Esprit LTR project 21917 PEGASUS II
Deliverable 5.1.1

A Networking Architecture for Nemesis

Bengt Ahlgren

`Bengt.Ahlgren@sics.se`
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
Tel: +46 8 752 1562
Fax: +46 8 751 7230

Austin Donnelly

`Austin.Donnelly@cl.cam.ac.uk`
University of Cambridge
Computer Laboratory
Cambridge, CB2 3QG
Tel: +44 1223 334600

October 29, 1997

1 Introduction

This document describes the new networking architecture that is being designed for Nemesis. It is the deliverable from Task 5.1 "Support for connectionless protocols".

Section 2 gives the background to this area. In section 3, we describe our prototype implementation, while section 4 presents our latest design. Finally, we present our initial experimental results in section 5.

2 Background

2.1 The structure of Nemesis

The primary concern in the design of the Nemesis operating system is to provide guaranteed quality of service (QoS) to application programs. It is therefore of utmost importance to account all resources spent by or on behalf of an application to that application, and this clearly includes network resources consumed.

QoS crosstalk occurs when multiple clients contend for a single resource whose access is synchronised or protected by a shared server. In order to keep this crosstalk to a minimum, the Nemesis approach is to reduce the use of shared servers, requiring applications to perform as much of their own processing as possible. Where shared servers are unavoidable, they are required to schedule and account all resources they use or mediate on behalf of clients.

This leads to the *vertically structured* design of Nemesis, in which functionality normally associated with the operating system is achieved by executing shared library code in the application itself. For networking this implies that received data must be demultiplexed to its ultimate destination as early as possible; applications can then perform protocol processing themselves. This is entirely in accordance with the views expressed in [7].

2.2 Flows

The accounting required to support communications QoS is most easily performed in the presence of explicit *flows* since this provides a well-defined entity to which guarantees can be attached, and against which charges can be made for resources used. Supporting a connectionless datagram-based networking technology such as IP leads to the need to identify, for each packet, which flow it is a member of.

The initial communications support within Nemesis was entirely based on ATM, where the connection oriented nature made associations between VCIs and applications straightforward. IP connectivity was provided by the x-kernel, but this was not a suitable starting point for the provision of meaningful QoS guarantees. There are several reasons for this. Firstly, the x-kernel assumes a single communications server model, where one superserver manages all network-related activity, including dataflow. Secondly, it mixes control messages with data path messages, making it hard to send the data on a separate QoS-controlled path. Finally, the x-kernel memory management and synchronisation facilities were ill-matched to the Nemesis model.

These problems could have been surmounted, but implementing a network architecture from scratch allowed us integrate network functionality into Nemesis more easily than continuing with the x-kernel.

The x-kernel "session" primitive is analogous to a Nemesis flow. A flow is similar to, but not the same as, a "connection": a flow is the unit of resource reservation, and may carry multiple connections. All communication is classified into flows, including connectionless data streams. The important feature of a flow in Nemesis is that it is on this level that quality of service is provided.

This notion is similar to that used in RSVP [4]. An RSVP flow is recognised using a flow classifier, which in IPv4 is a generic packet filter expression. This means that arbitrary properties of the packets can be used to define and identify a flow. IPv6 instead uses a flow identifier field in the packet header to identify a flow, removing the need for the packet filter operation.

The concept of a flow needs to be flexible enough to accommodate connectionless protocols. There are cases when they can be treated as having an implicit connection, which can be the case for many continuous media applications, but also for other applications. As an example, with the Berkeley socket interface an application can "connect" a datagram (UDP) socket to a specific destination. But there are also cases when the remote end point varies from packet to packet. This is the case for servers processing RPC calls from many different clients using a connectionless transport, e.g., queries to a portmapper or a name server. We also need to support these applications, and so introduce "wildcard" flows which fix only the local communication endpoint.

