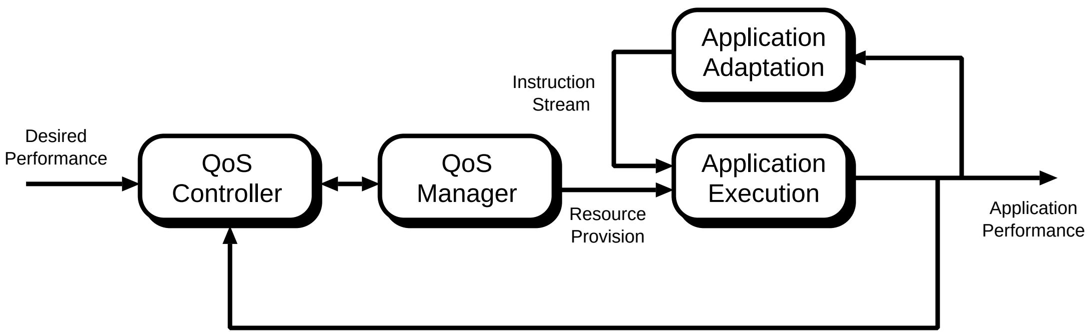

Fig. 1
QoS Feedback Control

The QoS Controller dictates the policy to be followed and can be directly dictated by the user, by an agent running on the user's behalf, or more normally both. The QoS Manager implements the allocation of resources to try and achieve the policies dictated by the QoS Controller, ensures their enforcement by informing the operating system and applications so they can adapt their behaviour.

This scheme is directly analogous to many window systems, where the Window Manager and Server are the counterparts of the QoS Controller and Manager. In a window system, applications are made aware of the pixels they have been allocated by the server and adapt accordingly; the server enforces these allocations by clipping; and users, by using a preference profile and resizing windows directly, interact with the window manager (their agent) to express their desired policies.

This approach allows applications (and application writers) to be free from the problem of determining exactly what resources an application requires at the cost of requiring them to implement adaptive algorithms. However a useful side effect is that it thus simplifies the porting of applications to new platforms.

Where does this lead in relation to providing QoS guarantees within the operating system? Can everything be left to the controller, manager and applications? A brief consideration of the feedback system leads to a conclusion: the forward performance function, that is, the application performance for a given set of resources and instruction stream, need not necessarily be *predictable* to obtain the desired performance but it must be *consistent*. Note that while efficiency and speed of execution are desirable, they are not as important to the stability of the QoS control system as consistency.

Consistency in turn requires that resources are accounted correctly to the applications that consume them, or to be more accurate, to the applications that cause them to be consumed, and that QoS crosstalk between applications be kept to a minimum.

B. QoS Crosstalk

When dealing with time-related data streams in network protocol stacks, the problem of *Quality of Service crosstalk* between streams has been identified. QoS crosstalk occurs because of contention for resources between different streams multiplexed onto a single lower-level channel. If the thread processing the channel has no notion of the component streams, it cannot apply resource guarantees to them and statistical delays are introduced into the packets of each stream. To preserve the QoS allocated to a stream, scheduling decisions must be made at each multiplexing point.

When QoS crosstalk occurs the performance of a given network association at the application level is unduly affected by the traffic pattern of other associations with which it is multiplexed. The solution advocated in [4],[5] is to multiplex network associations at a single layer in the protocol stack immediately adjacent to the network point of attachment. This allows scheduling decisions to apply to single associations rather than to multiplexed aggregates. While this particular line of work grew out of the use of virtual circuits in ATM networks, it can also be employed in IP networks by the use of packet filters [6],[7] and fair queueing schemes [8].

Analogously, application QoS Crosstalk occurs when operating system services and physical resources are multiplexed among client applications. In addition to network protocol processing, components such as device I/O, filing systems and directory services, memory management, link-loaders, and window systems are needed by client applications. These services must provide concurrency and access control to manage system state, and so are generally implemented in server processes or within the kernel.

This means that the performance of a client is dependent not only on how it is scheduled but also on the performance of any servers it requires, including the kernel. The performance of these servers is in turn dependent on the demand for their services by other clients. Thus one client's activity can delay invocations of a service by another. This is at odds with the resource allocation policy, which should be attempting to allocate resources among applications rather than servers. We can look upon scheduling as the act of allocating the real resource of the processor. Servers introduce virtual resources which must also be allocated in a manner consistent with application quality of service.

C. Requirements on the Operating System

Taking this model of QoS management, including its extension to cover all resources, gives rise to the following requirements:

- The operating system should provide facilities to allow the dynamic allocation of resources to applications.
- The operating system should ensure that the consumption of resources is accounted to the correct application.
- The operating system should not force applications to use shared servers where applications will experience crosstalk from other applications.

The first of these requirements can be met by what we have called the QoS manager. This runs occasionally as requests to change the allocation of resources are made. It does not run for any other reason and, to borrow a phrase from communications, can be said to run out of band with respect to the application computation.

The second and third of these requirements are strongly related. Both are concerned with in band application computation and, again to use the language of communication systems, lead to a philosophy of a low level multiplexing of all resources within the system. This consideration gives rise to a novel structure for operating systems.

III. Structural Overview

Nemesis is structured to provide fine-grained resource control and to minimise application QoS crosstalk. To meet these goals it is important to account for as much of the time used by an application as possible, to keep the application informed of its resource use, and enable the application to schedule its own subtasks. At odds with this desire is the need for code which implements concurrency and access control over shared state to execute in a different protection domain from the client (either the kernel or a server process).

A number of approaches have been taken to try and minimize the cost of interacting with such servers. One technique is to support thread migration; there are systems which allow threads to undergo protection domain switches, both in specialised hardware architectures [9] and conventional workstations [10]. However, such threads cannot easily be scheduled by their parent application, and must be implemented by a kernel which manages the protection domain boundaries. This kernel must as a consequence, provide synchronisation mechanisms for its threads, and applications are no longer in control of their own resource tradeoffs.

The alternative is to implement servers as separate schedulable entities. Some systems allow a client to transfer some of their resources to the server to preserve a given QoS across server calls. The Processor Capacity Reserves mechanism [11] is the most prominent of these; the kernel implements objects called *reserves* which can be transferred from client threads to servers. This mechanism can be implemented with a reasonable degree of efficiency, but does not fully address the problem:

- The state associated with a reserve must be transferred to a server thread when an IPC call is made. This adds to call overhead, and furthermore suffers from the kernel thread-related problems described above.
- Crosstalk will still occur within servers, and there is no guarantee that a server will deal with clients fairly, or that clients will correctly 'pay' for their service.

- It is not clear how nested server calls are handled; in particular, the server may be able to transfer the reserve to an unrelated thread.

Nemesis takes the approach of minimising the use of shared servers so as to reduce the impact of application QoS crosstalk: the minimum necessary functionality for a service is placed in a shared server while as much processing as possible is performed within the application domain. Ideally, the server should *only* perform privileged operations, in particular access control and concurrency control.

A consequence of this approach is the desire to expose some server internal state in a controlled manner to client domains. Section V describes how the particular use of interfaces and modules in Nemesis supports a model where all text and data occupies a single virtual address space facilitating this controlled sharing. It must be emphasised that this in no way implies a lack of memory *protection* between domains. The virtual to physical address *translations* in Nemesis are the same for all domains, while the *protection* rights on a given page may vary. What it does mean is that any area of memory in Nemesis can be shared, and virtual addresses of physical memory locations do not change between domains.

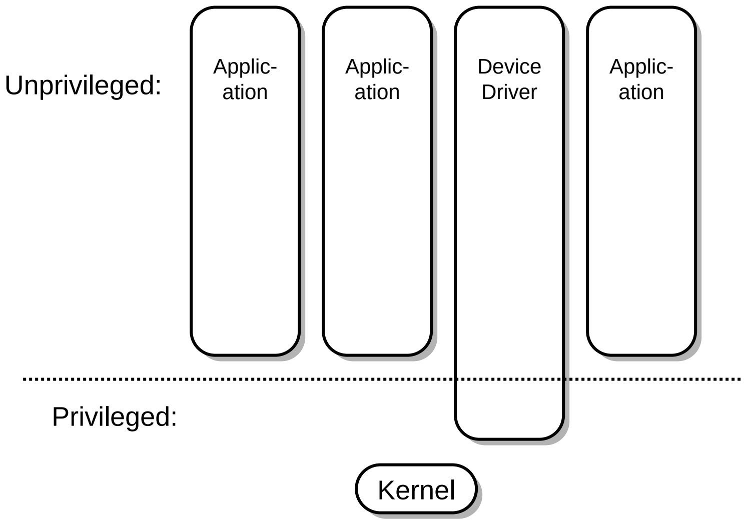

Fig. 2
NEMESIS SYSTEM ARCHITECTURE

The minimal use of shared servers stands in contrast to recent trends in operating systems, which have been to move functionality away from client domains (and indeed the kernel) into separate processes. However, there are a number of examples in recent literature of services being implemented as client libraries instead of within a kernel or server. Efficient user-level threads packages have already been mentioned. Other examples of user level libraries include network protocols [12], window system rendering [13] and Unix emulation [14].

Nemesis is designed to use these techniques. In addition, most of the support for creating and linking new domains, setting up inter-domain communication, and networking is performed in the context of the application. The result is a 'vertically integrated' operating system architecture, illustrated in figure 2. The system is organised as a set of *domains*, which are scheduled by a very small kernel.

A. The Virtual Processor Interface

The runtime interface between a domain and the kernel serves two purposes:

- It provides the application with information about when and why it is being scheduled.
- It supports user-level multiplexing of the CPU among distinct subtasks within the domain.

