# Protocol Implementation in a Vertically Structured Operating System

Richard Black Paul Barham Austin Donnelly Neil Stratford
University of Cambridge
Computer Laboratory
Cambridge, CB2 3QG, U.K.
E-mail: *First.Last@cl.cam.ac.uk*, e.g. *Paul.Barham@cl.cam.ac.uk*
Tel: +44 1223 334600

## Abstract

*A vertically structured Operating System is one in which neither the "kernel" nor "servers" perform work on behalf of applications – the former because it exists only to multiplex the CPU, and the latter in order to avoid Quality of Service interference between the applications. Instead, wherever possible, the applications perform all of their own processing. Such a vertical structure provides many advantages for applications but leads to some interesting problems and opportunities for protocol stack implementation. This paper describes the techniques we used in our protocol implementation and the benefits that the vertical structure provided.*

## 1. Introduction

A *vertically structured* operating system is one in which applications perform as much of their own processing as possible rather than having a kernel or shared servers do this processing on their behalf. In order to avoid duplication of code, applications can call on a collection of shared code modules to implement for themselves the system services traditionally carried out by the kernel or shared servers.

Two examples are the Exokernel [9] and *Nemesis* [10], however the motivation for these two systems is different. In the case of the Exokernel, the principal motivation was to permit applications to optimize the implementations of various system services using application-specific knowledge, and thereby improve performance. In the case of *Nemesis*, the motivating factor was the desire to simplify resource accounting as an essential step towards the provision of Quality of Service (QoS). Performance gains were merely a pleasant side-effect.

The accounting required to support communications QoS is most easily performed in the presence of explicit *connections*. The initial communications support within *Nemesis* was entirely based on ATM, where the connection oriented nature made associations between VCIs and applications straightforward. The datagram-based nature of IP presents an interesting challenge in this respect.

This work has taken *Nemesis* and added to it support for standard Internet protocols. It has some clear similarities to previous work on user-space protocol implementation [3, 8, 15], however there are many novel aspects. Frequently in other work, a user space implementation is an enhancement designed to improve performance and/or reduce latency. The kernel protocol implementation remains and is used for handling the unusual cases – timeouts, retransmissions, data transfer when the application is swapped-out, connection establishment and the like. In some cases, wire-compatibility has been sacrificed, and in others the application must be trusted not to forge addresses in transmitted packets.

In our work, the intent is to provide *Nemesis*-style QoS guarantees (see section 2) to IP "flows" and to provide the secure environment required when no backup kernel protocols are available. Although the design of the network stack will necessarily be focused on IP, we believe it to be flexible enough to cope with other networking technologies.

In section 2 we describe the relevant background of *Nemesis* and related work. Section 2.2 describes the buffering and packet I/O mechanisms in *Nemesis*, section 3 describes the protocol stack design and section 4 its current implementation. Finally sections 5 and 6 present the performance and conclusions.

## 2. Background

The environment being considered here is that of a general purpose workstation being used to run a variety of applications, some of which will be processing continuous media. The application mix and load will be dynamic and even individual applications may change their demands as they proceed through "phase changes".

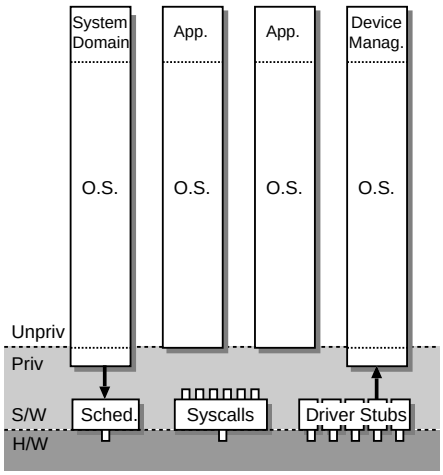


**Figure 1.** *Nemesis* **system architecture**

*Nemesis* is a multimedia operating system developed from scratch at the Cambridge University Computer Laboratory with the aim of supporting applications with soft real-time constraints. This means QoS-directed scheduling of *all* resources within a system, and the suitable allocation of resources to applications on a longer time-scale. Although processor scheduling is an important aspect of providing quality of service to applications, it is not the complete solution. Guaranteed access to the network is often more important for today's network-centric applications.

Applications can interfere with one another in many ways. Contention for *real* resources such as network capacity, physical memory, cache, and frame buffer will occur, but so can contention for any *virtual* resources which an operating system introduces over these. Where such resources are shared, crosstalk between competing applications can arise. This is called QoS-crosstalk.

The *Nemesis* philosophy is to to multiplex each real resource precisely once. Application *domains*,[1] rather than the kernel or a shared server, are responsible for abstracting these real resources to provide the standard high level operating system services. Familiar APIs are provided by shared libraries rather than by a veneer of library code over a set of kernel traps and/or messages to server processes.

## 2.1. Structure

Accounting resource usage accurately while retaining secure control over shared state has been considered in detail and a number of general principles developed.