2.3 IDC in Nemesis

Nemesis supports lightweight inter-domain[1] communication (IDC) by using shared memory for transporting the marshalled arguments, and event counts[2] to synchronise access to the shared buffer. This is in part possible because Nemesis is a single address space operating system, so all domains view the same mappings from virtual to physical pages (albeit with different access rights).

[1] A Nemesis *domain* can be thought of as equivalent to a Unix process.
[2] The basic IDC primitive in Nemesis.

An interface definition language (IDL) is used to specify the types of interfaces between components of the system, allowing sophisticated run-time typechecks. The IDL is also used by tools to auto-generate marshalling code and client / server stubs at compile time.

Interfaces represent boundaries at which protection domains may optionally be crossed, thus the definition of the interfaces involved in specifying a service also constrains how the implementation of the service can be apportioned to various domains.

2.4 Bulk I/O

IDC is a mechanism for the transport of small amounts of data with tightly specified flow of control. It is generally unsuitable for the movement of large quantities of data, due to the synchronisation overhead. The Nemesis operating system provides the *Rbufs* mechanism for the transport of bulk I/O, described in fuller detail in [3].

The design is based on a separation of the three logical issues in I/O:

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

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

2.4.1 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 and a fast mechanism is provided for converting virtual addresses into physical addresses for use by drivers programming DMA hardware.

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 that the memory area is always accessible to the device driver without page-faulting. The data area is considered volatile and is always updateable by the domain generating the data.

2.4.2 Control areas

A collection of regions in the data area may be grouped together 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 domains pass iorecs between them to indicate which addresses are being used for each packet. 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 counts are 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 1).

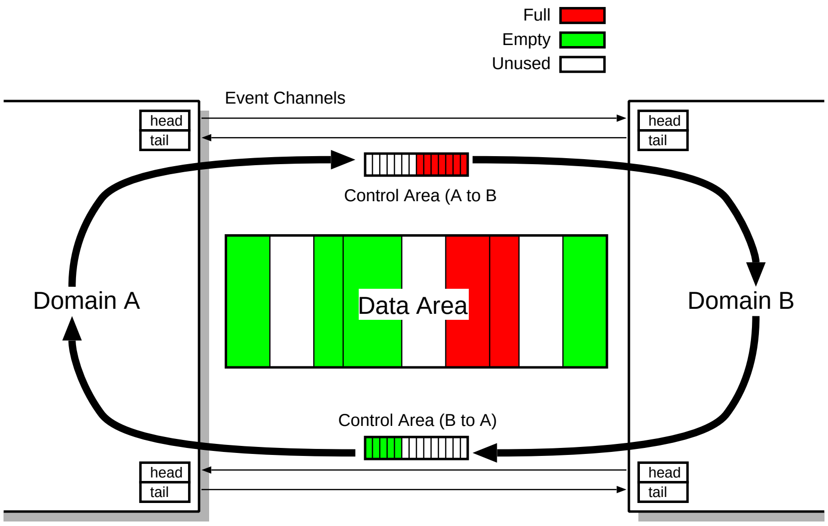

Figure 1: Control areas for an I/O channel

The sizes of the control areas are fixed per-channel; this provides a limit on the number of packets outstanding and an effective form of back-pressure preventing live-lock.

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.

2.4.3 Buffer management

An I/O channel can be operated in one of two different modes. In Transmit Master Mode (TMM), the data source chooses the addresses in the Rbuf data area, and the area is mapped writable by the source. By contrast, in Receive Master Mode (RMM) the *receiver* chooses the addresses, and the buffers are mapped with the permissions reversed such that the source still has write access to the buffers.

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

2.4.4 Complex channels

Multi-destination communications (e.g., reception of multicast packets to multiple domains) can be 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 The Initial Implementation

3.1 Separation of control and data

By separating control operations from data ones, the data path can be processed in a manner as unencumbered as possible. Given the vertical structure of Nemesis, it is desirable for user applications to interact directly with the hardware if possible. Unfortunately with today's devices this is not frequently realistic, and a *thin server* is needed for protection purposes.

