# *The Pronto Platform*

## *A Flexible Toolkit for Programming Networks using a Commodity Operating System*

Gísli Hjálmtýsson
AT&T Labs - Research
AT&T Shannon Laboratories
Florham Park, NJ 07932
gisli@research.att.com

***Abstract***

*In recent years there has been an increasing interest in introducing more and more advanced services and functionality into the network. This is the primary driver behind the ongoing research on active and programmable networking. A central part of this research is to design an extensible architecture for network nodes that can be dynamically specialized to provide customized network services, and to converge on a set of basic facilities provided by all nodes, and the interface to these facilities. While several architectures have been proposed, they have so far been realized using prototype operating systems or proprietary programming languages. Lack of robust platforms for experimentation remains a significant obstacle for research on new network services.*

*We have built a platform for research on active and programmable networks, based on Linux. Our changes to the kernel proper are minimal and are contained in a small kernel patch compatible with multiple versions of Linux (2.2.2 and higher), enabling us to benefit from the rapid evolution and industrial support of Linux in general, and specifically those parts that are outside our interests. In this paper we describe the platform and give examples of its use.*

## 1. Introduction

In recent years there has been an increasing interest in introducing more and more advanced services and functionality into the network. Although telephony services have long been network centric, we now see a significant push in Internet services to perform functions traditionally performed by hosts (end-systems) inside the network. The functions now moving into the network are more complex and operating at finer time scales than those traditionally performed by telephony services. Examples of these include firewalls, proxy cache services, hosting site load management [1,2,3] and transcoding services at streaming gateways [4]. New services such as multicast, reliable multicast, and wireless data services are also examples of services where advanced functions are moving inside the network. Although in general, new functionality may be put throughout the network, we see the majority of new service functions appearing at logical network boundaries, such as boundaries of domains of trust and/or authority such as at a campus gateway, of technology such as wireless and wired networks, or of capacity such as where the backbone network meets the access network.

While significant work has appeared over the last few years on enhanced functionality inside the network, each attempt introduces a new service architecture, a set of mechanisms and an approach on how to exploit the mechanisms in a novel manner. A major part of the ongoing research on active and programmable networking is to design an extensible architecture for network nodes that can be dynamically specialized to provide customized network services. An essential part of this research is to converge on a set of basic facilities provided by all nodes, and the interface to these facilities. The ultimate goal is to see independent service creators developing services on heterogeneous equipment from multiple vendors of commodity network elements.

We therefore need an operating system for an extensible network node – a *node OS* – that provides access essential network facilities sufficient for routers, proxies, and smart servers. Many parts of such a node OS are standard OS functions that while essential are independent of the networking parts, and of little interest in research and experimentation with active and programmable networking. Conversely, although researchers working on higher level network services, such as proxy services or smart server farms, need much of the functionality of a programmable router their primary interest is on other aspects of the operating systems, for example, file systems, and transport protocol enhancements. Research prototype OSes tend to have limited support for hardware and services outside the scope of the research that drives (drove) the OS. Further aggravated by limited maintenance (bug fixes) and support, users of a research prototype OS are forced to trade off good support for the parts they are interested in for the enhanced network support.

A number of architectures and service models have been proposed for network nodes supporting active and programmable networking [5, 6, 7, 8, 9, 10, 11,12]. Of those [5,7] and [9] operate at the user level, [8] provides programmability solely at the kernel, [6,11,12] use a combination of a user level environment and a proprietary kernel, and [10] uses a proprietary language and OS level runtime support. Whereas the user level execution environments are really above the interface of the node's operating system, kernel level programmability is more suited for vendor driven router extensions than for third party service development. Using a proprietary language and OS level support is a significant challenge for other practitioners.

We have built a platform for research on active and programmable networks. The platform is based on Linux, which is a commercially supported commodity operating system. Our changes to the kernel proper are minimal and consist only of elementary additions of hooks for the enhanced facility, for each component a single pointer and a couple of symbols. These changes are contained in a small and simple kernel patch compatible with multiple versions of Linux (2.2.2 and

higher). In contrast other efforts based on Linux, including previous versions of our platform, gratuitously change the Linux kernel effectively creating a proprietary version of the OS and thus are unable to keep up with the commercial evolution of Linux. The functions enhancing the commodity version of Linux are provided in kernel level modules that, after applying the simple patch, work with all versions of Linux post 2.2. This way we are able to benefit from the rapid evolution and industrial support of Linux in general, and specifically those parts that are outside our interests.