*Nemesis* takes the approach of minimising the use of shared servers to reduce application QoS crosstalk: only the smallest necessary part of a service is placed in a shared server while as much processing as possible is performed within the application domain (often using shared library code). The server should *only* perform privileged operations, in particular access control and concurrency control.

A frequent consequence is the need to expose some server internal state in a controlled manner to client domains. In *Nemesis* this is easily achieved using shared memory, in conjunction with interfaces and modules [14]. *Nemesis* uses a single virtual address space, however this in no way implies a lack of memory *protection* between domains, since the protection rights on any given page may vary. The single virtual address space greatly reduces the memory-system related penalties of the higher context-switch rate required to support time-sensitive applications, and is clearly advantageous for applications processing high-bandwidth multimedia data.

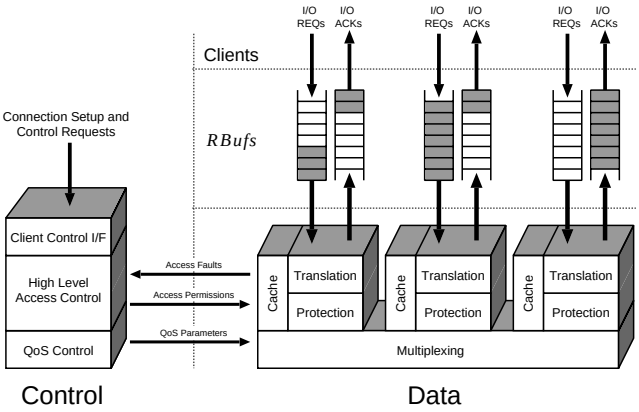


**Figure 2.** *Nemesis* **device driver architecture**

The result is the operating system architecture illustrated in figure 1: *Nemesis* is organized as a set of domains, which are scheduled by a tiny kernel. The *Nemesis* kernel consists almost entirely of the scheduler, and interrupt and trap handlers; there are *no* kernel threads (one kernel stack per physical CPU). Kernel traps are provided to send events, yield the processor and control activations [1]. Trusted domains can also register interrupt handlers, mask interrupts and if necessary (on some processors) ask to be placed in a particular processor privileged mode (e.g. to access the TLB etc).

Although *Nemesis* has a small kernel we avoid the term "micro-kernel" due to its modern connotations of server structure.[2] Instead we prefer the term *"vertically integrated"*.

As previously mentioned, QoS guarantees for the CPU resource alone are insufficient for multimedia applications which by their very definition are frequently I/O intensive.

[1] The closest Nemesis analogue of a UNIX *process*.

[2] At about 4000 lines of code including initialisation and debugging support it's a good deal smaller than most micro-kernels too.

Device I/O under *Nemesis* is very different from contemporary operating systems (figure 2). Once again a rigid separation of control- and data-path operations is enforced. The data-path portion of a device driver does the minimum necessary to securely multiplex the *real* hardware resources according to pre-negotiated QoS guarantees. Such servers are coded with care to avert as much QoS crosstalk as possible; we address this problem for network device drivers in this paper.

In an ideal world *all* devices would provide an interface where they could be accessed on the data path without use of a device driver. Such devices have been termed "User Safe Devices" [13]. The U-Net network interface [3] provides secure low-latency communication via direct application access, but would require additional quality of service functions in the *Nemesis* environment.

Space here prevents us from describing the overall structure of *Nemesis* in greater detail, however the core of the system and its general state in May '95 were described in [10]. We now proceed to describe the work we have done in implementing traditional protocol families on that base.

## 2.2. Communications support

The bulk (asynchronous) data transport mechanism in *Nemesis* is known as an Rbuf channel [5]. The design is based on a separation of the three logical issues in I/O:

- The actual data buffering memory.
- The aggregation mechanisms (for Application Data Unit (ADU) support).
- The memory allocation.

To preserve QoS the I/O channels are designed to be completely independent. No resources are shared between them.

**2.2.1. Data area.** The Rbuf Data Area consists of a small number (e.g. 1) of contiguous regions of the virtual address space. These areas are always backed by physical memory (from the *real* resources allocated by the memory system) and a fast mechanism is provided for converting virtual addresses into physical addresses for use by drivers which perform DMA.

Access rights for the data area are determined by the direction 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. Together these arrangements guarantee to a device driver that the memory area is always accessible without page-faulting. The data area is considered volatile and is always updateable by the domain generating the data.

**2.2.2. Control areas.** 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 similar to the Unix structure called an iovec, except that as well as a sequence of base and length pairs, an iorec includes a header indicating the number of such pairs which follow it (the header is padded to make it the same size as a pair).

The domains pass iorecs between them to indicate which addresses are being used for the "transfer". The iorecs are passed using two circular buffers known as *control areas* each of which is managed in a producer / consumer arrangement (and mapped with appropriate memory permissions). A pair of event count channels[3] is provided between the domains to mediate access to each circular buffer. Thus each simplex I/O channel has two control areas and four event channels (figure 3).