The rôle of this server is to check permissions in a permission cache, translate requests or replies if needed, and multiplex hardware resources. If multiplexing occurs it should be scheduled to avoid QoS crosstalk, as discussed earlier in section 2.

In a networking context, this thin server is the network device driver.

3.2 Data path

In order to support the delivery of data directly to clients without the overhead of protocol processing, the network device driver must perform packet classification. This classification 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. If the hardware 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 classification. The driver 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 classifier has determined which protocol stack instance (i.e., 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-classification operation and (particularly) the copy consume CPU resources which are *not* directly accounted to the client and therefore have implications for QoS provision.

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 is achieved using a simple compare and mask; there is no demultiplexing on the fields and all information is pre-computed. Note that since the Rbuf data area remains concurrently writable by the client, the header of the packet is copied to some device-driver private memory as part of the checking process.

3.3 Control path

Flow setup and teardown is handled by a Flow Manager server domain. Once a flow 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 end point.

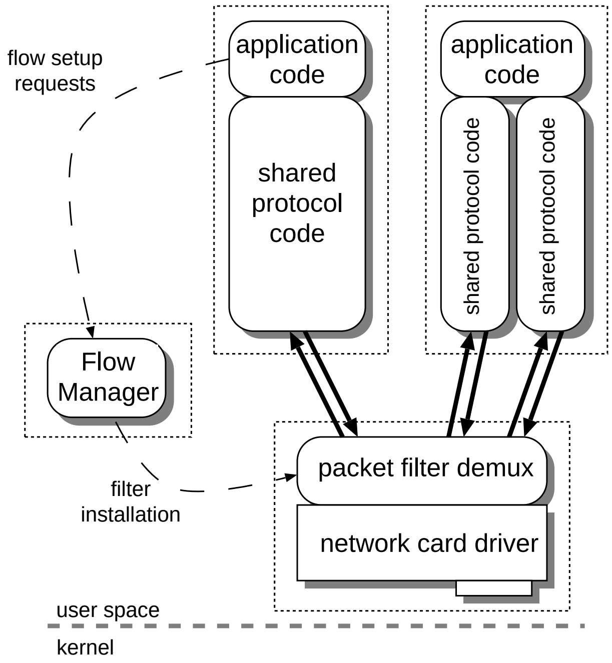

Figure 2: Interaction of components in the protocol stack

Figure 2 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 allocations and update driver demultiplex state in a consistent fashion.

Although the Flow Manager is responsible for installing the flow specification in the device driver it does not take part in the TCP 3-way handshake, which is performed entirely by the client.

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 ICMP handling.

4 The New Design

This section focuses on the parts of the networking architecture that have been redesigned and/or refined since the "snap3" version of Nemesis.

In the new design we redesign the interface to the flow manager to get a better separation between generic and protocol specific functions. Some protocol specific functions, such as routing table lookup, are now handled by the application protocol library. The flow manager interface uses a new Inter Domain Communication (IDC) feature in Nemesis which automatically instantiates a client specific interface to the flow manager at bind time. We are implementing an RBuf channel which avoids copying data from the device driver to the application specific buffer. This design also supports reception of multicast traffic to multiple clients. Support for multicast is a prerequisite for IP version 6.

Figure 3 shows a more detailed view of the Nemesis networking architecture. As described in the previous section, the three main components are the flow manager, the shared application protocol library and the network interfaces with packet filters for demultiplexing.

4.1 The flow manager

The flow manager is the logical home of a protocol stack. It implements the full functionality of the protocol stack and processes all packets that are not directed elsewhere. Examples are handling of ARP, ICMP, IGMP, sending TCP RST in response to incoming TCP SYNs for which there are no listener and sending ICMP port unreachable for UDP packets to ports for which there is no receiver.