The primary contribution of this paper is the description of the platform which provides kernel level node OS for active network support without sacrificing the benefits of a commercial OS. The platform consists of: (*i*) a *set of kernel modules* that provide the enhanced functionality for supporting active and programmable networking; (ii) a kernel patch for a commodity version of Linux (2.2.2 and higher) introducing the hooks to allow the kernel to exploit the new modules; (iii) and a user level library, with a defined API to access these facilities. Since the node OS facilities are provided at the kernel level, the impact on performance is negligible (see Section 7). The platform is highly modular with very little coupling between individual components, allowing interested researchers to exploit only those facilities desired and easily combine their own components with those of the platform. For example, one may change the implementation of individual modules, while retaining the API and useing the user level library. The platform is developed as part of the Pronto router project [13].

The rest of the paper is organized as follows. After discussing related work in Section 2, we describe the Pronto architecture in Section 3. The Pronto router is designed to support multiple programmable models and is thus ideally suited as a Node OS for research on active and programmable networking, as well as for research on novel network services. In Section 4 we describe the kernel level part of the platform, and the user level library in Section 5. In Section 6 we describe some applications using the platform. We give three examples of services that we have prototyped using this platform: a) a active congestion adaptation, b) a service specific media streaming, c) and simple multicast. In Section 7 we finally provide some performance numbers for the various kernel level modules.

## 2. Related work

Although research on active and programmable networking is relatively new, the work described herein os most directly related to work on architectures and implementations of operating systems support at network elements. In addition this work is highly related to some of the work on programmable models and execution environments.

Peterson et al have been building a prototype OS, Scout, experimenting with novel systems abstractions for communication oriented operating systems and extensible routers [11]. Although, Scout is a valuable vehicle for research on system support for active networking, as a platform for research on other aspects of active and programmable networking Scout suffers from being a research prototype – limited maintenance, and limited support for features not needed for networking. Moreover, by virtue of its novel abstractions, user level programmers must cope with non-standard interfaces and with limited programmer support. These problems are the primary drivers behind creating the Pronto platform as a set of simple enhancements to Linux, rather than constructing a proprietary operating system.

A group at MIT has built active networking support on the Exokernel [12,14], designed and implemented to support the ANTS [5] capsule model. Although potentially useful for other capsule type active networking, being a research prototype this platform suffers from all of the problems discussed above. In fact the basic OS is even further from traditional ones, reducing the potential user base substantially.

The work on Router Plugins [8] is an approach to router programmability where router modules can be plugged in at the kernel level, thus customizing a router without the performance penalties of user level protections. Security and safety is provided externally to the programmable model. While we share many of the reservations expressed in this work with capsule type active networking, the Pronto platform is designed to allow the exploration of engineering choices for various models of active networking. One of the primary drivers of the Pronto work is to provide service differentiation using commodity components. In that context the forwarding engine is seen as the commodity network element, having service generic facilities provided by the router vendor, with service specific functions provided by service providers, users or third party service developers. The router plugin model does not support this separation between the service generic facilities and the service specific ones. In contrast, this separation is a hallmark of the Pronto architecture. Although not the topic of this paper, the Pronto router does support programmability below the abstraction interface. However, rather than being used by service providers, we see this as valuable for router vendors in evolving and maintaining the service generic functions.

The DARPA sponsored research on active networking has recently yielded a document on Node OS support for active network nodes [15]. The Pronto Platform supports all of the essentials in that Node OS, and using a popular commercial OS.

The most influential model so far on active networking is the capsule type active networking proposed by [16], and realized in ANTS [5]. In the framework of the Pronto architecture, ANTS constitutes an execution environment and may be realized using the in-data-path service model of Pronto (see below). The same applies for other programmable models and execution environments including [5,6,8,9,10,17]. However, unlike these models where the traditional forwarding model is replaced with a new active one, we focus on identifying and separating the service specific functions from the service generic facilities and define the interface between the two. The Pronto interface supports all of the capabilities of these models, and in addition offers a range of other service models. Finally, this work also draws strongly on some of the work on control plane programmability in ATM networks [18]. The first service model of Pronto provides comparable capabilities (see below).