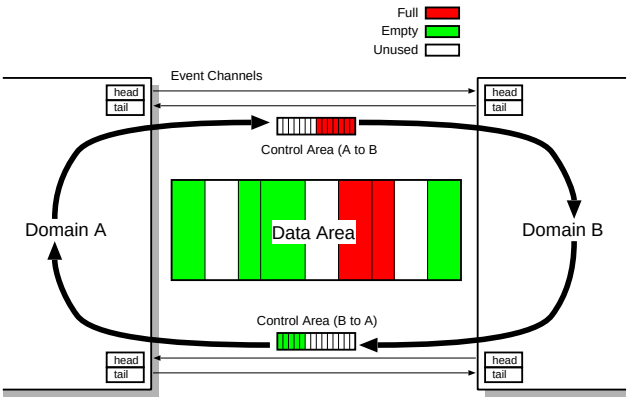


**Figure 3. Control areas for an I/O channel**

Each control area functions as a FIFO buffer for the meta information about the communication between the two domains. 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. The sizes of the control areas are fixed per-channel; this provides for a limit on the number of packets outstanding and an effective form of back-pressure preventing livelock.

Since the event counts for both control areas are available to the user of an Rbuf 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 a packet ready for collection and also that there will be space to dispose of it in the other control area. Routines for both blocking and non-blocking access are standard parts of the Rbuf library.

[3] The basic Inter-Domain Communication (IDC) primitive in *Nemesis*.

**2.2.3. Buffer management.** An I/O channel can be operated in one of two different modes. In Transmit Master Mode (TMM), it is the originator of the data which chooses the addresses in the Rbuf data area. Whereas 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 non-master side. It is the receiver of the data which chooses the addresses in the Rbuf data area.

As well as choosing the addresses, the Manager is also responsible for keeping track of which parts of the data area are "free" and which are "busy".

**2.2.4. Complex channels.** Multi-destination communications (e.g. reception of multicast packets to multiple domains) is handled using multiple TMM where the master writes the same iorecs into more than one channel and reference counts the use of a single shared data area. If any domain is receiving too slowly (e.g. to the extent that its control area fills) then the transmitting domain will drop further packets to that one client only; other multicast clients will be unaffected.

Rbuf channels may also be extended to form chains spanning additional domains; in such cases the other domains may also have access to the data area and additional control areas and event channels are required.

## 3. Design

At a philosophical level, the inarguable design goals of Operating System networking design are to maximize the tradeoff between "performance" and "resources" without compromising "security". Exactly what this represents at a lower level tends to vary less in security considerations – that a principal may not forge its name (i.e. send packets which purport to be from another IP address, port number, or Ethernet address) or access other principal's data (i.e. be able to read data for others), than in performance considerations.

Performance goals can be various. In the oft-criticized Berkeley mbuf design, the goal was to permit communication on slow networks using as little memory as possible bearing in mind that the entire process may be swapped out. In the more recent U-Net system [3] the goal was low latency. Likewise, there is frequently a bandwidth tradeoff associated with the number of clients supported.

Our performance goal is to sustain the QoS guarantees available to applications. Fortunately this is compatible with latency and bandwidth desires.

### 3.1. Hardware considerations

In [5], it was proposed that network adapters be classified into two categories, *self-selecting* and *non-self-selecting*. Self-selecting hardware is able to demultiplex received data to its (hopefully final) destination directly; DEC's ATMWorks 750 (OTTO) board is a good example of such hardware. 3Com's popular 3c509 Ethernet board is an example of non-self-selecting hardware; it delivers all incoming data to a small on-card circular buffer, leaving the device driver or (more commonly) the protocol stack to perform the demultiplexing and *copy* the data to its eventual destination.

Some protocols have sufficiently simple (de)multiplexing that it is reasonable to make use of such support directly in the hardware. Obvious examples include basic AAL5 access over either the U-Net modified Fore Systems ATM interface, or the *Nemesis* ATMWorks [2] driver.[4] Less obvious examples are the Autonet-One Buffer Queue Index (BQI) scheme for TCP connections [15], or the Ipsilon scheme for allocating ATM circuits to IP flows [12]. In such schemes the security information referred to above is assumed to have been transmitted to the peer during flow establishment.

For other protocols (such as the Internet Protocol family), things are somewhat more tricky and can rarely be performed in hardware. For receive it is at least necessary to perform some form of packet-classify operation on the packets. It is to these protocols and network interfaces that we now turn our attention.

**3.1.1. Receive.** In our current architecture, all network adapter cards are abstracted into ones capable of self-selection. This requires drivers for non-self-selecting cards to include some packet filtering in order to demultiplex received data. This must be done before the memory addresses that the data is to be placed in can be known, and will usually require that the data be placed in some private memory belonging to the device driver where it can be examined. On hardware which presents the data as a FIFO from which the CPU requests each successive byte or word, the device driver can combine copying the header out of the FIFO with packet filtering. It can then leave the main body of the data in the FIFO, to be retrieved once its final destination is known. This avoids the need for private driver memory and (more importantly) a copy from it into the client buffers.