When the flow manager sets up a new flow on behalf of an application domain, it *delegates* certain parts of the protocol processing to the protocol library in the application domain. These parts consist at minimum of the

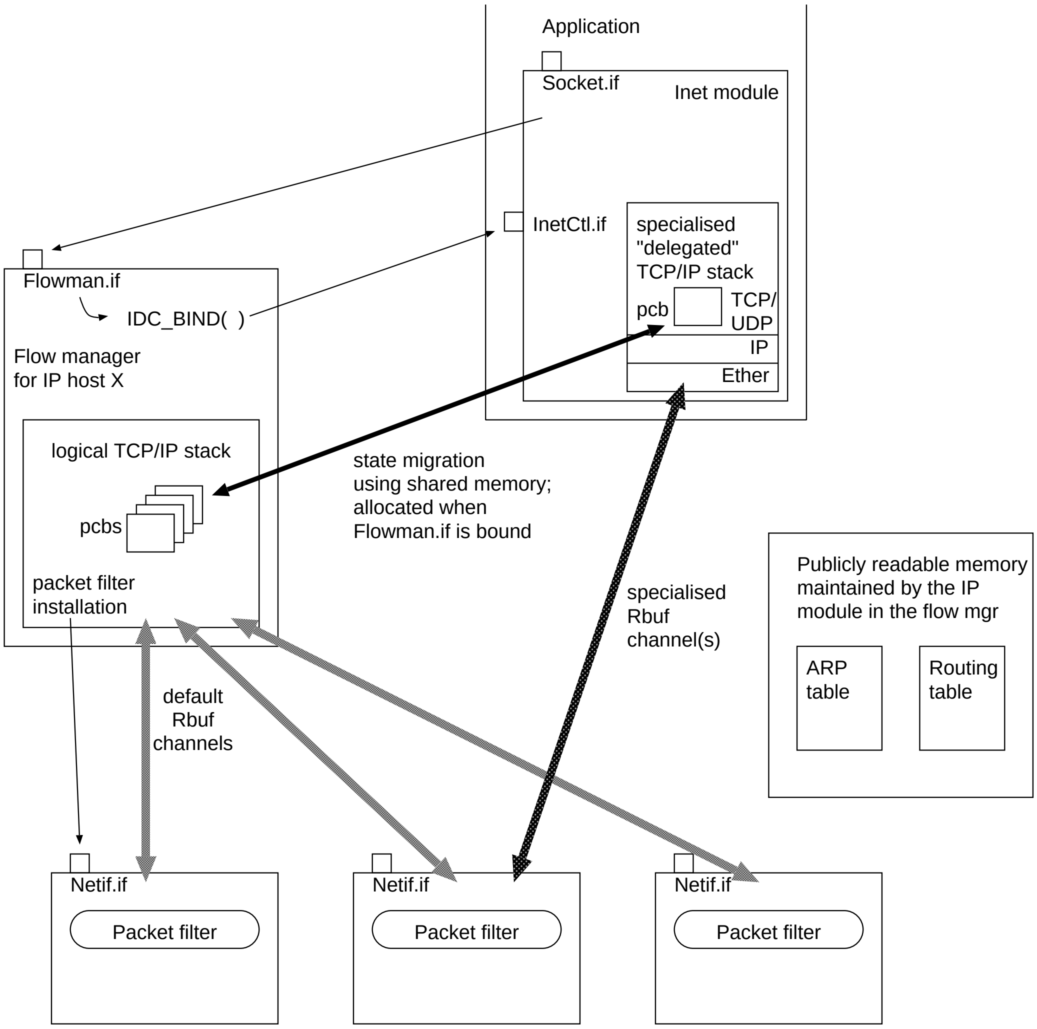

Figure 3: Nemesis networking architecture.

transfer of user data. The protocol library is specialised for the flow, which means that some protocol processing can be simplified and thus implemented more efficiently. Demultiplexing is already done by the packet filter that the flow manager has installed. For example, the TCP processing code can assume that all packets which it receives are for a single client "socket".