## 3. The Pronto Architecture

The Pronto active router architecture is depicted in Figure 1. It consists of three major parts: a forwarding engine providing service generic facilities, an execution environment where service specific functions reside and execute, and the interface between the two. The most essential part of the architecture is the separation of the service generic facilities of the forwarding engine from the service specific functions running in an execution environment. The execution environment consists of a virtual machine and an environment manager. The environment manager is responsible for authenticating, installing and managing the programs executing in its environment. In addition, the environment manager dispatches control messages to programs in its environment. Each environment manager has access to an HTTP proxy, and a certificate authority proxy. Typically both are run on the router. Although execution environments can be dynamically installed or removed, we assume that these are infrequent operations, and are performed with super-user privileges. The only execution environment we have used in the prototyping of the interface described herein is the native virtual machine. However, the router can support other environments that have been proposed, including those proposed in [5] and [9]. We are currently implementing the interface in Java. Moreover the router can support multiple execution environments simultaneously.

Below the abstraction interface is the forwarding engine and other service generic operating systems and hardware facilities. We have identified six main abstract facilities. In the data path

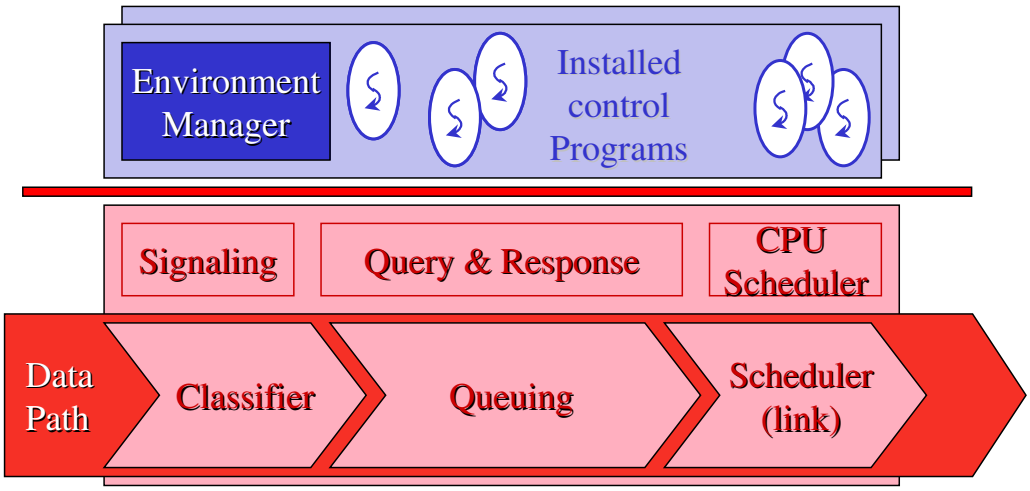


**Figure 1 : The Pronto architecture**

the three major components are a flow classifier, a queue manager and a link scheduler (see Figure 1). These facilities currently exist in all routers, although typically less capable than our architecture allows. For example, in traditional routers the classifier classifies only on destination prefix, and the queue manager has a single FIFO queue, and thus a NULL link scheduler. More recent higher-end routers support richer classification, the ability to configure multiple queues and provide simple scheduling among these queues. We push the envelope on each of these components in terms of functionality, flexibility and programmability. In particular the data-path can support fine-grained buffer management and link scheduling among flows and groups of flows.

Outside of the data path, Pronto includes a service generic signal processor, a resource manager, and a query and response engine (QRE). The first enhances and generalizes current IP signaling to better support service specific signaling. The resource manager schedules access to processing and memory, and ensures resource isolation of the service specific programs. The QRE supports queries and subscription for flow statistics and packet content. An important issue is to minimize data movement across the abstraction boundary. This is achieved using "frame peeking" that supports queries (subscriptions) to a limited number of bytes per packet. The frame peeking gives limited read access to the data-path, asynchronously to data forwarding and thus incurs negligible performance degradation. After processing the result of a query (or subscription) the service control program may request per packet actions on the packets in the queue, in particular packet discard, changing the drop priority or setting a congestion-indication bit. Part of the ongoing research conducted as part of this work is to conservatively enrich the set of QRE primitives and published events, and evaluate the benefits and complexities of each enrichment.