Once the packet filter has determined which protocol stack instance (or "flow") the packet is for the device driver must copy the packet into the Rbuf data area associated with that flow. Unfortunately both the packet-filtering operation

[4] This requires a very small amount of work in the device driver to transfer the descriptor rings to and from the interface, and to demultiplex the report ring.

and (particularly) the copy consume CPU resources which are *not* directly accounted to the client and therefore have implications for QoS provision. We return to the copy problem below in section 3.5.

**3.1.2. Transmit.** For transmission the device driver has a similar, though slightly simpler procedure to perform. The header of the outgoing packet must be checked to ensure compliance with security. This is like packet filtering except that, for almost all protocols, it can be achieved using a simple compare and mask; there is no demultiplexing on the fields and much information is pre-computable. Note that since the Rbuf data area remains concurrently writable by the client, the header of the packet must be copied to some device-driver private memory either before or as part of the checking process.

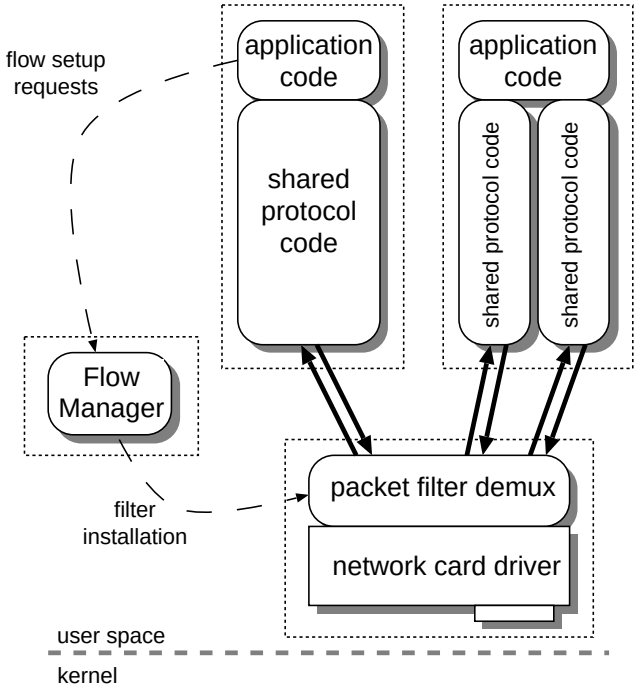


**Figure 4. Logical interaction of components in the protocol stack. Actual structure is as shown in figure 1.**

## 3.2. Flow manager

Flow setup and teardown is handled by a Flow Manager server process. Once a connection has been set up, the Flow Manager takes no further part in handling the actual data for client applications; the Flow Manager is on the control path only. All protocols (including datagram protocols such as UDP) are treated as based on a flow between a local and a remote Service Access Point (SAP).[5]

Figure 4 shows the logical interactions between four user-space domains (two applications, the Flow Manager, and a device driver), each having its own protection domain (shown as dotted rectangles). Each solid arrow represents a simplex Rbuf I/O channel; there are two for each duplex flow.

Since the Flow Manager is a system process, it is trusted to perform port allocation and update driver packet filters in a consistent fashion. For standard protocols, this means we can use built-in native code to perform the packet filtering, and for non-standard protocols, packet filter descriptions can safely be compiled down to machine code to be executed directly by the device driver. Our use of packet filtering technology also allows for easy extension in the range protocols that can be handled.

The Flow Manager ensures that it is the last resort handler for unrecognised packets; it can of course introduce discard patterns into the filter if it wishes. By this means, unsolicited packets cause the appropriate ICMP or TCP RST segment to be returned. If the packet was a TCP connection request then it performs a call-back to the application which had previously registered the SAP. Note that in principle, an application that was concerned about TCP opens could arrange that they be handled by an appropriately constructed flow, in a domain that had the required QoS guarantees. This provides a way of limiting the effects of SYN attacks [6], since only the attacked service will be degraded.

The Flow Manager is also responsible for maintaining per-host and per-interface state. This includes arbitrating access to resources such as port numbers, ARP cache maintenance, and holding the routeing tables. Changes to the routeing tables, and other out-of-band changes, are treated as changes in the real underlying resource (e.g. of interface bandwidth) and the flow must be re-organised (e.g. to the new device driver); the application will receive a call-back with the new details. Since higher level protocols such as TCP are attempting to monitor the dynamic effective bandwidth available, this explicit notification of real network change does them no dis-service.

## 3.3. Application stack

The application contains a vertically structured protocol stack for each IP flow. By this, we mean that it never interacts with stacks for other flows.[6] The Flow Manager has provided the application with a secure channel through to

[5] This is effectively required by RFC1122 (Host requirements) section 4.1.3.3, though many other networking stacks ignore this.