The flow manager provides the `FlowMan.if` interface to the application domain. It is an interface specific to an application domain that is created automatically at bind time. `FlowMan.if` includes the following methods:

`Bind`	Requests the allocation of a local connection end point identifier, i.e., a port number for TCP or UDP.
`UnBind`	Deallocates a local end point.
`Open`	Opens a new flow on a specific interface with a specific quality of service.
`AdjustQoS`	Modifies the quality of service on an existing flow.
`Close`	Closes a flow and deallocates all resources associated with it.
`Attach`	Attaches a connection to a flow. This makes the flow manager install a packet filter for the connection which attaches the connection to the flow.
`ReAttach`	Changes a previously installed packet filter for a connection. A typical use of this method is to fix the remote end point (port number) after a connection is set up.
`Detach`	Removes the packet filter for a connection.

Figure 4 illustrates the finite state machine for the relationship between the methods. The state machine is hierarchical in three levels. The outermost level describes the binding of a local end point (port number). The second level opens and closes a flow, i.e., creates and destroys the communication channel to the network device driver. The third level controls the installation of packet filters.

When an application wants to open a TCP connection it (or a library routine) does the following:

1. binds the local port with `FlowMan.Bind`,

2. finds out which interface to use via a protocol dependent function (which should if possible be provided by an application library function),