The key concepts are *activations* by which the scheduler invokes the domain, and *events* which indicate when and why the domain has been invoked. If each domain is considered a virtual processor, the activations are the virtual interrupts, the events the virtual interrupt status.

An important data structure associated with the virtual processor interface is the Domain Control Block (DCB). This contains scheduling information, communication end-points, a protection domain identifier, an upcall entry point for the domain, and a small initial stack. The DCB is divided into two areas: one is writable by the domain itself, the other is readable but not writable. A privileged service called the *Domain Manager* creates DCBs and links them into the scheduler data structures. The details of some of the fields in the DCB are described below.

A.1 Activations

The concept of activations is similar to that presented in [15]. When a domain is allocated the CPU by the kernel, the domain is normally upcalled rather than being resumed at the point where it lost the CPU. This allows the domain to consider scheduling actions as soon as it obtains CPU resource. The exceptional case of a resumption is only used when the domain is operating within a critical section where an activation would be difficult to cope with, entailing re-entrant handlers. The domain controls whether it is activated or resumed by setting or clearing the *activation bit* in the DCB. This can be considered as disabling the virtual interrupts.

A Nemesis domain is provided with an array of *slots* in the DCB, each of which can hold a processor context. For example, in the case of the Alpha/AXP implementation, there are 32 slots, each consisting of 31 integer and 31 floating-point registers, plus a program counter and processor status word. At any time, two of the slots are designated by the application as the *activation context* and *resume context*.

When a domain is descheduled, its processor context is saved into the activation slot or the resume slot, depending on whether the activation bit is set or not. When the domain is once again scheduled, if its activation bit is clear, the resume context is used; if the activation bit is set, the bit is cleared and an upcall takes place to a routine specified in the DCB. This entry point will typically be a user-level thread scheduler, but domains are also initially entered this way. Figure 3 illustrates the two cases.

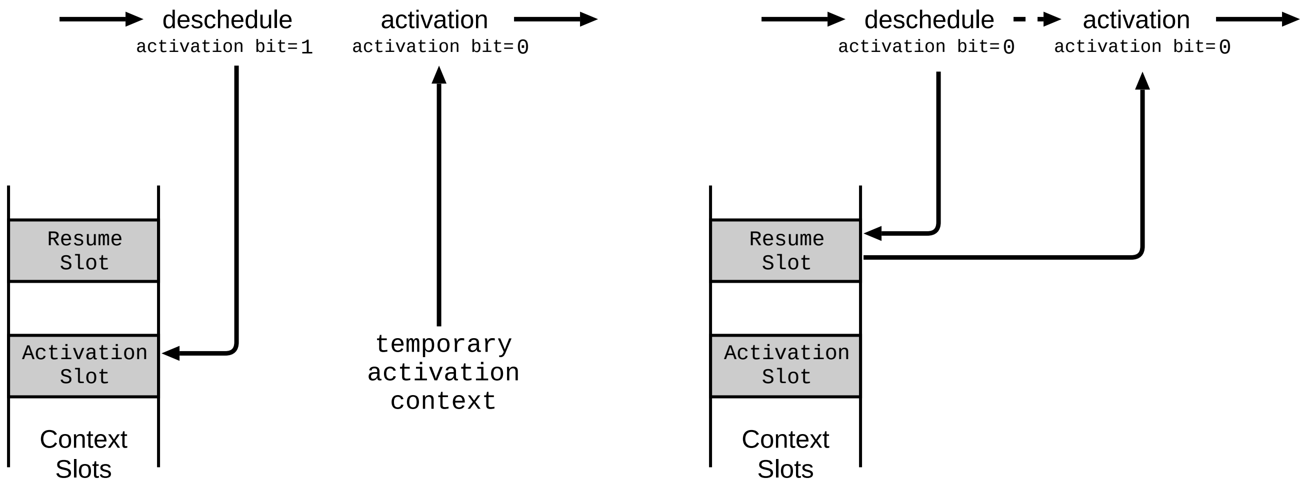

Fig. 3
Deschedules, Activations and Resumptions

The upcall occurs on a dedicated stack (again in the DCB) and delivers information such as current system time, time of last deschedule, reason for upcall (e.g. event notification) and context slot used at last deschedule. Enough information is provided to give the domain a sufficient execution environment to schedule a thread. A threads package will typically use one context slot for each thread and change the designated activation context according to which thread is running. If more threads than slots are required, slots can be used as a cache for thread contexts. The activation bit can be used with appropriate exit checks to allow the thread scheduler to be non-reentrant, and therefore simpler.

A.2 Events

In an operating system one of the requirements is to provide a mechanism for the various devices and components to communicate. The Nemesis kernel provides *events* and *event channels* to provide the underlying notification mechanism on which a range of communications channels can be constructed. There were a number of important considerations for the event mechanism in Nemesis.

- The mechanism must not force synchronous behaviour on domains which would find an asynchronous mechanism more convenient.
- It must be possible to communicate in a non-blocking manner (for example, for device drivers or servers which are QoS conscious).
- In loosely coupled multiprocessors, any explicit memory synchronisation required by the communication should only have to be performed when the mechansism is invoked. This requirement is to enable portable use of partially ordered memory systems such as an Alpha AXP multiprocessor, or the Desk Area Network.
- A thread scheduler within a domain can map communications activities to scheduling requirements efficiently; this necessitates that the communications primitives be designed in conjunction with the concurrency primitives.

These requirements dictate a solution which is asynchronous and non-blocking, and which can indicate that an arbitrary number of communications have occurred to the receiver.

The scheme is based on *events* the value of which can be conveyed from a sending domain via the kernel to a recipient domain via an *event channel*. An event is a monotonically increasing integer which may be read and modified atomically by the sending domain. This domain can request the current value be conveyed to the recipient domain by performing the kernel system call, `send()`. The recipient domain holds a readonly copy of the event which is updated by the kernel as a result of a `send()`.

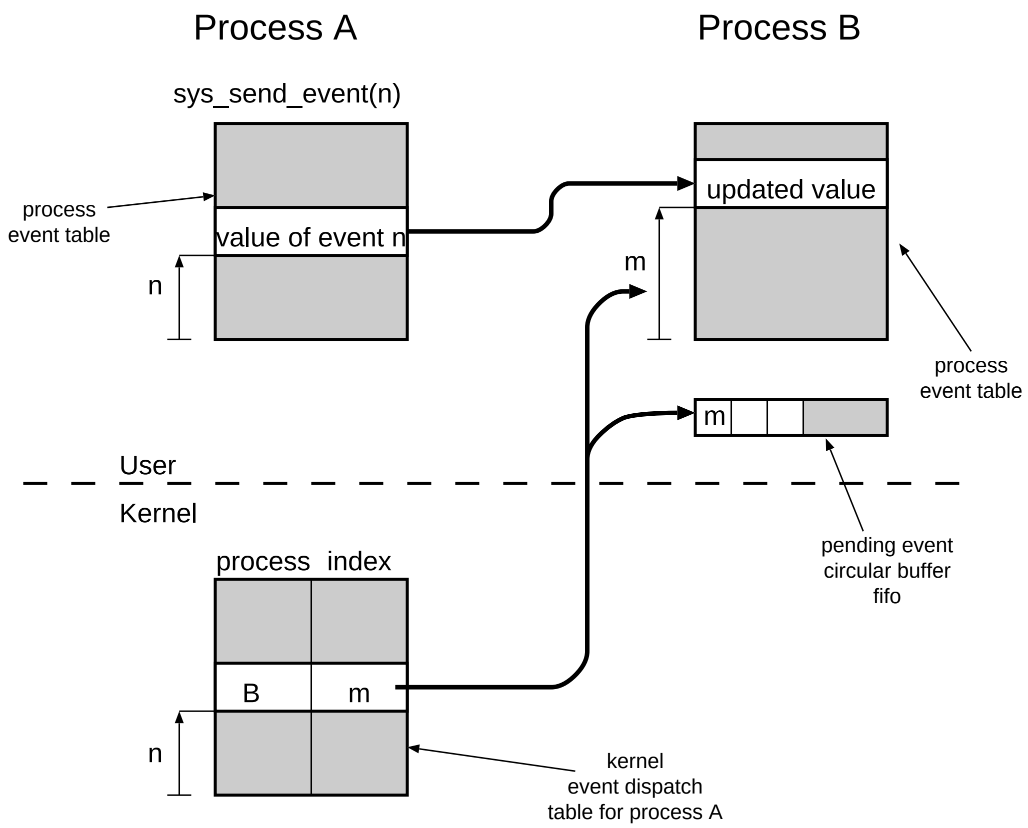

Fig. 4
EXAMPLE OF SENDING AN EVENT UPDATE

As an example, figure 4 shows the value of event number **n** in domain **A** being propagated (by the `send` system call) to domain **B** where it is event number **m**. The mapping table for event channels from **A** has the pair (**B**,**m**) for entry **n** so the kernel copies the value from **A**'s event table to the appropriate entry in **B**'s event table and places **B**'s index (in this case **m**) into **B**'s circular buffer fifo.

For each domain, the kernel has a protected table of the destinations of the event channels originating at that domain. A management domain called the *Binder*, described in section VI-A, is responsible for initialising these tables and thereby creating communication channels. Currently only "point-to-point" events are implemented, although there is nothing to prevent "multicast" events if needed in the future.

Exactly what an event represents is not known by the kernel, but only by the domains themselves. The relationship between these Nemesis events and event counts and sequencers [16] within the standard user thread library is discussed in section IV-D. Events are also used as the underlying notification mechanism supporting interrupt dispatch and inter-domain communication facilities at a higher level of abstraction – currently support is provided for inter-domain invocations (section VI), and streaming data operations (section VI-C).

A.3 Time

In a multimedia environment there is a particular need for a domain to know the current time, since it may need to schedule many of its activities related to time in the real world.