[6] Unless, of course, an application should choose to multiplex a number of flows onto the same Rbuf channel.

the device driver, and the application has obtained an Rbuf data area for the channel. These unshared resources are used by each protocol instance to ensure the QoS for the application.

Since the *Nemesis* scheduler is providing QoS guarantees for the CPU to the applications, the protocol code is capable of implementing any timers and retransmissions itself. The *Nemesis* virtual processor interface makes it simple to arrange for upcalls at the appropriate times.

In practice, most protocol stacks created by an application will use a standard shared library of protocol processing code, as described in section 4.2. An application is however free to implement its protocol stack as it wishes, since it cannot possibly compromise the rest of the system. The concept of Integrated Layer Processing (ILP) which may be used for efficiency reasons within the application's stack [7] is completely orthogonal to the mechanism by which the stacks are separated.

### 3.4. Driver scheduling

The device driver is responsible for policing use of the network by client applications: an application should not be able to use transmit bandwidth it has not explicitly been granted, nor should applications swamped by unprocessed received data cause data loss for other applications which *are* able to keep up.

Our device drivers perform traffic shaping by scheduling transmissions according to resource allocations set up by the Flow Manager. Receive backlogs are dealt with automatically since Rbuf channels provide hard back-pressure due to their explicit buffering. If an application does not send empty buffers to the driver frequently enough, the driver drops *its* packets. Thus applications which keep up are not unfairly penalised as a result of those which don't.

### 3.5. Moving the copy

In section 3.1.1 we described a problem for QoS in which the data copy required in the device driver for non-self-selecting interfaces causes CPU time not to be attributed to the application but to the device driver. We now describe a scheme for averting this problem.

**3.5.1. Call-Priv.** *Nemesis* has a concept called a *call-priv* (originally developed for the windowing system and described in [2]). This is best thought of as a pseudo-opcode — that its name is similar to the Alpha processor's call_pal is not accidental. A call-priv is a trap into trusted code provided by a device driver which allows some operation to take place in the privileged protection domain of the callee, but on the CPU time of the caller. The following requirements must be met:

- Interrupts (and scheduling) are disabled throughout.
- The code must have a maximum running time which is sufficiently small so as not to alter the scheduling behaviour of the system.
- The code must make no callbacks to application code.
- The code must not attempt to perform any synchronisation (e.g. block).
- The code must not access any addresses which may page-fault.

The Spin system [4] also permits downloading code into the kernel. In that case the intention is to permit any application to provide compiler signed safe code into the kernel. Whilst some of the requirements (such as modifiability and being call-back free) are the same as for call-privs, the scheduling requirements are very different – for their much larger kernel, the halting problem requires that they run the downloaded sections within schedulable kernel threads which are absent from the *Nemesis* architecture.

Note that the call-priv is very different from the thread migration model, since the execution environment during the privileged state is very restricted. The restriction that call-privs may not make callbacks implicitly prohibits nesting them.

**3.5.2. Call-Priv in communications.** When using non-self-selecting hardware, it is possible to arrange that the device driver receives all of the data into some protected private memory. The device driver can then operate the Rbuf channels to the applications' protocol stacks in TMM giving in the Rbuf control areas the *addresses* of the data within the device driver's private memory. The application can not access the data directly, but can use the call-priv mechanism to copy the data to the final required position.

This technique effectively moves the copy from the device driver into the application at the cost of the data structures which must be maintained by the device driver to permit the call-privs to check their requests. Since the performance goal is QoS rather than outright bandwidth this technique is advantageous.

## 4. Current implementation

This section describes our current (simple) implementation of the described design. Although under active development it is, nevertheless, already possible to measure the reduction in QoS crosstalk that this design provides.

### 4.1. Flow manager implementation

Applications communicate with the Flow Manager using same-machine RPC.[7] Applications wishing to use the network make a flow setup request to the Flow Manager; if the local port requested is free the request succeeds. The Flow Manager constructs a transmit filter, extends the receive packet filter, and installs them within the device driver domain. The Flow Manager completes the binding between the two domains and returns a Flow Information Block (FIB) to the calling application.

Applications may use the FIB to configure the protocol code to format packets correctly for that particular flow. This configuration information includes such details as the local and remote IP address and port numbers, but also includes the source and destination (or next hop) Ethernet addresses. Although all of these details are provided by the Flow Manager to the application to enable it to generate correctly formatted packets, the *guarantee* that they are correct comes from the transmit filter downloaded from the Flow Manager into the device driver.

Our current implementation of the Flow Manager is used primarily for connection setup; its other functions as described in section 3.2 are not yet fully implemented.

### 4.2. Application protocol implementation

In order to minimize QoS crosstalk a fresh set of resources is associated with each new flow, so that no contention for them can occur. Each new flow receives two[8] Rbuf I/O channels, an Rbuf data area, and a FIB. The Rbuf data area is sized in an application specific manner.