### 3.1 Summary of the supported service models

The Pronto architecture supports a range of programming models that differ in how they interact with the data path. Thus the Pronto router does not impose a model, but is designed to facilitate research on various execution environments and programmable models. On one extreme, a Pronto service could operate purely in the control plane, with no interference in the forwarding path. On the other extreme, the service could reside fully in the data path, operating on every data packet. In addition, we have identified two important intermediate cases. In the former, the service exploits the QRE facilities to support fine time-scale control plane functions and frame peeking, to tailor policies based on network conditions and on parts of the content of each packets. The second intermediate service model provides a node-local multicast where packets are forwarded normally, but also delivered to the associated user process. This gives us *four basic service models*: a pure control plane, finer grained control plane with frame peeking, local multi-

cast, and the last acting fully in the data path. These four service models become progressively more capable and costly, with the first two having no impact on data forwarding performance, the third incurring a duplication cost for local delivery, and the in-data-path processing incurring the execution of a user-level specific program on every datagram. Strict control plane programmability as in [18] is supported by the first model. The approaches to active networking of [5] and [9] are supported by the last.

An important part of the architecture is that for the first three service models, the service specific programs are applied asynchronously from data forwarding. This is clearly the case for the control plane service model. Using the frame peeking and the local multicast service models, data packets are forwarded asynchronously to the service specific processing. Consequently, services employing these service models act as enhancements only, and are not essential for forwarding correctness. Due to the enhancement nature it is not essential to process every packet. Clearly the more predictable scheduling of the service specific programs the greater the enhancement. In our work on Pronto, we emphasize typed flows rather than individually typed packets. This allows for optimizations and amortization of cost over multiple packets, but also provides an abstraction for services to act at the connectivity level as well as at the data level.

A crucial property of the Pronto architecture is that different service models may be applied to different flows and sub-flows. In particular, one may use a different model for the forward flow than the reverse flow of a particular application level connection. For example, to facilitate retransmissions of lost packets from a Pronto based proxy to a client, the proxy would trigger the local multicast capabilities to receive a copy of every packet of the forward flow, without interfering with the forwarding path. The proxy would then use the in-data-path model for the reverse flow, intercepting all the ACK or NACK packets from the receiver(s) and use those to trigger the retransmission of lost packets. A Pronto node may use a different service model at different times of the transmission, depending on observed data, network conditions, and resource availability. As an example of this, a service level router might intercept the first few packets of a new flow to determine the appropriate policies and service quality to apply to the flow, before delegating the work to the networking layer for the remainder of the flow. In yet another case, a Pronto router would leave the forwarding path untouched except during congestion, where it would selectively discard certain packets (such as the *B* frames in an MPEG stream). Some of most intriguing research issues in the interface design of the architecture are how and to what extent this type of delegation can be achieved through the interface.

Another valuable property of the Pronto architecture is that the service specific programs may employ different service models at different parts of the network. The appropriateness of a programmable service model in part depends on the ratio of CPU cycles to bandwidth. This ratio is different at different parts in the network hierarchy. An access gateway node terminating modem connections in software is anyway fully in the datapath and has significant number of CPU cycles per bit transferred. A high-end backbone router has enormous throughput, but comparatively limited processing capacity. In between, this ratio between bandwidth and processing gradually changes. For a particular flow a service specific program may exploit a different service model at different parts of the hierarchy, for example using only control plane programmability on the backbone, and acting more aggressively towards the network edge.

## 4. The kernel level facilities – The Node OS

The Pronto platform defines a set of interface primitives that service specific programs used to interface (and interact) with the underlying forwarding engine. The architecture defines six major components below the interface, a packet classifier, a queue manager, a link scheduler, a CPU scheduler, a control message dispatch module (signaling), and a query and response engine

(QRE). We have implemented all of these components and their interfaces. The rest of this section describes the interface to each of these components.