In many systems (e.g. UNIX), the current time of day clock is derived from a periodic system ticker. CPU scheduling is done based on this ticker, and the system time is updated when this ticker interrupt occurs taking into account an adjustment to keep the time consistent with universal time (UT) (e.g. using NTP [17]). In these circumstances, a domain may only be able to obtain a time value accurate to the time of the last timer interrupt and even then the value actually read may be subject to significant skew due to adjustments.

To overcome these problems in Nemesis, scheduling time and UT are kept separate. The former is kept as a number of nanoseconds since the system booted and is used for all scheduling and resource calculations and requests. The expected granularity of updates to this variable can be read by applications if required. Conversion between this number and UCT can be done by adding a system base value. It is this base value which can be adjusted to take account of the drift between the system and mankind's clock of convention. The scheduling time is available in memory readable by all domains, as well as being passed to a domain on activation.

B. Kernel Structure

The Nemesis kernel consists almost entirely of interrupt and trap handlers; there are *no* kernel threads. When the kernel is entered from a domain due to a system call, a new kernel stack frame is constructed in a fixed (per processor) area of memory; likewise when fielding an interrupt.

Kernel traps are provided to `send()` events, `yield()` the processor (with and without timeout) and a set of three variations of a "return from activation" system call, `rfa()`. The return from activation system calls perform various forms of atomic context switches for the user level schedulers. Privileged domains can also register interrupt handlers, mask interrupts and if necessary (on some processors) ask to be placed in a particular processor privileged modes (e.g. to access TLB etc).

On the Alpha AXP implementation, the above calls are all implemented as PAL calls with only the scheduler written in C (for reasons of comprehension).

Nemesis aims to schedule domains with a clear allocation of CPU time according to QoS specifications. In most existing operating systems, the arrival of an interrupt usually causes a task to be scheduled immediately to handle the interrupt, preempting whatever is running. The scheduler itself is usually not involved in this decision; the new task runs as an interrupt service routine.

The interrupt service routine (ISR) for a high interrupt rate device can therefore hog the processor for long periods, since the scheduler itself hardly gets a chance to run, let alone a user process. Such high frequency interruptions can be counter productive; [18] describes a situation where careful prioritising of interrupts led to high throughput, but with most interrupts disabled for a high proportion of the time.

Sensible design of hardware interfaces can alleviate this problem, but devices designed with this behaviour in mind are still rare, and moreover they do not address the fundamental problem: scheduling decisions are being made by the interrupting device and interrupt dispatching code, and not by the system scheduler, effectively bypassing the policing mechanism.

The solution adopted in Nemesis decouples the interrupt itself from the domain which is handling the interrupt source. Device drivers are implemented as privileged domains – they can register an interrupt handler with the system, which is called by the interrupt dispatch code with a minimum of registers saved. This ISR typically clears the condition, masks the source of the interrupt, and sends an event to the domain responsible. This sequence is sufficiently short that it can be ignored from an accounting point of view. For example, the ISR for the LANCE Ethernet driver on the Sandpiper is 12 instructions long.

Sometimes the recipient domain will actually be a specific application domain (e.g. an application which has exclusive access to a device). However, where the recipient domain of the event is a device driver domain providing a (de-)multiplexing function (e.g. demultiplexing Ethernet frame types), this domain is under the control of the QoS based scheduler like any other and can (and does) have resource limits attached.

A significant benefit of the single virtual address space approach for ISRs is that virtual addresses are valid regardless of which domain is currently scheduled. The maintenance of scatter-gather maps to enable devices to DMA data directly to and from virtual memory addresses in client domains is thus greatly simplified.

IV. Scheduling

A Nemesis scheduler has several goals: to reduce the QoS crosstalk between applications; to enable application-specific degradation under load; to support applications which need some baseline resource by providing some real guarantees on CPU allocation.

A key concept is that applications should be allocated a *share* of the processor. These shares are allocated by the QoS Manager, based on an understanding of the available resources and input from the QoS Controller (and hence from both applications and the user). A key decision was made in order to simplify the computation required by the scheduler on context switches — the QoS Manager will ensure that the scheduler can always meet its short term demands by ensuring that less than 100% of the processor is "contracted out" to domains requiring QoS.

The QoS Manager takes a long term view of the availability of resources and uses algorithms with significant hysteresis to provide a consistent guaranteed resource to the application. However, this does not imply that the system is not work-conserving – any "slack" time in the system is supplied to those applications that request it with the information that this is "optimistic" processor capacity and they should not adapt their behaviour to rely upon it.

A mechanism for specifying CPU time QoS must serve three purposes: it must allow applications, users, or user agents to specify an application's desired CPU time, enable the QoS Manager to ensure the processor resource is not over allocated and enable the scheduler to allocate processor time efficiently.

As decribed in section II, Nemesis adopts an approach in which users or user agents are expected to provide overall control (by observation) of resouce allocation. This leads to a simple QoS specification. In the case of CPU time, there is further advantage in a simple QoS specification: it reduces the overhead for the scheduler in recalculating a schedule during a context switch.

A. Scheduling Architecture and Service Model

As well as the (relatively simple) code to switch between running domains, the Nemesis scheduler has a variety of functions. It must:

- account for the time used by each holder of a QoS guarantee and provide a policing mechanism to ensure domains do not overrun their allotted time,
- implement a scheduling algorithm to ensure that each contract is satisfied,
- block and unblock domains in response to their requests and the arrival of events,
- present an interface to domains which makes them aware both of their own scheduling and of the passage of real time,
- provide a mechanism supporting the efficient implementation of potentially specialised threads packages within domains.

Applications in Nemesis specify neither priorities nor deadlines. The scheduler deals with entities called *scheduling domains*, or *sdoms*, to which it aims to provide a particular *share* of the processor over some short time frame. An sdom may correspond to a single Nemesis domain or a set of domains collectively allocated a share.

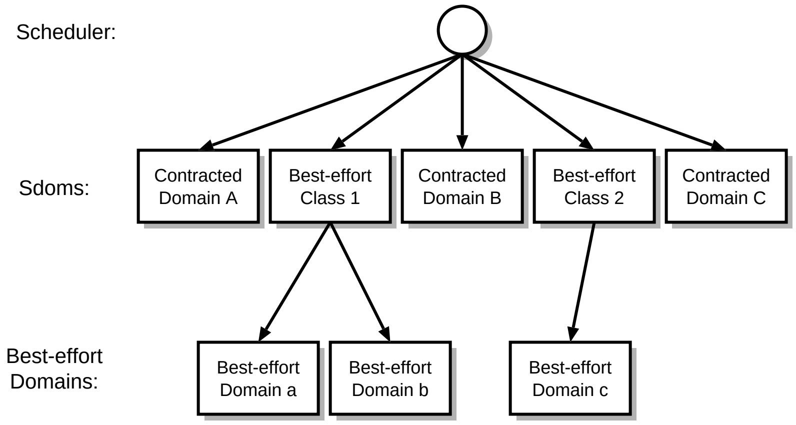

Fig. 5
Scheduling service architecture

The service architecture is illustrated in figure 5. Sdoms usually correspond to contracted domains, but also correspond to best-effort classes of domains. In the latter case, processor time allotted to the sdom is shared out among its domains according to one of several algorithms, such as simple round-robin or multi-level feedback queues. The advantage of this approach is that a portion of the total CPU time can be reserved for domains with no special timing requirements to ensure that they are not starved of processor time. Also, several different algorithms for scheduling best-effort domains can be run in parallel without impacting the performance of time-critical activities.

It has already been mentioned that within Nemesis scheduling using shares is a core concept; however, the particular scheduling *algorithm* is open to choice. The Atropos scheduler, now the "standard" Nemesis scheduler is described in detail below.

B. The Atropos Scheduler

With the Atropos scheduler shares are specified using an application dependent period. The share of the processor each sdom receives is specified by a tuple $\{s, p, x, l\}$. The slice s and period p together represent the processor bandwidth to the sdom: it will receive at least s *ticks* of CPU time (perhaps as several time slices) in each period of length p. x is a boolean value used to indicate whether the sdom is prepared to receive "slack" CPU time. l, the *latency hint*, is described below.

The Atropos scheduler *internally* uses an Earliest Deadline First (EDF) algorithm to provide this share guarantee. However the deadlines on which it operates are *not* available to or specified by the application – for the implementation of the scheduler to be simple and fast it relies on the fact that the QoS Manager has presented it with a soluble problem – this could not be ensured if the applications were allowed to specify their own deadlines.

An sdom can be in one of five states and may be on a scheduler queue:

Queue	State	
	running	running in guaranteed time
Q_r, d_r	runnable	guaranteed time available
Q_w, d_w	waiting	awaiting a new allocation of time
	optimistic	an sdom running in slack time
Q_b	blocked	awaiting an event

For each sdom, the scheduler holds a deadline d, which is the time at which the sdom's current period ends, and a value r which is the time remaining to the sdom within its current period. There are queues Q_r and Q_w of runnable and waiting sdoms, both sorted by deadline (with d_x and p_x the deadlines and periods of the respective queue heads), and a third queue Q_b of blocked sdoms.

The scheduler requires a hardware timer that will cause the scheduler to be entered at or very shortly after a specified time in the future, ideally with a microsecond resolution or better[2]. When the scheduler is entered at time t as a result of a timer interrupt or an event delivery:

1. the time for which the current sdom has been running is deducted from its value of r.
2. if r is now zero, the sdom is inserted in Q_w.
3. for each sdom on Q_w for which $t \geq d$, r is set to s, its new deadline is set to $d + p$, and it is moved to Q_r.
4. a time is calculated for the next timer interrupt depending on which of d_r or $d_w + p_w$ is the lower.
5. the scheduler runs the head of Q_r, or if empty selects an element of Q_w.