An application is free to use whatever protocol code it chooses, but in order to allow us to experiment with a wide variety of protocol compositions, we have implemented a highly modular stack. Each protocol layer presents two Rbuf-like interfaces by which data can be sent to or received from the network. Because each protocol layer looks just like the direct connection to the device driver, layers can be composed with ease. Note that although the semantics of an Rbuf communications channel are used between layers, there is no need to actually have a real Rbuf I/O channel present: no protection domain is being crossed. Instead, ordinary procedure calls are used to pass iorecs from layer to layer, leading to an efficient and flexible implementation.

Since iorecs are used to describe packets in transit, a naïve way of adding a header to a packet would be to rewrite the iorec to have an additional entry for the new header (similar to adding an mbuf on the front of a chain in Unix). This is inefficient as it increases the length of the iorec, and adds embedded store management. Instead the flow-based nature of the communication is used to pre-calculate the total size of header required for all layers. Whilst this is standard practice in many fixed composition systems, it is rarer when protocols can be composed arbitrarily on a per-flow basis.

### 4.3. Device drivers

There currently exist device drivers for a number of network cards. Various support libraries exist to allow transformation of the iorecs into physical addresses for DMA, and for performing the packet filtering checks required by the Flow Manager. Application protocol stacks are required to use data areas with a sensible alignment. Non-aligned packets will be discarded. This permits numerous optimisations in the packet filtering code.

In our initial implementation the packet filtering was performed using the Berkeley Packet Filter (BPF) [11] because of its ease in porting to our environment. More recently we have exploited the trusted nature of the Flow Manager to use native code to demultiplex well-known protocols. For non-standard protocols we have adopted some of the parse tree flattening from MACH Packet Filter (MPF).

An additional benefit of MPF is its ability to demultiplex IP fragments by keeping relevant state. We intend to offer applications two different selective filters – one which does the complete job, and another which will only receive a packet if the fragments arrive in the correct order. The latter is likely to prove beneficial for environments in which unexpected fragment arrival is almost certainly due to fragment loss, and / or where reordering would be tedious for the application. Such an environment might be NFS requests over a local Ethernet segment.

We use a template-based integrated copy-and-check transmit filter routine, giving us a zero-copy transmit path.

### 4.4. Transmit scheduling

Since transmit bandwidth on a network interface is a real resource, the driver must provide mechanisms to schedule it according to QoS parameters. For interfaces like the DEC ATMWorks 750, this rate control is provided by the hardware. For other devices (e.g. Ethernet) this scheduler is provided by software.

The choice of scheduling algorithm is often dependent on the constraints of the device. The fact that the underlying real resource server may not be working at constant rate but is affected by the other traffic sharing the same media (e.g. Ethernet) complicates matters.

One of the principal issues is the ability of applications to explicitly state the time period over which their guarantee should hold, and therefore limit the “burstiness” which they

[7] Also implemented using shared memory and event channels.

[8] or in rare simplex cases, one.

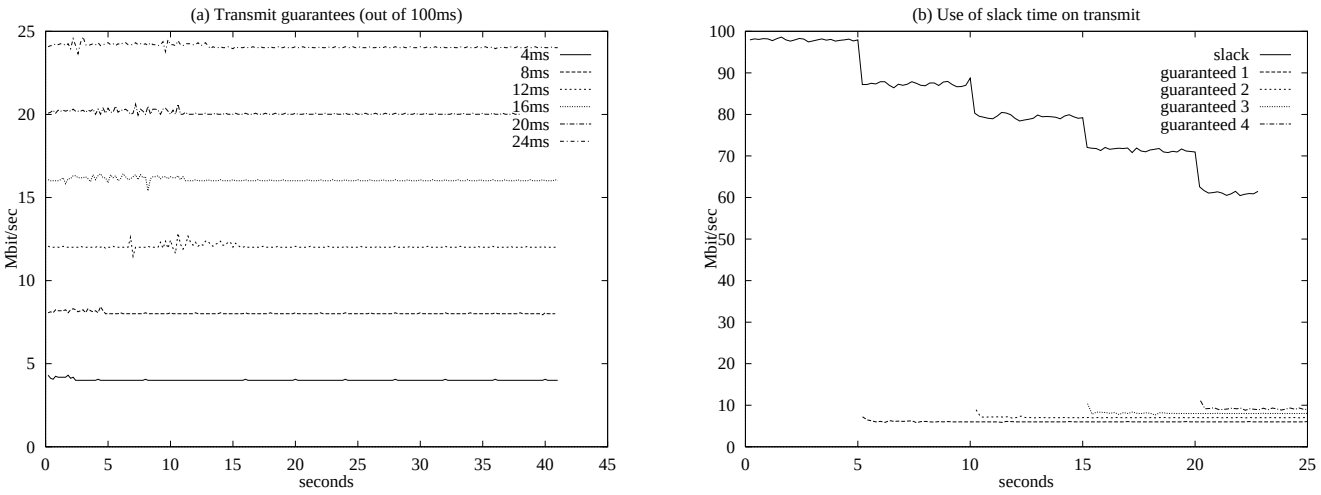


**Figure 5. Ethernet transmit: (a) QoS levels, and (b) slack time**

will observe. There is also a familiar latency / bandwidth tradeoff.