### 4.1 The packet classifier

The most essential facility of the forwarding engine is the packet classifier, which separates the packet stream into flows. Abstractly, the classifier maps packets into equivalence classes based on attributes of the packet. Typically these attributes include the IP header and the transport level header. The equivalence relation is defined by a filter, $F$, so that two packets, $p_i$ and $p_j$, are equivalent, iff $F(p_i) = F(p_j)$. We define the equivalence classes in terms of a byte mask, M, and a value $V$, such that $F(p) = ( p \ \& \ M = V )$, where the & is the bitwise and operator. Multiple filters may map to the same equivalence class, i.e., an equivalence class is defined as $E = \{ p \mid F_0(p)$ or $F_1(p) \ldots F_n(p) \}$. Each equivalence class has associated with it a state, in particular the forwarding state. The forwarding state consists of a set of queues and output devices (in the case of a unicast this set is a singleton), or a local socket, or both. Additional attributes may be assigned to an equivalence class, including a request for flow pinning, reservations, restoration and more. The most important attribute for network programming however is the program associated with the equivalence class, which effectively defines it's type.

The packet classifier supports the following primitive operations:

- **add**( filter_definition, devices, queues, socket ) **return** flow_handle
- **remove**( flow_handle )
- **update**( flow_handle, device, out_queue, socket ) **return** flow_handle
- **get**( flow_handle ) **return** { list of <device, queue> pairs, socket }

### 4.2 The queue manager

Queues are manipulated indirectly through the link scheduler, and the query and response engine. Whereas the link scheduler manipulates the queues by scheduling packet transmissions, and by assigning queues to the scheduling hierarchy, the content of queues can be manipulated through the query and response engine. The only direct interaction with the queue manager is to create a queue, using the primitive:

- **create_queue**( name, device ) **return** queue

### 4.3 The query and response engine

The query and response engine allow the installed control programs to access the data path asynchronously either by querying or through subscriptions. The interface implements *frame peeking* allowing the service programs to copy a small number of bytes out of each packet from a given offset within the packet. Subscriptions allow the service specific controllers to monitor the progress of flows, for example by subscribing to sequence numbers of a particular flow. Upon issuing a subscription, the interface returns a handle (a future), which is evaluated at some later time by the user level program by retrieving the result. The result of a subscription is a stream of fixed length peek segments, one segment for each packet forwarded. Each segment is the result of peeking into the packet as declared in the subscription.

A query gives a limited read access to flow statistics, and to the packets in the queue at the time the query is executed. The result of a query is a signed structure, containing flow statistics and an array of segments of peek results, one segment for each packet in the queue (Figure 2). A peek result is simply the bytes copied from each packet (i.e., peeked at). The statistics include data transmitted, data queued and data lost, given in bytes and packets.

| *signature* | *statistics* | *Array of peek results* |
|---|---|---|

**Figure 2 - The query result structure**

After processing a query the service specific program may respond with a signed array of operations, one operation for each packet (Figure 3). The signature of the response is that of the query and allows the forwarding engine to decide if the response is valid. The response engine processes the response by applying the operations to the corresponding packets. As the query is processed asynchronously to data forwarding, some number of packets may be transmitted before the response arrives. To account for this the forwarding engine keeps a count of packets sent since the last query, and ignores the operations for those packets already gone. When the count exceeds a threshold it invalidates the signature (a simple way to realize this is to initialize the counter with the queue length when the query is issued and then decrement the counter on packet transmission down to 0).

| *signature* | *Array of operations, one operation per pakcet* |
|---|---|

**Figure 3 The response structure.** For each packet of the corresponding query, the response specifies an operation. Operations may vary in length.

The four primitives supported by the query and response engine are:

- **query**( queue, offset, length, buffer *, buffer_len) **return** bytes_received
- **subscribe**( queue, offset, length, buffer *, buffer_len) **return** subscription_handle
- **retrieve**( subscription_handle ) **return** bytes_received
- **response**( queue, response structure )

### 4.4 Resource management

To provide performance isolation and to support service differentiation we have implemented fine-grained hierarchical start time fair scheduling, both for processing and network transmission. Our scheduler does not provide real time guarantees, but provides cumulatively predictable scheduling (for details see [19]). The schedulers schedule the two resources *independently*. This is consistent with our enhancement service model, greatly simplifies the scheduling implementation, and provides sufficient assurances for the applications we have considered.