This basic algorithm will find a feasible schedule. This is seen by regarding a 'task' as 'the execution of an sdom for s nanoseconds' – as the QoS Manager has ensured that the total share of the processor allocated is less than 100% (i.e. $\sum s_i/p_i < 1$), and slices can be executed at any point during their period – this approach satisfies the conditions required for an EDF algorithm to function correctly [19],

This argument relies on two simplifications: firstly, that scheduling overhead is negligible, and secondly that the system is in a steady state. The first is addressed by ensuring there is sufficient slack time in the system to allow the scheduler to run and by not counting time in the scheduler as used by anybody. The second is concerned with moving an sdom with a share allocation from Q_b to Q_r. A safe option is to set $d := t + p$ and $r := s$; this introduces the sdom with the maximum scheduling leeway and since a feasible schedule exists no deadlines will be missed as a result. For most domains this is sufficient, and it is the default behaviour.

In the limit, all sdoms can proceed simultaneously with an instantaneous share of the processor which is constant over time. This limit is often referred to as *processor sharing*. Moreover, it can efficiently support domains requiring a wide range of scheduling granularities.

B.1 Interrupts and Latency hint

In fact, when unblocking an sdom which as been asleep for more than its period, the scheduler sets $r := s$ and $d := t + l$, where l is the latency hint. The default behaviour just described is then achieved by setting $l := p$ for most domains. However, in the case of device drivers reacting to an interrupt, faster response is sometimes required. If the device domain is using less than its share or processor capacity, the unblocking latency hint l provides a means for a device driver domain to respond to an interrupt with low latency.

The consequences of reducing l in this way are that if such an sdom is woken up when the complete system is under heavy load, some sdoms may miss their deadline for one of their periods. The scheduler's behaviour in these circumstances is to truncate the running time of the sdoms: they lose part of their slice for that period. Thereafter, things settle down.

At a high interrupt rate from a given device, at most one processor interrupt is taken per activation of the driver domain, so that the scheduling mechanism is enforcing a maximum interrupt and context switch rate. Hence, as the activity in the device approaches the maximum that the driver domain has time to process with its CPU allocation, the driver rarely has time to block before the next action in the device that would cause an interrupt, and so converges to a situation where the driver polls the device whenever it has the CPU.

When device activity is more than the driver can process, overrun occurs. Device activity which would normally cause interrupts is ignored by the system since the driver cannot keep up with the device. This is deemed to be more desirable than having the device schedule the processor: if the driver has all the CPU cycles, the 'clients' of the device wouldn't be able to do anything with the data anyway. If they could, then the driver is not being given enough processor time

[2]Such a timer is available on the DECchip EB64 board used to prototype Nemesis, but has to be simulated with a 122μs periodic ticker on the Sandpiper workstations.

by the domain manager. The system can detect such a condition over a longer period of time and reallocate processor bandwidth in the system to adapt to conditions.

B.2 Use of Slack Time

As long as Q_r is non-empty, the sdom at the head is due some contracted time and should be run. If Q_r becomes empty, the scheduler has fulfilled all its commitments to sdoms until the head of Q_w becomes runnable. In this case, the scheduler can opt to run some sdom in Q_w for which x is true, i.e. one which has requested use of slack time in the system. Domains are made aware of whether they are running in this manner or in contracted time by a flag in their DCB.

The current policy adopted by the scheduler is to run a random element of Q_w for a small, fixed interval or until the head of Q_w becomes runnable, whichever is sooner. Thus several sdoms can receive the processor "optimistically" before Q_r becomes non-empty. The best policy for picking sdoms to run optimistically is a subject for further research. The current implementation allocates a very small time quantum (122 μs) to a member of Q_w picked cyclically. This works well in most cases, but there have been situations in which unfair 'beats' have been observed.

C. Other Schedulers

The Nemesis system does not prescribe a scheduler *per se*; the Atropos scheduler is simply the one in common use. Other schedulers can be used where appropriate.

A alternative scheduler, known as the Jubilee scheduler, has been developed. It differs from the Atropos scheduler in that CPU resources are allocated to applications using a single system defined frequency. The period of this system wide frequency is known as a *Jubilee* and will typically be a few tens of milliseconds. The Jubilee scheduler has scheduling levels in a strict priority order, one for guaranteed CPU, the others for successively more speculative computations. Like Atropos, it has a mechanism for handing out slack time in the system. The use of priority is internal to the scheduler and not visible to client domains.

The fixed Jubilees remove the need for EDF scheduling and is particular suited to situations where the application load is well understood and where the a single Jubilee size can be chosen. Complete details can be found in [20].

D. Intra-domain scheduling

This section considers the implementation of an intra-domain scheduler to provide a familiar threaded environment. The intra-domain scheduler is the code which sits above the virtual processor interface. The code is not privileged and can differ from domain to domain. It may be very simple (in the case of a single threaded domain), or more complex.

The base technique for synchronization that was adopted within domains was to extend the use of the core Nemesis events already present for interdomain communication, and provide event counts and sequencers [16]. These event counts and sequencers can be purely *local* within the domain or attached to either *outbound* events (those which can be propagated to another domain using `send()`) or *inbound* events (those which change asynchronously as a result of some other domain issuing a `send()`).

D.1 Event counts and sequencers

There are three operations available on an event count **e** and two on a sequencer **s**. These are:

read(e) This returns the current value of the event count **e**. More strictly this returns some value of the event count between the start of this operation and its termination.

await(e,v) This operation blocks the calling thread until the event count **e** reaches or exceeds the value **v**.

advance(e,n) This operation increments the value of event count **e** by the amount **n**. This may cause other threads to become runnable.

read(s) This returns the current value of the sequencer **s**. More strictly this returns some value of the sequencer between the start of this operation and its termination.

ticket(s) This returns the current member of a monotonically increasing sequence and guarantees that any subsequent calls to either **ticket** or **read** will return a higher value.

In fact there is little difference between the underlying semantics of sequencers and event counts; the difference is that the **ticket** operation does not need to consider awaking threads, whereas the **advance** operation does (therefore it is wrong for a thread to **await** on a sequencer). The initial value for sequencers and event counts is zero; this may be altered immediately after creation using the above primitives. An additional operation **await_until(e,v,t)** is supported, which waits until event **e** has value **v** or until time has value **t**.

By convention an **advance** on an outbound event will cause the new value to be propagated by issuing the `send()` system call. Only **read** and **await** should be used on incoming events as their value may be overwritten at any time.

In this way, both local and interdomain synchronization can be achieved using the same interface and, unless required, a user level thread need not concern itself with the difference.

D.2 Concurrency primitives using events

In contrast to many other systems where implementing one style of concurrency primitives over another set can be expensive it is very efficient to implement many schemes over event counts.

The mutexes and conditional variables of SRC threads [21], POSIX threads, and the semaphores used in the Wanda system have all been implemented straightforwardly and efficiently over event counts. Details can be found in [20].

Implementing threads packages over the upcall interface has proved remarkably easy. A Nemesis module implementing both preemptive and non-preemptive threads packages, providing both an interface to the event mechanism and synchronisation based on event counts and sequencers comes to about 2000 lines of heavily commented C and about 20 assembler opcodes. For comparison, the POSIX threads library for OSF/1 achieves essentially the same functionality over OSF/1 kernel threads with over 6000 lines of code, with considerably inferior performance.

V. Interfaces and Invocation

The architecture introduced in section III raises a number of questions concerning the structure of applications, how services traditionally provided by servers or a kernel are provided, and how applications process their own exceptions.

In order to describe the inter-domain communciation system of Nemesis it is necessary to present some of the higher level constructs used in Nemesis to complement the single virtual address space approach. A full account can be found in [22]. The key aspects are the extensive use of typing, transparency and modularity in the definition of the Nemesis interfaces and the use of *closures* to provide comprehensible, safe and extensive sharing of data and code.

Within the Nemesis programming model there are concepts of an *interface reference*, and an *invocation reference*, the latter being obtained from the former by *binding*. An interface reference is an object containing the information used as part of binding to build an invocation reference to a particular instance of an interface. The invocation reference will be a *closure* of the appropriate type and may be either a simple pointer to library code (and local state) or to a *surrogate* for a remote interface. In the local case an interface reference and invocation reference have the same representation – a pointer to a closure – *binding* is an implicit and trivial operation.

In Nemesis, as in Spring [14], all interfaces are strongly typed, and these types are defined in an interface definition language (IDL). The IDL used in Nemesis, called Middl, is similar in functionality to the IDLs used in object-based RPC systems, with some additional constructs to handle local and low-level operating system interfaces. A Middl specification defines a single abstract data type by declaring its supertype, if any, and giving the *signatures* of all the operations it supports. A specification can also include declarations of exceptions, and concrete types.

The word *object* in Nemesis denotes what lies behind an interface: an object consists of state and code to implement the operations of the one or more interfaces it provides. A *class* is a set of objects which share the same underlying implementation, and the idea of object class is distinct from that of type, which is a property of interfaces rather than objects[3]

When an operation is invoked upon an object across one of its interfaces, the environment in which the operation is performed depends only on the internal state of the object and the arguments of the invocation. There are no global symbols in the programming model. Apart from the benefits of encapsulation this provides, it facilitates the sharing of code.

In order to overcome the awkwardness that the lack of global symbols might produce (consider having to pass a reference to a memory allocation heap on virtually every invocation), certain interfaces are treated as part of the thread context. These are known as *pervasives*. The programming model includes the notion of the currently excuting thread and the current pervasives are always available. These include exception handlers, thread operations, domain control operations and the default memory allocation heap.

The programming model is supported by a linkage model. A *stub compiler* is used to map Middl type definitions to C language types. The compiler, known as `middlc` processes an interface specification and generates a header file giving C type declarations for the concrete types defined in the interface together with special types used to represent instances of the interface.