We have considered Stride Scheduling [16] and various other related algorithms [17], but at present we are using a modified form of the CPU scheduler already present in *Nemesis* (the *atropos* scheduler described in [10]). It guarantees a *rate* specified by an allocation period and a slice within that period, and also supports best-effort service-classes. This scheduler permits some amount of experimentation with the above issues.

## 5. Evaluation

Previous publications on Nemesis have reported on its abilities in handling Audio and Video streams. In this section we present four experiments which we have performed to measure the effectiveness of our QoS schemes for conventional protocols and network interfaces; we have chosen IP and 100Mbit Ethernet for our apparatus.

### 5.1. Transmit

The purpose of these experiments is to show that *Nemesis* provides an adequate mechanism for control of the real resource of transmit bandwidth on an interface. To demonstrate this, a simple test application was written which transmits a stream of MTU-sized UDP packets as fast as possible. The bandwidths obtained are recorded on another machine and integrated over 200ms intervals to give an approximation of instantaneous bandwidth. During the experiments, the *Nemesis* machine is also running various system daemons and is responding to ARP and ICMP correctly and in a timely manner.

In the first experiment, 6 identical copies of the test application are started. Each application however is given a different QoS guarantee, expressed as a number of ms slice out of a period of 100ms, and no stream is allowed to use slack time. The first stream is allocated 4ms out of 100ms (i.e. 4Mbits/sec); each successive stream is allocated a further 4ms, so that the sixth stream is allocated 24ms out of 100ms (for an expected bandwidth of 24Mbit/sec). Since all 6 applications are running concurrently, this sums to a total bandwidth of approximately 84Mbit/sec. Note that the guarantees are of access to the Ethernet device, not necessarily to the network due to collisions etc.

Figure 5(a) shows the bandwidths obtained by each application. It can be seen that the allocations are stable over the whole length of the experiment, and that each application obtains a transmit bandwidth proportional to its QoS guarantee.

In the second experiment, 5 copies of the same application are run on the *Nemesis* machine. The first copy is given no guarantee, but is allowed to make use of any otherwise unused transmit bandwidth. The subsequent applications are started at 5 second intervals and have guarantees ranging from 6ms to 9ms (in 1ms steps) out of every 100ms.

Figure 5(b) shows the Ethernet bandwidths obtained by each application. It can be seen that the bandwidth achieved by the initial application starts off at the full line rate, but decreases as each guaranteed stream comes on-line. The first guaranteed stream gets 6Mbit/sec and each subsequent ones get an additional 1Mbit/sec. They are unaffected by the first stream's use of slack time.

Finally, a comparison of best-effort transmit bandwidth against a modern Unix variant (Linux 2.0.29) was made. Since it is rare for Unix implementations to offer traffic shaping functions, best-effort performance is all that was tested. In both cases, two applications send UDP packets as fast as possible.