#### 4.4.1 The link scheduler

The link scheduler implements predictable and fair bandwidth allocation. Equivalence classes are mapped to queues and output devices (ports). For each output device a scheduler is created and initialized, creating a root for the scheduling hierarchy and a default FIFO queue. The link scheduler interface supports the following seven primitives:

- **init_scheduler**( device ) **return** scheduler
- **get_root**( device ) **return** node
- **add**( name, path, weight ) **return** internal-node
- **add_leaf**( name, path, weight, service-policy ) **return** leaf-node
- **find**( name, path ) **return** node
- **remove**( node )
- **attach**( queue [, weight ] )
- **move**( queue [, weight ] )

### 4.4.2 The CPU scheduler

The CPU scheduler is realized in an analogous manner as the link scheduler. However, the scheduler is initialized and the root node created as part of the kernel initialization. Apart from the root manipulation, the interface is analogous:

- **add_node**( name, path, weight, service-policy )
- **find**( name, path ) **return** node
- **remove**( node )
- **attach**( processor-id )
- **move**( processor-id )

## 4.5 Control Message Dispatcher

The control message dispatch interface is used by a signal processor to subscribe to control messages from the forwarding engine. There are two levels of signal message processing, one processing service generic messages, the other processing service specific messages. Each environment manager is responsible for dispatching service specific messages to the appropriate control processes. Although in principle service generic messages could be processed by a single entity, for simplicity in our prototype we simply provide a library for this processing that may be linked into each environment manager. In our prototyping each environment manager is a user level process, started with super-user privileges and thus able to open a raw socket. The only primitive of the control message dispatch, **register environment**, is thus essentially setting IP level socket options to this raw socket. The primitive is specified as:

- **Register_environment**( socket, protocol_id, alert values (as a vector))

# 5. The user level library

The user level library is a class library for C++. A hierarchy of classes represent each facility. At the root of each hierarchy is an abstract class, defining the basic facility interface. This interface represents the system calls supported by the particular facility (in C++ defined as an abstract class). Derived from this class are implementations classes implementing the user level mapping and some additional library support. In some cases, only one implementation exists. For other facilities different classes represent different specializations of the facility, providing a more "user friendly" interface specialized for particular tasks, adding methods that automate common actions. Because of this hierarchy it is easy to customize the user level interface without affecting the system call implementation. To use a facility, the user program simply declares an instance of the appropriate implementation class.

We do not support cross communication between different execution environments. This is simply not a priority. Instead we are primarily concerned with being able to experiment with those environments of interest in isolation. We will address this issue when research on execution environments converges with the emergence of a consensus on what type of inter-EE communication is needed

# 6. Applications – Functional evaluation

To verify the usefulness of the platform and the viability of the Pronto architecture we have implemented numerous applications. We discuss three of them below. The first one is stream thinning as congestion adaptation and uses the QRE interface to do selective discard on a media stream, using only the low-end programming models supported by Pronto. Thinning an MPEG stream has become the canonical example of active networking, but all prior discussion puts the stream adaptation fully in the data-path. The second example, application-specific traffic shaping on a transparent Pronto proxy, also programs only the control plane, using the QRE interface to

```
1.  Selective_Discard() {
2.    query frame_type(Q, offset, sizeof(byte), buflen);
3.    While (1) {
4.      Sleep(some time);
5.      queryResult<byte> matches := frame_type.query ();
6.      int Qsize = matches.qsize();
7.      if (Qsize < highWaterMark) continue;
8.      response<int> r = new response( matches );
9.      for ( i=0; i < matches.size(); i++ ) {
10.       byte frame_type :=  matches[i];
11.       if (frame_type != I-frame and Qsize > lowWaterMark) {
12.            r[i] := DISCARD;  Qsize−−;
13.       } // endif;
14.     } // endfor
15.     frame_type.respond( r );
16.   } // endwhile
17. }
```

**Figure 4: Code segment implementing selective discard for MPEG**

get statistics on the progress of the media stream. In the third and last example, reliable multicast, the control program exploits the more aggressive program models, using the local copy capabilities for the forward path, but acts fully in the data-path on the reverse flow, demonstrating how the different program models can be applied on per flow (or even sub-flow) basis.