An interface is represented in memory as a *closure*: a record of two pointers, one to an array of function pointers and one to a state record. To invoke an operation on an interface, the client calls through the appropriate element of the operation table, passing as first argument the address of the closure itself.

VI. Inter-Domain Communication

Nemesis provides a framework for building various Inter-Domain Communication (IDC) mechanisms and abstractions using events for notification and shared memory for data transfer.

One such model is inter-domain invocation; use is made of the Nemesis run-time type system to allow an arbitrary interface to be made available for use by other domains. The basic paradigm adopted is then dictated by the Middl

[3]This is different from C++ where there is no distinction between class and type, and hence no clear notion of an interface. C++ abstract classes often contain implementation details, and were added as an afterthought [23, p. 277].

interface definition language: Remote Procedure Call (RPC) with the addition of 'announcement' operations, which allow use of message passing semantics.

The use of an RPC paradigm for invocations in no way implies the traditional RPC implementation techniques (marshalling into buffer, transmission of buffer, unmarshalling and dispatching, etc.). There are cases where the RPC programming *model* is appropriate, but the underlying *implementation* can be radically different. In particular, with the rich sharing of data and text afforded by a single address space, a number of highly efficient implementation options are available.

Furthermore, there are situations where RPC is clearly not the ideal paradigm: for example, bulk data transfer or continuous media streams are often best handled using an out-of-band RPC interface only for control. This is the case with the RBuf mechanism presented in section VI-C, which employs the binding model described here and an interface-based control mechanism.

Operating systems research to date has tended to focus on optimising the performance of the communication systems used for RPCs, with relatively little attention given to the process of binding to interfaces. By contrast, the field of distributed processing has sophisticated and well-established notions of interfaces and binding, for example the "Trader" within the ANSA architecture [24]. The Nemesis binding model shares many features with the ANSA model.

This section describes briefly the Nemesis approach to inter-domain binding and invocation, including optimisations which make use of the single address space and the system's notion of real time to reduce synchronisation overhead and the need for protection domain switches, followed by an outline of the support for stream based IDC.

A. Binding

In order to invoke operations on a remote interface, to which a client has an *interface reference*, a client requires a local interface encapsulating the implementation needed for the remote invocation. This is what we have previously described as an *invocation reference*. In Nemesis IDC, an invocation reference is a closure pointer of the same type as the remote interface – in other words a *surrogate* for the remote interface.

An interface reference typically arrives in a domain as a result of a previous invocation. *Name servers* or *traders* provide services by which clients can request a service by specifying its properties. An interface reference is matched to the service request and then returned to the client.

In the local case (described in section V), an interface reference is simply a pointer to the interface closure, and binding is the trivial operation of reading the pointer. In the case where communication has to occur across protection domain boundaries (or across a network), the interface reference has to include rather more information and the binding process is correspondingly more complex.

A.1 Implicit v. Explicit binding

An implicit binding mechanism creates the state associated with a binding in a manner invisible to the client. An invocation which is declared to return an interface reference actually returns a closure for a valid surrogate for the interface. Creation of the surrogate can be performed at any stage between the arrival of the interface reference in an application domain and an attempt by the application to invoke an operation on the interface reference. Indeed, bindings can time out and then be re-established on demand.

The key feature of the implicit binding paradigm is that information about the binding itself is hidden from the client, who is presented with a surrogate interface indistinguishable from the 'real thing'. This is the approach adopted by many distributed object systems, for example Modula-3 Network Objects [25] and CORBA [26]. It is intuitive and easy to use from the point of view of a client programmer, and for many applications provides all the functionality required, provided that a garbage collector is available to destroy the binding when it is no longer in use.

On the other hand, traditional RPC systems have tended to require clients to perform an explicit bind step due to the difficulty of implementing generic implicit binding. The advent of object-based systems has recently made the implicit approach prominent for the reasons mentioned above. However, implicit binding is inadequate in some circumstances, due to the hidden nature of the binding mechanism. It assumes a single, 'best effort' level of service, and precludes any explicit control over the duration of the binding. Implicit binding can thus be ill-suited to the needs of time-sensitive applications.

For this reason, within Nemesis bindings can also be established explicitly by the client when needed. If binding is explicit, an operation which returns an interface reference does not create a surrogate as part of the unmarshalling process, but instead provides a local interface which can be later used to create a binding. This interface can allow the duration and qualities of the binding to be precisely controlled at bind time with no loss in type safety or efficiency. The price of this level of control is extra application complexity which arises both from the need to parameterise the binding and from the loss of transparency: acquiring an interface reference from a locally-implemented interface can now be different from acquiring one from a surrogate.

B. Remote Invocation

The invocation aspect of IDC (how invocations occur across a binding) is independent of the binding model used. Ideally, an IDC framework should be able to accommodate several different methods of data transport within the computational model.

RPC invocations have at least three aspects:

1. The transfer of information from sender to receiver, whether client or server
2. Signalling the transfer of information
3. The transfer of control from the sender to the receiver

Current operating systems which support RPC as a local communications mechanism tend to use one of two approaches to the problem of carrying a procedure invocation across domain boundaries: message passing and thread tunnelling.

With care, a message passing system using shared memory regions mapped pairwise between communicating protection domains can provide high throughput, particularly by amortising the cost of context switches over several invocations – in other words by having many RPC invocations from a domain outstanding. This separation of information transfer from control transfer is especially beneficial in a shared memory multiprocessor, as described in [27].

The thread tunnelling model achieves very low latency by combining all components into one operation: the transfer of the thread from client to server, using the kernel to simulate the protected procedure calls implemented in hardware on, for example, Multics [28] and some capability systems such as the CAP [9]. An example is the replacement of the original TAOS RPC mechanism by Lightweight RPC [10].

In these cases, the performance advantage of thread tunnelling comes at a price; since the thread has left the client domain, it has the same effect as having blocked as far as the client is concerned. All threads must now be scheduled by the kernel (since they cross protection domain boundaries), thus applications can no longer reliably internally multiplex the CPU. Accounting information must be tied to kernel threads, leading to the crosstalk discussed in section III.

B.1 Standard mechanism

The 'baseline' IDC invocation transport mechanism in Nemesis operates very much like a conventional RPC mechanism. The bind process creates a pair of event channels between client and server. Each side allocates a shared memory buffer and ensures that it is mapped read-only into the other domain. The server creates a thread which waits on the incoming event channel.

An invocation copies the arguments (and the operation to be invoked) into the client's buffer and sends an event on its outgoing channel, before waiting on the incoming event channel. The server thread wakes up, unmarshals the arguments and calls the concrete interface. Results are marshalled back into the buffer, or any exception raised by the server is caught and marshalled. The server then sends an event on its outgoing channel, causing the client thread to wake up. The client unmarshals the results and re-raises any exceptions.

Stubs for this transport are entirely generated by the MIDDL compiler, and the system is used for cases where performance is not critical. Measurements have been taken of null RPC times between two domains an otherwise unloaded DEC3000/400 Sandpiper. Most calls take about 30μs, which compares very favourably with those reported in [29] for Mach (88μs) and Opal (122μs) on the same hardware. Some calls (20% in the experiments) take between 55μs and 65μs; these have experienced more than one reschedule between event transmissions. Nemesis does not currently implement full memory protection for its domains; the cost of a full protection domain switch consists of a single instruction to flush the 21064 data translation buffer (DTB), followed by a few DTB misses. This cost of a DTB fill on the current hardware has been estimated at less than 1μs.

C. Rbufs

The inter-process communication mechanism described above fits the needs of inter-domain invocation quite well, but is not appropriate for stream based bulk transfer of data. Besides **Pipes** and **Streams**, schemes for controlling such transfers are more often integrated with network buffering and include **Mbufs** [30], **IOBufs** [18], **Fbufs** [31] and other schemes to support application data unit (ADU) transfer such as the IP trailers [32] scheme found in some versions of BSD. A full discussion of these schemes can be found in [20].

The scheme presented here, **RBufs**, is intended as the principal mechanism for both interdomain streams and for streams between devices and application domains. The main design considerations are based on the requirements for networking and it is in that context it is presented; however, as is demonstrated with the fileserver example, it is also intended for more general stream I/O use.

The requirements for an I/O buffering system in Nemesis are slightly different from all of the above systems. In Nemesis, applications can negotiate for resources which possess availability guarantees. This means that an application can have a certain amount of buffer memory which will not be paged. If the system is short of memory then the QoS Manager will require the application to free a certain amount. Hence, like the Fbuf system, there is no need for highly dynamic reallocation of buffers between different I/O data paths. Also it would be preferable if multi-recipient data need not be copied.

C.1 Device Hardware Considerations

It is useful to distinguish between network interfaces which are *self-selecting* and those which are not. *Self-selecting* interfaces use the VCI (or similar) in the header of arriving data to access the correct buffer control information. These are typically high bandwidth interfaces with DMA support. *Non self-selecting* interfaces require software copying of data (e.g. Ethernet).

Examples of self-selecting interfaces include the Aurora TURBOchannel interface [33] and the Jetstream / Afterburner combination [34]. In Jetstream the arriving packets enter a special buffer memory based on the arriving VCI. The device driver then reads the headers and instructs a special DMA engine to copy the data to the final location. Knowledgeable applications may make special use of the buffer pools in the special memory.

It has been recent practice in operating systems to support a protocol independent scheme for determining the process for which packets arriving at an interface are destined. This is known as packet filtering [6] and this technology is now highly advanced [7],[35]. For non-self-selecting interfaces, packet filtering can determine which I/O path the data will travel along as easily as it can determine which process will be the receiver. This property is assumed in the Nemesis buffer mechanism derived below.