3. opens a communication channel to that interface with `FlowMan.Open`,

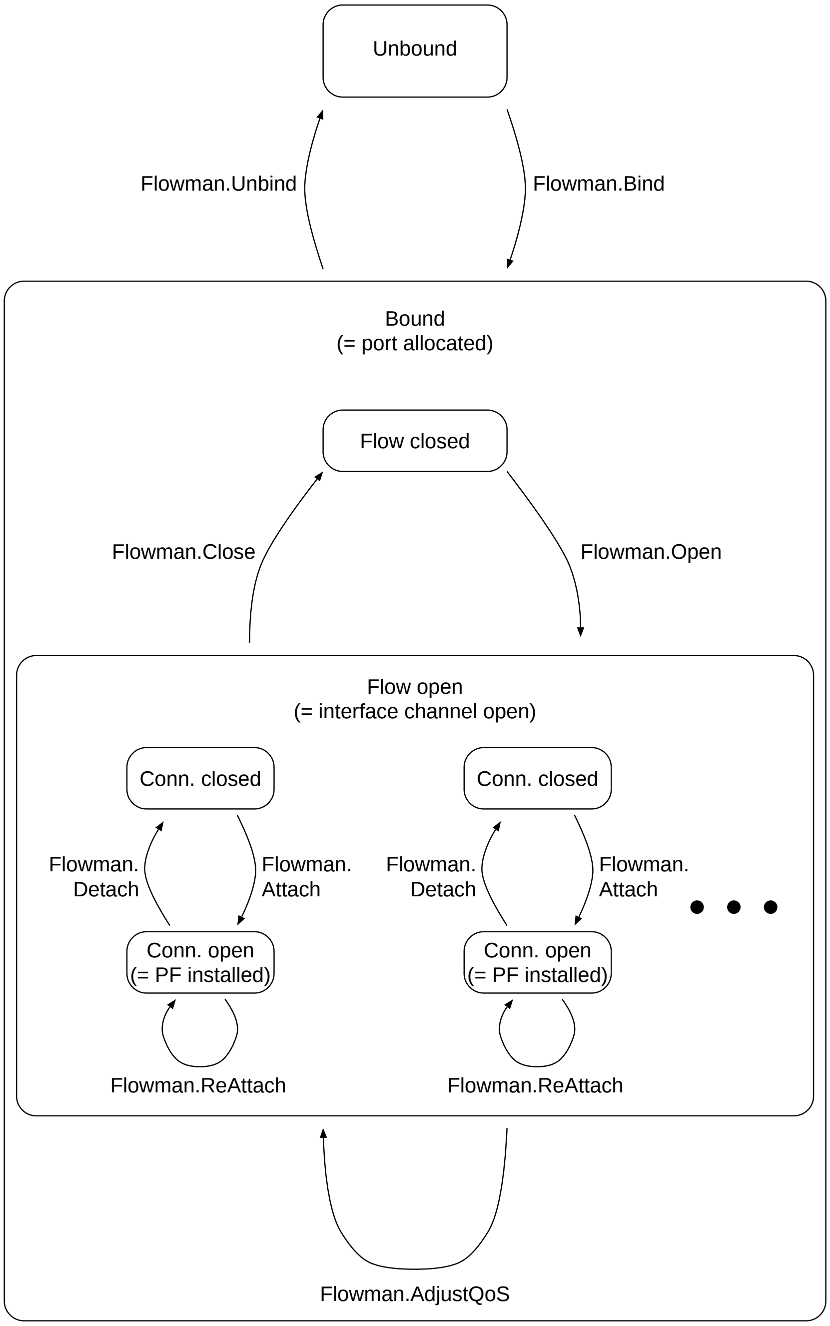

Figure 4: Flow Manager state machine.

4. attaches a packet filter with `FlowMan.Attach` for the port numbers of the connection, and finally
5. continues with regular TCP processing, i.e., sends a TCP SYN segment. The client application thus performs the complete 3-way SYN–SYN/ACK–ACK handshake of TCP. Since the installed packet filters restrict the application to using the correct addresses and port numbers, there is no loss in security with this design compared with a server design.

Listening for incoming connections is slightly more cumbersome. The application:

1. binds the local port with `FlowMan.Bind`,
2. finds out which interface to use via a protocol dependent function (which should if possible be provided by an application library function),
3. opens a communication channel to that interface with `FlowMan.Open` (possibly to more than one interface),
4. attaches a packet filter with `FlowMan.Attach` for the local port number of the connection to all channels,
5. waits for incoming TCP SYNs to the local port,
6. either opens a new flow for the connection (`FlowMan.Open` and `FlowMan.Attach`), or converts (one of) the existing flow(s) to a connection specific flow (`FlowMan.ReAttach`), and finally
7. continues with regular TCP processing, i.e., responds with a TCP SYN/ACK segment.

A problem that remains to be resolved is if and how it should be possible to hand off existing connections to new child domains. A possible solution is that the child domain uses a new flow for the same connection after the parent has completed the connection setup handshake for it.

4.2 Packet filtering

There are two packet filter implementations in Nemesis snap3, BPF, the Berkeley packet filter, and LMPF, a lightweight packet filter. We will also look at the DPF, dynamic packet filter from MIT [6], the MPF, Mach packet filter which has support for IP fragmentation, and the PathFinder to see whether they are suitable for our purposes.

4.3 RBuf communication channels

In Nemesis snap3, the RBuf communication channels from the packet filters to the application domains have their own heap where the packets have to be stored as previously explained in Section 2.4. This means that the packet filters have to copy data from the buffer into which the network device stored the packet using DMA.

The new Nemesis communication architecture removes the copy to the application domain by introducing a new RBuf implementation using a common heap where all received packets are stored both for packet filter processing and for application domain protocol processing. This is essentially the idea mentioned in Section 2.4.4 combined with the basic buffer handling from Fbufs [5]. The shared heap consists of click size stretches to be able to control page protection for each click separately. As with regular RBuf channels, there are separate control areas for each receiver. The new RBuf implementation can only run in TMM, transmit master mode, i.e., the sender selects how and where buffers are allocated.

This design also nicely supports no-copy demultiplexing of incoming multicast packets.

No additional QoS crosstalk is introduced with this design if the shared buffer pool is allocated a size that is (at least) the sum of the sizes of individual buffer pools, and if the delivery of packets to an application flow is limited by that application flow's backlog. This is similar to a credit based flow control scheme per application, but instead of stopping the flow when all credits are used, the newly arriving packet is not delivered to the backlogged application.

4.4 IP hosts

The new Nemesis networking architecture supports the notion of multiple "IP Hosts". Each IP host has its own port spaces for TCP and UDP. If the Nemesis system has more than one IP host defined, the client application has to choose which one it wants to use when it opens a new flow.

If there is one network interface for a particular IP host, it adheres to the "strong" host model. The IP host can also have multiple interfaces, which then share the same port space, and makes it adhere to the "weak" host model.