### 6.1 Service specific congestion adaptation

We have implemented selective discard as congestion adaptation for two types of video streaming. The former usies normal MPEG encoding, the second uses a multiple-descriptor coder (MDC) developed at AT&T Labs [20]. MPEG is hierarchically coded using three types of frames: I-frames, P-frames and B-frames. Whereas the loss of an I-frame can affect all frames until the next I-frame (a group of pictures), loosing a small number of P- and B- frames only marginally degrades quality. The MPEG selective discard policy discards the less important P- and B-frames in an attempt to protect the I-frames. MDC video has two (or more) substreams of equal importance. The receipt of either of the substreams for a particular frame provides "adequate" quality, while receiving both substreams renders the complete picture. Our selective discard policy strives to ensure that each frame is received from at least one of the streams.

Our approach to congestion adaptation is based on the observation that during congestion buffers are long and thus queuing time significant. Figure 4 shows a pseudo code for the MPEG stream thinning. When run, the video stream controller executes a query (Figure 4, line 5) and blocks. When invoked the request returns a *queryResult*, including the flow and queue statistics, and a list of entries for each packet currently in the queue, each entry containing the result of the frame peeking query (in this case the frame type from the packet). Most of the time, the queue is small (or empty) and the controller goes to sleep. If however, the queue exceeds a high water mark, indicating congestion, the controller prepares a *response*, consisting of the signature of the query and a vector of operations, one for each packet, discarding all B and P frames until a low watermark is crossed. On completion the controller returns the response using the respond primitive (Figure 4, line 15).

This example also nicely illustrates the effectiveness of frame peeking for stream thinning. We used this scheme on a short MPEG movie, encoded into 26,546 packets, a total of 13.86 MB. Most datagrams are of size 560 bytes, with few of size 170 bytes, with an average of 522 bytes

per datagram. Peeking at the full application level header (17 bytes) would have resulted in about 450 KB (3.3%) of the data being copied to user space. Copying only the one byte needed to identify the MPEG frame type (as is the case in Figure 4), reduces this down to 27 KB (0.2%). Blocking a packet takes only one byte of response keeping the data copying at 0.2%. Enlarging the packets (the MTU in our testbed is 1500 bytes) would lower this ratio further.

### 6.2 Application-specific multimedia smoothing

As another example to demonstrate the functionality of the Pronto interface we consider a work-ahead smoothing service. This service could be offered on a last hop router or by a service enhancement proxy. The service effectively performs application-specific traffic shaping to reduce the burstiness of a variable-bit-rate video stream, while avoiding underflow and overflow of the client playback buffer [21]. The online-smoothing service has two natural time scales: a medium time scale for computing a transmission schedule consisting of target transmission rates (based on the frame sizes, and their playback times), and a fine time scale for coordinating the transmission of individual packets based on the schedule. The smoothing service subscribes to flow statistics, including packet sizes and RTP timestamps, but does not need to inspect or copy the packet contents. When activated by the CPU scheduler, the smoothing controller executes code to determine the next rate, and gives it to the packet scheduler. At the data-path level, the packet scheduler output packets at the specified rate. While the controller applies service-specific criteria to set the schedule, the performance-critical work of scheduling packets for transmission is delegated to the forwarding engine. For the smoothing to be effective, the CPU scheduler must run the smoothing controller before the schedule is depleted (it is acceptable to run it too early). Assuming that each rate is used for transmitting several packets this requirement is not hard to meet. Therefore, using the Pronto platform the transparent smoothing proxy service can be realized in an asynchronous manner with negligible impact on forwarding performance.

### 6.3 Reliable multicast

Whereas the first two examples exploit the QRE interface, to implement reliable multicast in an efficient and scalable manner we exploit the ability to act more aggressively in the data-path. There are two main issues in scalable reliable multicast: to delegate retransmission work from the sender to other participants in the multicast, and to limit retransmissions to only those receivers needing the retransmitted packet. We assume losses are indicated by NACKs.