On older hardware many devices which used DMA required a single non-interrupted access to a contiguous buffer. On more recent platforms such as the TURBOchannel [36] the bus architecture requires that a device burst for some maximum period before relinquishing the bus. This is to prevent the cache and write buffer being starved of memory bandwidth and halting the CPU. Devices are expected to have enough internal buffering to weather such gaps. Also, the high bandwidth that is available from DMA on typical workstations depends on accessing the DRAMs using page mode. Such accesses mean that the DMA cycle must be re-initiated on crossing a DRAM page boundary. Furthermore most workstations are designed for running Unix with its non-contiguous Mbuf chains. The result of this is that most high performance DMA hardware is capable of (at least limited) scatter-gather capability.

C.2 Protocol Software Considerations

Most commonly used protocols wish to operate on a data stream in three ways. These are:

1. To add a header (e.g. Ethernet, IP, TCP, UDP, XTP)
2. To add a trailer (e.g. XTP, AAL5)
3. To break up a request into smaller sizes.

Headers

Headers are usually used to ensure that data gets to the right place, or to signify that it came from a particular place. We can consider how such operations affect high performance stream I/O, particularly in respect of security. In the Internet much of the security which exists relies on secure port numbers. These are port numbers which are only available to the highest authority on a given machine, and receivers may assume that any such packet bears the full authority of the administrators of the source machine rather than an arbitrary user. It is similarly important that machines accurately report their own addresses. For this reason the transmission of arbitrary packets must be prohibited; transmission must include the correct headers as authorised by the system. This has been one reason for having such networking protocols in the kernel or, in a micro-kernel, implemented in a single "networking daemon". However this is not a foregone conclusion.

It is possible instead to have protocol implementations within the user process, and still retain the required security. The device driver must then perform the security control. There is a broad spectrum of the possible ways of engineering such a solution. In one extreme the device drivers actually include code (via a trusted library) which "understands" the protocol and checks the headers; which is close to implementing the protocol itself in each device driver.

Alternatively, the device driver could include an "inverse" packet filter, code which determines if the packet is valid for transmission (rather than reception). As with a packet filter for reception this process can be highly optimised.

For any implementation the volatility of the buffer memory must be taken into consideration; the driver must protect against the process corrupting the headers after they have been checked. This may entail copying the security related fields of the header before checking them. Another solution may rely on caching the secure part of the header in the device driver's private memory and updating the per-packet fields.

For many other fields such as checksums, the user process is the only one to suffer if they are not initialised correctly. Thus for UDP and TCP only the port values need to be secured. For IP all but the *length* and *checksum* fields must be secured, and for Ethernet all the fields must be secured.

One final possible concern would be with respect to flow control or congestion avoidance; conceivably a user process could have private code which disobeyed the standards on TCP congestion control. There are various answers to this. First, a malevolent user process could simply use UDP, which has no congestion control, instead if it wished. Second, since the operating system is designed with quality of service support, the system could easily limit the rate at which a process is permitted to transmit. Third, the application may in fact be able to make better use of the resources in the network due to application specific knowledge, or by using advanced experimental code.

Trailers

Unlike headers, trailers do not usually contain any security information. Trailers are most easily dealt with by requiring the user process to provide enough space (or the correct padding) for the packet on both receive and transmit. If there is not enough, the packet will simply be discarded - a loss to the user processes. Providing this space is not difficult for a process once it is known how much is necessary; this value can be computed by a shared library or discovered using an IPC call.

Application data units

Many applications have application specific basic data units which may be too large for individual network packets. For example, NFS blocks over Ethernet are usually fragmented at the IP level. Ideally a system should permit the application to specify receive buffers in such a way that the actual data of interest to the application ends up in contiguous virtual addresses.

On the other hand for some applications, the application's basic data unit (i.e. the unit over which the application considers loss of any sub part to be loss of the total) may be very small. This may be found in multimedia streams such as audio over ATM, and compressed tiled video. For such streams, the application should not have to suffer very large numbers of interactions with the device driver; it should be able to handle the data stream only when an aggregate of many small data units is available.

C.3 Operation

The Rbuf design separates the three issues of I/O buffering, namely:

- The actual data.
- The offset/length aggregation mechanisms.
- The memory allocation and freeing concerns.

An I/O channel is comprised of a data area (for the actual data) and some control areas (for the aggregation information). The memory allocation is managed independently of the I/O channel by the owner of the memory.

Data Area

The Rbuf Data Area consists of a small number of large contiguous regions of the virtual address space. These areas are allocated by the system and are always backed by physical memory. Revocation of this memory is subject to out of band discussion with the memory system. To as large an extent as possible the memory allocator will keep these contiguous regions of virtual addresses backed by contiguous regions of physical addresses (this is clearly a platform dependent factor).

The system provides a fast mechanism for converting Rbuf Data Area virtual addresses into physical addresses for use by drivers that perform DMA. On many platforms a page table mapping indexed by virtual page number exists for use by the TLB miss handler; on such platforms this table can be made accessible to device driver domain with read only status.

Protection of the data area is determined by the use of the I/O channel. It must be at least writable in the domain generating the data and at least readable in the domain receiving the data. Other domains may also have access to the data area especially when an I/O channel spanning multiple domains (see section VI-C.4) is in use.

One of the domains is logically the owner in the sense that it allocates the addresses within the data area which are to be used.

The Rbuf data area is considered volatile and is always updateable by the domain generating the data.

Data Aggregation

A collection of regions in the data area may be grouped together (e.g. to form a packet) using a data structure known as an I/O Record or **iorec**. An iorec is closest in form to the Unix concept of an **iovec**. It consists of a header followed by a sequence of base and length pairs. The header indicates the number of such pairs which follow it and is padded to make it the same size as a pair.

This padding could be used on some channels where appropriate to carry additional information. For example the exact time at which the packet arrived or partial checksum information if this is computed by the hardware. [37] points out that for synchronisation it is more important to know exactly when something happened than getting to process it immediately.

Control Areas

A control area is a circular buffer used in a producer / consumer arrangement. A pair of event channels is provided between the domains to control access to this circular buffer. One of these event channels (going from writer to reader) indicates the *head* position and the other (going from reader to writer) indicates the *tail*.

A circular buffer is given memory protection so that it is writable by the writing domain and read-only to the reading domain. A control area is used to transfer **iorec** information in a simplex direction in an I/O channel. Two of these control areas are thus required to form an I/O channel and their sizes are chosen at the time that the I/O channel is established.

Figure 6 shows a control area with two **iorec**s in it. The first **iorec** describes two regions within the Rbuf data area whereas the second describes a single contiguous region.

Usage

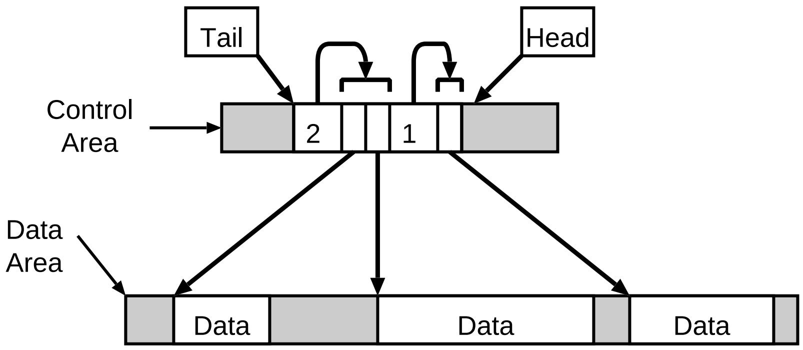

Fig. 6
Rbuf memory arrangement

Figure 7 shows two domains A and B using control areas to send **iorecs** between them. Each control area, as described above, provides a fifo queue of **iorec**s between the two ends of an I/O channel. Equivalently, an I/O channel is composed of two simplex control area fifos to form a duplex management channel. The control areas are used indistinguishably no matter how the I/O channel is being used.

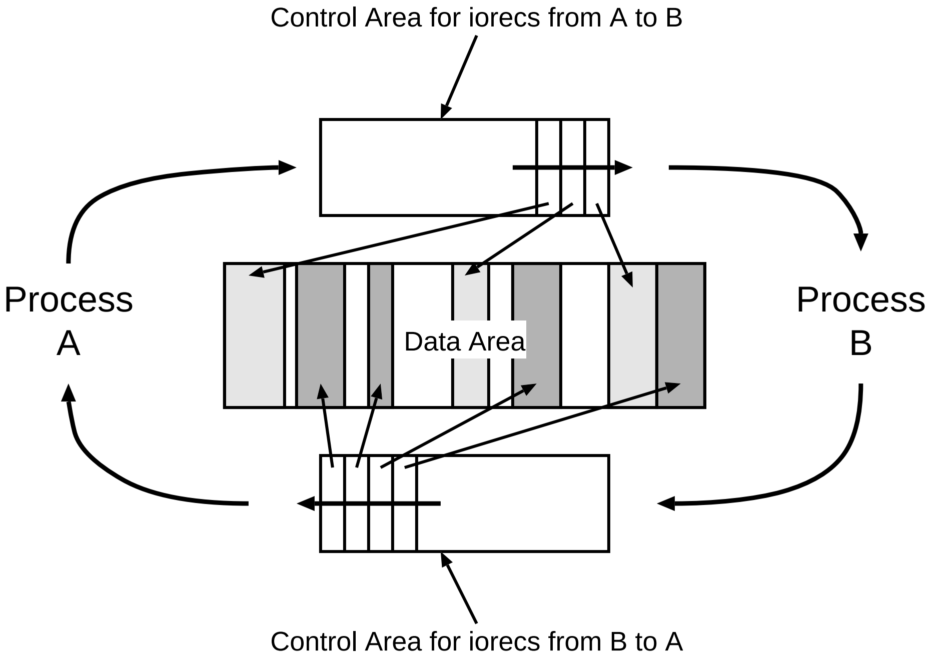

Fig. 7
Control Areas for an I/O channel between A and B

A typical I/O channel is in fact a simplex data channel operating in one of two modes. The purpose of these two modes is to allow for the support of ADUs in various contexts. Note that there is no requirement for either end of the I/O channel to process the data in a FIFO manner, that is merely how the buffering between the two ends is implemented.