4.5 IP multicast support

The "canonical" Unix IP multicast implementation from Xerox PARC decouples reception of multicast packets on the IP level and delivery of the packets to specific transport layer (UDP) ports. The IP layer first accepts an incoming IP multicast packet if one or more sockets have requested to join the multicast group to which the packet is addressed. If the packet is accepted by IP, the destination IP multicast address of the packet is not significant any longer. It is only the destination UDP port that is used to deliver the packet to one or more applications. The packet is thus delivered to *all* UDP sockets that have bound to the packet's destination port (and that have multicast options set), regardless of whether that socket joined the group addressed by that specific packet.

In Nemesis we want port spaces for each IP multicast group that is separate from the unicast port space. We will therefore treat each IP multicast group to which a specific IP host has joined as a "sub-host" with its own port space.

A consequence of this architecture is that it is not possible to receive datagrams from several multicast groups on one IO channel. A separate channel has to be opened for each multicast group. The Unix IP multicast implementation uses `setsockopt` to add and drop membership. One possibility in Nemesis is instead to just bind the local address to the multicast address. This binding is however different from unicast in that the bound address is not used for outgoing packets.

The multicast IP hosts need to belong to a certain IP host, because its network interfaces, interface addresses and routing table is needed for proper operation. In particular, a source IP address is needed, so an interface is needed.

5 Results

Preliminary results from the initial implementation are encouraging. The following graphs were taken from a recently published paper [2].

Figure 5 shows the bandwidth obtained by five applications over the duration of the experiment. Each application attempts to send MTU-sized UDP packets as fast as possible. The applications are given different QoS guarantees, expressed as a guaranteed *slice* of time given over a *period*. The first application is allowed to use any unused transmit bandwidth: initially it sends at around 98Mbit/sec. The next applications are started at five second intervals, and have guarantees ranging from 6ms to 9ms (in 1ms steps) out of every 100ms. As each application starts up, the amount of slack time available falls, and the first application's bandwidth is reduced.

Figure 6 shows the bandwidths received by two applications sinking data, one a stream of UDP packets rate limited at source to 40Mbit/sec, the other limited to 45Mbit/sec. However, in this experiment, the behaviour of the receiving applications differ: the trace marked "no-processing" is for an application which discards all the data it receives as soon as it reaches the top of its protocol stack, while the trace marked "processing" is for an application which spends time working on the data an consequently cannot keep up