Figure 6 shows the bandwidths received over time for

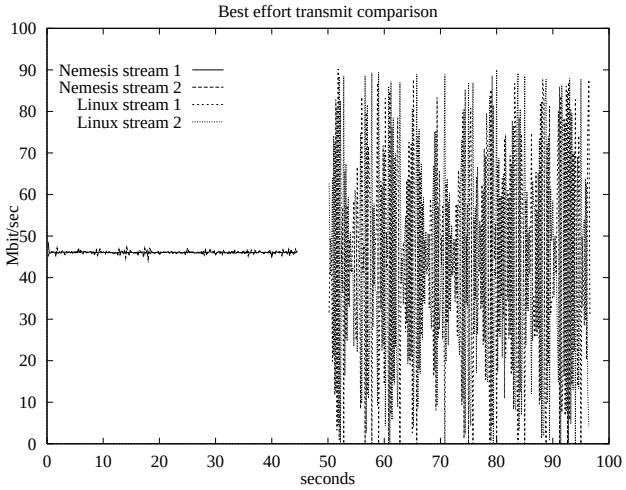


**Figure 6. Transmit, best-effort comparison**

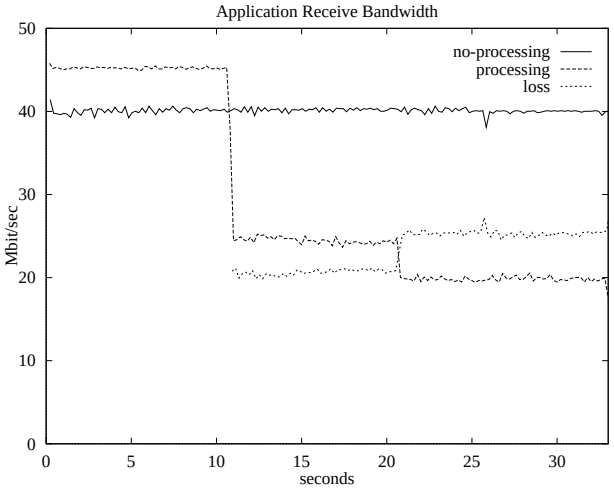


**Figure 7. Receive QoS performance**

the streams on *Nemesis* (the left trace) and Unix (the right trace). Over the time period graphed the average bandwidth obtained was 46.1Mbit/sec for *Nemesis* and 44.1Mbit/sec for Unix, furthermore it can be seen that *Nemesis* delivered that bandwidth in a very much smoother manner than Unix.

### 5.2. Receive

The purpose of this experiment is to show that the *Nemesis* structure we have developed avoids QoS crosstalk even in the case of receive overload.

In this experiment, a *Nemesis* machine (now known to transmit at a very smooth rate) is used to source two streams of UDP datagrams at precisely 40Mbits/sec and 45Mbits/sec.

A second *Nemesis* machine receives these streams, and runs one consumer application per stream. The first consumer application discards the UDP packets as soon as they reach the application's main program (at the top of the application stack). The second application "processes" each packet before reading the next. In addition, after reading every 20000 packets, it deliberately slows down its packet processing by spending a greater amount of CPU time on each packet – this is intended to emulate an overloaded application falling behind in its processing of network data. The desired behaviour is that the overloaded application, whilst causing the device driver to discard its own packets, should have no effect on the first application.

Both applications log each packet's size and arrival time, while the device driver logs every packet which it must discard.

In figure 7, "no-processing" is the Ethernet bandwidth received by the UDP application which is steadily receiving data at 40Mbits/sec without processing each packet. The second application's received bandwidth is plotted as "processing", and the packet losses recorded by the device driver are shown labelled "loss".

Each time the second application decreases its processing rate, the "processing" trace drops while the "loss" trace rises by the same amount. This occurs because the application is processing each packet for sufficiently long that the buffering it has been allocated has overflowed. Because packets are being discarded early by the device driver in a controlled manner, the well-behaved application receiving 40Mbit/sec is completely unperturbed by the actions of the trailing domain.

## 6. Conclusions

We are interested in implementing user-space protocol code for reasons very different from those previously published. Our concern for QoS support has led to a system where the device driver does as little per-packet processing as possible.

To this end, we have developed a scheme whereby, even for "dumb" network interfaces, the security of the data in both receive and transmit can be maintained without a copy in either direction. This scheme is implemented using a flow-oriented approach to all communications. An out-of-band Flow Manager is responsible for creating the data path and transmit and receive filters that will be used for the duration of the flow.

An initial implementation of our design has already been measured, and our goal of preserving QoS has been shown to have been accomplished.

We recommend the use of IP Flows and user protocol stacks for Quality of Service in addition to the previously published reasons.

## 7. Acknowledgements

Many people have worked on *Nemesis* over the years; without them, this work would not have been possible.

Figures 1,2,3 and 4 are reproduced in this paper from the *Nemesis* documentation with the permission of their respective copyright owners.

## References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] P. R. Barham. Devices in a Multi-Service Operating System. Technical Report 403, University of Cambridge Computer Laboratory, October 1996. Ph.D. Dissertation.

[3] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review*, 29(5):40–53, December 1995.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review*, 29(5):267–284, December 1995.

[5] R. Black. Explicit Network Scheduling. Technical Report 361, University of Cambridge Computer Laboratory, December 1994. Ph.D. Dissertation.

[6] CERT. *TCP SYN Flooding and IP Spoofing Attacks*, September 1996. Availible via ftp from info.cert.org.

[7] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Computer Communication Review*, volume 20(4), pages 200–208. ACM SIGCOMM, September 1990.

[8] 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*, volume 24, pages 14–23. ACM SIGCOMM, September 1994.

[9] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review*, 1995.

[10] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

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

[12] P. Newman, W. L. Edwards, R. Hinden, E. Hoffman, F. C. Liaw, T. Lyon, and G. Minshall. Ipsilon Flow Management Protocol Specification for IPv4 - Version 1.0. *Internet RFC 1953*, May 1996.

[13] I. Pratt. *User-Safe Devices*. PhD thesis, University of Cambridge Computer Laboratory, 1997. (In Preparation).

[14] T. Roscoe. Linkage in the Nemesis Single Address Space Operating System. *ACM Operating Systems Review*, 28(4):48–55, October 1994.

[15] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. In *Computer Communication Review*, volume 24, pages 64–73. ACM SIGCOMM, September 1993.

[16] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Mangement. Technical report, MIT Laboratory for Computer Science, June 1995. Technical Memo MIT/LCS/TM-528.

[17] H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. In *Computer Communication Review*, volume 21(4), pages 113–121. ACM SIGCOMM, September 1991.