In Transmit Master Mode (TMM), the originator of the data chooses the addresses in the Rbuf data area, places the data into the Rbufs, and places the records into the control area. It then updates the *head* event for that control buffer indicating to the receiver that there is at least one record present. As soon as the downstream side has read these records from the control buffer it updates the other (*tail*) event, freeing the control buffer space for more records. When the downstream side is finished with the data it places the records into the control area for the queue in the other direction and signals its *head* event on that control buffer. The originator likewise signals when it has read the returned acknowledgement from the control buffer. The originator is then free to reuse the data indicated in the returning control buffer.

In Receive Master Mode (RMM), the operation of the control areas is indistinguishable from TMM; the difference is that the Rbuf data area is mapped with the permissions reversed and the data is placed in the allocated areas by the downstream side. It is the receiver of the data which chooses the addresses in the Rbuf data area and passes **iorec**s which indicate where it wishes the data to be placed to the downstream side. The downstream side uses the other control area to indicate when it has filled these areas with data.

The Master end, which is choosing the addresses, is responsible for managing the data area and keeping track of which parts of it are "free" and which are "busy". This can be done in whatever way is deemed appropriate. For some applications, where FIFO processing occurs at both ends, it may be sufficient to partition the data area into **iorecs** at the initiation of an I/O channel, performing no subsequent allocation management.

Table I presents a summary of the differences between TMM and RMM for the diagram shown in figure 7; without loss of generality **A** is the master - it chooses the addresses within the data area.

TABLE I

TMM AND RMM PROPERTIES

	TMM	RMM
Chooses the Addresses	A	A
Manages data area	A	A
Write access to data	A	B
Read access to data	B	A

Since the event counts for both control areas are available to a user of an I/O channel it is possible to operate in a non-blocking manner. By reading the event counts associated with the circular buffers, instead of blocking on them, a domain can ensure both that there is an Rbuf ready for collection and also that there will be space to dispose of it in the other buffer. This functions reliably because event counts never lose events. Routines for both blocking and non-blocking access are standard parts of the Rbuf library.

C.4 Longer channels

Sometimes an I/O channel is needed which spans more than two domains. An example may be a file serving application where data arrives from a network device driver, passes to the fileserver process, and then passes to the disk driver.

When such an I/O channel is set up it is possible to share certain areas of Rbuf data memory which are already allocated to that domain for another I/O channel. A domain may wish to have some private Rbufs for each direction of the connection (i.e. ones which are not accessible to domains in the other direction) for passing privileged information. In the fileserver example, the fileserver may have Rbufs which are used for **inode** information which are not accessible by the network device driver.

The management of the channel may either be at one end or it may be in the middle. In the example of the fileserver, it is likely to be in TMM for communicating with the disk driver, and RMM for communicating with the network driver. The important point is that the data need not be copied in a longer chain provided trust holds.

Figure 8 shows the I/O channels for a fileserver. For simplicity, this only shows the control paths for writes. The **iorec**s used in the channel between the fileserver and the disk driver will contain references to both the network buffer data area and the private inode data area. Only the network data buffer area is used for receiving packets. The fileserver (operating in RMM) will endeavour to arrange the **iorec**s so that the disk blocks arriving (probably fragmented across multiple packets) will end up contiguous in the single address space and hence in a suitable manner for writing to disk.

C.5 Complex channels

In some cases the flow of data may not be along a simple I/O channel. This is the case for multicast traffic which is being received by multiple domains on the same machine. For such cases the Rbuf memory is mapped readable by all the recipients using TMM I/O channels to each recipient. The device driver places the records in the control areas of all the domains which should receive the packet and reference counts the Rbuf areas so that the memory is not reused until all of the receivers have indicated they are finished with it via their control areas.

Apart from the lack of copying, both domains benefit from the buffering memory provided by the other compared with a scheme using copying.

A problem potentially arises if one of the receivers of such multicast data is slower at processing it than the other and falls behind. Ideally it would not be able to have an adverse affect on the other receiver. This can be done by limiting the amount of memory in use by each I/O channel. When the limit is reached, the **iorec**s are not placed in that channel and the reference count used is one less. The buffers are hence selectively dropped from channels where the receiver is unable to keep up. An appropriate margin may be configured based on the fan-out of the connection.

One approximate but very efficient way of implementing this margin is to limit the size of the circular control buffer. **Iorec**s are then dropped automatically when they cannot be inserted in the buffer in a non-blocking manner. Even if

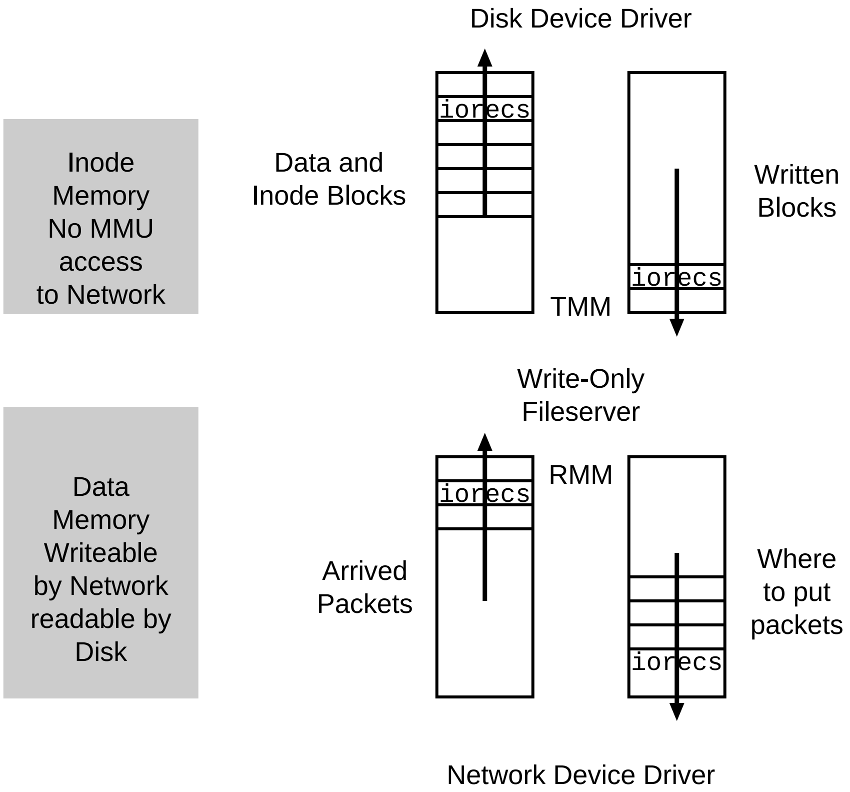

Fig. 8
A longer Rbuf channel: Control path for Fileserver Writes

a more accurate implementation of the margin is required, the Rbuf scheme ensures that the cost is only paid for I/O channels where it is required, rather than in general.

TABLE II
Comparison of Buffering Properties

	Mbufs	IOBufs	Fbufs	Rbufs
page faults possible	No	No	Yes	No
alignment OK	No	Yes	??	Yes
copy to user process	Yes	No	No	No
copy to clever device	Yes	No	No	No
copy for multicast	Yes	Yes	Yes?	No
copy for retransmission	Yes	Yes	No	No
support for ADUs	No	No	No	Yes
limit on resource use	Yes[4]	No	No	Yes
must be cleared	No[5]	Yes	No[6]	No[6]

VII. Conclusions

A. Current State

Nemesis is implemented on Alpha AXP, MIPS and ARM platforms. C libraries have been ported to these platforms to allow programming in a familiar environment; this has required, amongst other things the integration of pervasives

[4]This limit is actually as a result of socket buffering.
[5]This is because of the copy to user process memory. However some networking code rounds up the sizes of certain buffers without clearing the padding bytes thus included; this can cause an information leak of up to three bytes.
[6]Buffers must be cleared when the memory is first allocated. This allocation is not for every buffer usage in Fbufs but is still more frequent in Fbufs than in Rbufs.

rather than statics within the library code. Schedulers as described have been developed; the Jubilee scheduler so far has only been implemented on the ARM platform[7].

A window system which provides primitives similar to X has been implemented. For experimentation this has been implemented both as a traditional shared server to provide server-based rendering and, in the more natural Nemesis fashion, as a library to provide client-based rendering. Experiments have shown that the client based rendering system reduces application crosstalk enormously when best effort applications are competing with continuous media applications rendering video on the display. This work will be reported in detail in the near future.

The ATM device driver for the DEC ATMWorks 750 TURBOchannel takes full advantage of the self-selecting feature of the interface to direct AAL5 adaptation units directly into memory at the position desired by the receiving domain; rendering video from the network requires a single copy from main memory into the frame buffer. (The application must check the frames to discover where they should be placed in the frame buffer.) The combination of the ATMWorks interface and the device driver mean that contention between outgoing ATM streams occurs only as a result of scheduling the processor and when when cells hit the network wire.

B. Future Plans

Nemesis is immature with much to work still to be done. It represents a fundamentally different way of structuring an operating system; indeed it could be regarded as a family of operating systems. The core concepts of events, domains, activations, binding, rbufs and minimal kernel do not define an operating system. Our next task is to create the libraries, device drivers and system domains to provide a complete operating system over the Nemesis substrate.