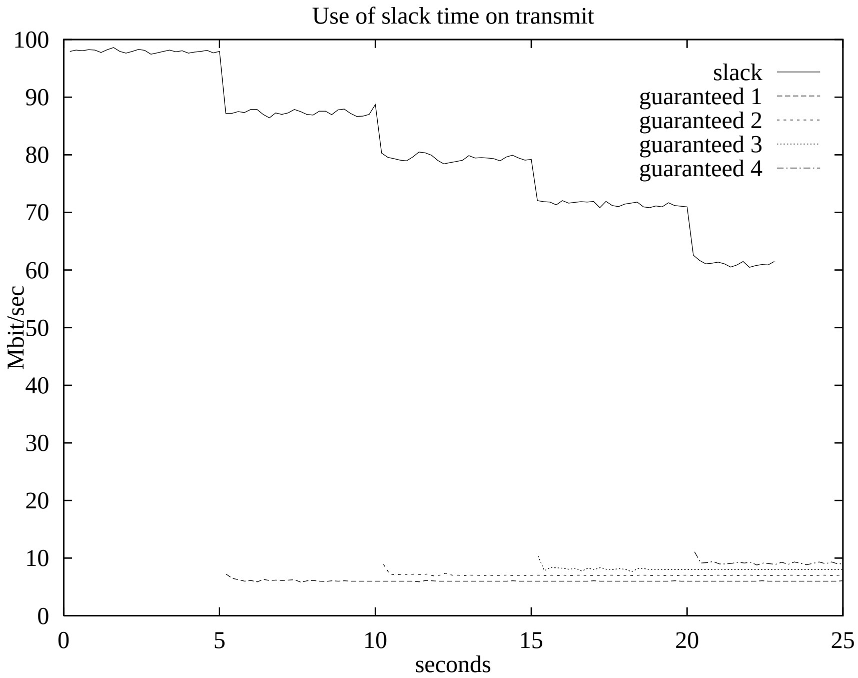

Figure 5: Transmit QoS

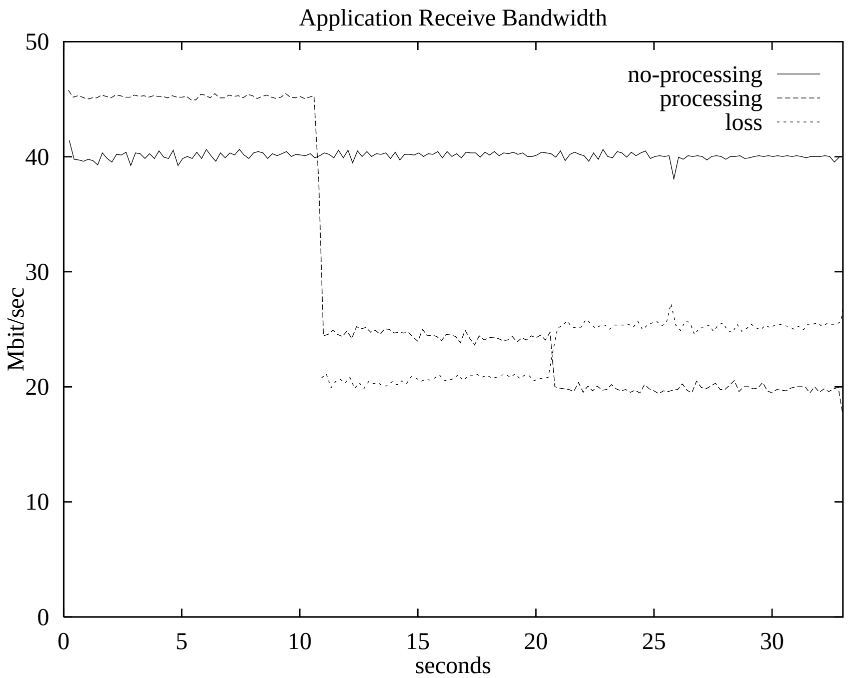

Figure 6: Receive QoS

with the data rate. Each time the processing application slows its processing rate further, the "processing" trace drops sharply and the "loss" trace rises by a corresponding amount; the "loss" trace shows the bandwidth thrown away by the device driver due to insufficient client-supplied buffers. Note that the other application which *is* keeping up is unaffected by the changes in behaviour of the lagging application.

References

[1] Bengt Ahlgren. *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption.* PhD thesis, Uppsala University, Uppsala, Sweden, March 1997. DoCS 97/80, ISSN 0283-0574. Also as SICS Dissertation Series 24, ISSN 1101-1335.

[2] Richard Black, Paul Barham, Austin Donnelly, and Neil Stratford. Protocol implementation in a vertically structured operating system. In *IEEE 22nd Annual Conference on Computer Networks (LCN)*, Minneapolis, Minnesota, USA, November 2–5 1997.

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

[4] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 functional specification. Internet RFC 2205, September 1997.

[5] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Asheville, North Carolina, USA, December 5–8, 1993. ACM SIGOPS Operating Systems Review, 27(5).

[6] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM '96 Conference Proceedings*, pages 53–59, Palo Alto, CA, USA, August 26–30, 1996. ACM SIGCOMM Computer Communication Review, 26(4).

[7] David L. Tennenhouse. Layered multiplexing considered harmful. In Harry Rudin and Robin Williamson, editors, IFIP WG 6.1/WG 6.4 International Workshop on *Protocols for High-Speed Networks*, pages 143–148, Zürich, Switzerland, May 9–11, 1989. North-Holland, 1989.