As with the window system (and indeed the filing system and the IP protocol stack) this will often be by providing an initial server-based implementation by porting code from other platforms and then moving towards a client-based execution implementation. Work with the filing system and protocol stack has just entered this second stage.

As these components become stable we expect to develop (and port) applications to take advantage of the facilities, in particular the flexible quality of service guarantees, available from the Nemesis architecture. We also will be using Nemesis as a means on instrumenting multimedia applications; we can trace resource usage directly to application domains and thereby get an accurate picture of application performance.

C. Conclusions

Nemesis represents an attempt to design an operating system to support multimedia applications which process continuous media. The consideration of quality of service provision and application crosstalk led to a design in which applications execute their code directly rather than via shared servers. Shared servers are used only for security or concurrency control.

Such an organisation gives rise to a number of problems with complexity which are solved by the use of typing, transparency and modularity in the definition of interfaces and the use of closures to provide comprehensible, safe and extensive sharing of code and data. Application programmers can be protected from this paradigm shift; API's need not change except when new facilities are required, the porting of the C library is a case in point.

The development of a new operating system is not a small task. The work here has been developed over at least four years with the help of a large number of people. We are indebted to all who worked in the Pegasus project.

References

[1] S. J. Mullender, I. M. Leslie, and D. R. McAuley, "Operating-system support for distributed multimedia", in *Proceedings of Summer 1994 Usenix Conference*, Boston, Massachusetts, USA, June 1994, pp. 209–220, Also available as Pegasus Paper 94–6.

[2] C. J. Lindblad, D. J. Wetherall, and D. L. Tennenhouse, "The VuSystem: A programming system for visual processing of digital video", in *Proceedings of ACM Multimedia*, San Francisco, CA, USA, Oct. 1994.

[3] G. Coulson, A.Campbell, P. Robin, G. Blair, M. Papathomas, and D. Sheperd, "The design of a QoS-controlled ATM-based communication system in chorus", *IEEE Journal on Selected Areas In Communications*, vol. 13, no. 4, pp. 686–699, May 1995.

[4] D.R. McAuley, "Protocol Design for High Speed Networks", Tech. Rep. 186, University of Cambridge Computer Laboratory, January 1990, Ph.D. Dissertation.

[5] D. L. Tennenhouse, "Layered multiplexing considered harmful", in *Protocols for High Speed Networks*, Rudin and Williamson, Eds. 1989, Elsevier.

[6] J. Mogul, "Efficient Use of Workstations for Passive Monitoring of Local Area Networks", in *Computer Communication Review*. ACM SIGCOMM, September 1990, vol. 20.

[7] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", in *USENIX Winter 1993 Conference*, January 1993, pp. 259–269.

[8] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of Fair Queueing Algorithm", *Journal of Internetworking: Research and Experience*, vol. 1, no. 1, 1990.

[9] M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, North Holland, 1979.

[10] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "Lightweight Remote Procedure Call", *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 37–55, February 1990.

[11] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda, "Processor capacity reserves: Operating system support for multimedia applications", in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[7] This is used in an embedded application where the number of processes is small.

[12] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska, "Implementing network protocols at user level", Tech. Rep. 93-03-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1993.

[13] P. Barham, M. Hayter, D. McAuley, and I. Pratt, "Devices on the Desk Area Network", *IEEE Journal on Selected Areas In Communications*, vol. 13, no. 4, May 1995.

[14] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System", Tech. Rep. 92-3, Sun Microsystems Laboratories, Inc., December 1992.

[15] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53–79, February 1992.

[16] D. Reed and R. Kanodia, "Synchronization with eventcounts and sequencers", Tech. Rep., MIT Laboratory for Computer Science, 1977.

[17] D. Mills, "Internet Time Synchronisation: The Network Time Protocol", Internet Request for Comment Number 1129, October 1989.

[18] M.J. Dixon, "System Support for Multi-Service Traffic", Tech. Rep. 245, University of Cambridge Computer Laboratory, September 1991, Ph.D. Dissertation.

[19] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[20] R.J. Black, "Explicit Network Scheduling", Tech. Rep. 361, University of Cambridge Computer Laboratory, December 1994, Ph.D. Dissertation.

[21] A.D. Birrell and J.V. Guttag, "Synchronization Primitives for a Multiprocessor: A formal specification", Tech. Rep. 20, Digital Equipment Corporation Systems Research Center, 1987.

[22] Timothy Roscoe, *The Structure of a Multi-Service Operating System*, PhD thesis, University of Cambridge Computer Laboratory, April 1995, Available as Technical Report no. 376.

[23] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

[24] Dave Otway, "The ANSA Binding Model", ANSA Phase III document APM.1314.01, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, October 1994.

[25] Andrew Birrell, Greg Nelson, Susan Owicki, and Ted Wobber, "Network Objects", *Proceedings of the 14th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review*, vol. 27, no. 5, pp. 217–230, Dec. 1993.

[26] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Draft 10th December 1991, OMG Document Number 91.12.1, revision 1.1.

[27] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors", *ACM Transactions on Computer Systems*, vol. 9, no. 2, pp. 175–198, May 1991.

[28] E.I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

[29] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska, "Sharing and Protection in a Single Address Space Operating System", Technical Report 93-04-02, revised January 1994, Department of Computer Science and Engineering, University of Washington, Apr. 1993.

[30] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

[31] P. Druschel and L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", in *Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 189–202.

[32] S. Leffler and M. Karels, "Trailer Encapsulations", Internet Request for Comment Number 893, April 1984.

[33] P. Druschel, L. Peterson, and B. Davie, "Experiences with a High-Speed Network Adaptor: A Software Perspective", in *Computer Communication Review*. ACM SIGCOMM, September 1994, vol. 24, pp. 2–13.

[34] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN", in *Computer Communication Review*. ACM SIGCOMM, September 1994, vol. 24, pp. 14–23.

[35] M. Yuhara, C. Maeda, B. Bershad, and J. Moss, "The MACH Packet Filter: Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages", in *USENIX Winter 1994 Conference*, January 1994, pp. 153–165.

[36] Digital Equipment Corporation TURBOchannel Industry Group, *TURBOchannel Specifications Version 3.0*, 1993.

[37] C.J. Sreenan, "Synchronisation services for digital continuous media", Tech. Rep. 292, University of Cambridge Computer Laboratory, March 1993, Ph.D. Dissertation.

Ian Leslie received the B.A.Sc in 1977 and M.A.Sc in 1979 both from the University of Toronto and a Ph.D. from the University of Cambridge Computer Laboratory in 1983. Since then he has been a University Lecturer at the Cambridge Computer Laboratory. His current research interests are in ATM network performance, network control, and operating systems which support guarantees to applications.

Derek McAuley obtained his B.A. in Mathematics from the University of Cambridge in 1982 and his Ph.D. on ATM internetworking addressing issues in interconnecting hetergeneous ATM networks in 1989. After a further 5 years at the Computer Laboratory in Cambridge as a Lecturer he moved to a chair in the Department of Computing Science, University of Glasgow in 1995. His research interests include networking, distributed systems and operating systems. Recent work has concentrated on the support of time dependent mixed media types in both networks and operating systems. This has included development of ATM switches and devices, leading to the DAN architecture, and the development of operating systems that can exploit these systems components.

Richard Black obtained a Bachelor's degree in Computer Science in 1990, and a Ph.D. in 1995, both from the University of Cambridge. He has been awarded a Research Fellowship from Churchill College, to continue research at the University of Cambridge Computer Laboratory. His research interests are on the interaction between operating systems and networking. Previous activities have included work on The Desk Area Network and the Fairisle ATM Switch.

Timothy Roscoe received the BA degree in Mathematics from the University of Cambridge in 1989, and the PhD degree from the University of Cambridge Computer Laboratory in 1995, where he was a principal architect of the Nemesis operating system and author of the initial Alpha/AXP implementation. He is currently Vice President of Research and Development for Persimmon IT, Inc. in Durham, North Carolina. His research interests include operating systems, programming language design, distributed naming and binding, network protocol implementation, the future of the World Wide Web, and the Chapel Hill music scene.

Paul Barham received the B.A. degree in computer science from the University of Cambridge, Cambridge, UK., in 1992 where he is now working towards the Ph.D. degree. He is currently involved in the Pegasus, Measure and DCAN projects at the Computer Laboratory, investigating Quality of Service provision in the operating system and particularly the I/O subsystem of multimedia workstations. His research interests include operating systems and workstation architecture, networking, and a distributed parallel Prolog. Recent work includes the PALcode, kernel and device drivers for Nemesis on the DEC 3000 AXP platforms, a client-rendering window system and an extent-based filesystem both supporting QoS guarantees.

David Evers received the B.A. degree in Physics and Theoretical Physics and the Ph.D. degree in Computer Science from the University of Cambridge, Cambridge, UK., in 1988 and 1993 respectively. He is currently a member of staff at Nemesys Research Ltd. in Cambridge, where he continues to work on software support, from devices to applications, for distributed multimedia in an ATM environment.

Robin Fairbairns received the B.A. degree in mathematics in 1967, and the diploma in computer science in 1968 from the University of Cambridge, UK. He worked on the CAP project at the University of Cambridge Computer Laboratory from 1969 to 1975, and has worked on digital cartography and satellite remote sensing. His current work has been in the Pegasus and Measure projects, and he is working towards a Ph.D. investigating the provision of Quality of Service within Operating Systems.

Eoin Hyden received the B.Sc., B.E. and M.Eng.Sc. degrees for the University of Queensland, Australia, and the Ph.D. from the University of Cambridge Computer Laboratory, UK. Currently he is a Member of Technical Staff in the Networked Computing Research department at AT&T Bell Laboratories, Murray Hill. His interests include operating systems, high speed networks and multimedia systems.