An application specific repair agent (program) is installed and run on routers in the multicast. For simplicity we assume this is done on all routers of the multicast. (Measurements on the MBONE [22] suggest that repairs need only be offered at a few strategic routers). The session uses multiple multicast groups, say $G_0$ to $G_n$. The original transmission is sent on group $G_0$. Retransmission of a packet whose sequence number mod $n$ is $s$ is sent on group $G_{s+1}$ (see [23] for details). All receivers subscribe to group $G_0$, and then join and leave the retransmission groups, $G_1$ to $G_n$, to recover losses. The repair agent uses the local multicast capabilities to receive a copy of all packets on the forward path of the multicast (i.e., opens up a local socket and registers it in the flow state of all $n + 1$ groups). For the reverse path, the agent goes fully in the data-path, intercepting every NACK.

The repair agent maintains a circular buffer of the last $k$ packets received on the forward path. When gaps in the packet sequence occur, it requests a retransmission as if it were a receiver, by sending a NACK upstream. When receiving a NACK for packet $i$, the repair agent determines if it has the packet, in which case it immediately sends it out on $G_{(i \bmod n)+1}$. If the packet is not locally available, but the agent is already awaiting its retransmission it drops the NACK. If neither is true, it forwards the NACK upstream, and then adds the incoming interface of the NACK to

the multicast group of the missing packet, $G_{(i \bmod n)+1}$. Finally the agent adds the packet (number) to a "*waited-for*" set on that interface for group $G_{(i \bmod n)+1}$. The last step is needed to account for wrap-around mod $n$, and thus the possibility that the agent is awaiting multiple packets on the same retransmission group. Under reasonable loss assumptions this probability small [23]. When receiving a retransmission on $G_j$ the agent removes the packet from the waited-for set on all output ports of $G_j$. If this leaves the waited-for set empty for a particular output port, the agent removes that port from the forwarding state of $G_j$.

Using this scheme the retransmission work is delegated to the routers of the multicast thus achieving scalability. The scheme effectively implements a data driven join and leave for the retransmission groups, enabling fast group management. The use of multiple multicast groups allows us to exploit the underlying forwarding engine and is thus efficient. Using relatively few groups the receivers receive only small amounts of unwanted retransmissions [23]. Note that the repair agent caches packets for a limited time (as newer packets override the older ones), and forwards NACKs that it cannot serve, but suppresses superfluous NACKs to avoid implosion. Receivers periodically repeat their NACKs for missing packets. Thus the *responsibility* for reliable delivery ultimately remains at the end-systems. In fact, this scheme might be implemented within a service domain transparently to the sender and receivers. Although the reverse path flows completely through the repair agent, the forward stream is forwarded directly by the forwarding engine. Under reasonable loss assumptions, the reverse stream will be a small fraction of the forward stream.

## 7. Evaluation – Performance numbers

Although agreeable that optimizing performance is not the first issue in a new research field, the underlying assumptions and platform capabilities inherently impact the performance achievable in ways that may not be correctable by subsequent optimizations. The Pronto platform is designed to minimally perturb the fast path of data forwarding, to ensure that any performance degradation is caused by the service specific functions, and not by the forwarding engine facilities. This is crucial to enable service level research to make meaningful trade-offs between the various programmable service models of the Pronto platform. In addition, as the active and programmable networking community struggles to demonstrate the potential value of programmability, it is important to minimize the hurdles that have to be crossed. A significant hurdle so far is achieving adequate performance.

In this section we present performance measurements of the Pronto router. The router is constructed using Linux 2.2.2, and the Pronto platform. The Pronto router is a single CPU Dell PC with a 450MHz Pentium processors using 10/100 Mbps 3Com network interfaces and their standard Linux driver. The end systems in this test use the same hardware. We measure the throughput of the Pronto router using two metrics: packets per second, and bits per second. The former metric peaks with minimum packet size, and is thus limited by the CPU. The latter is limited by network bandwidth but peaks just below the MTU of 1500 bytes. For each measure we show four different configurations, Linux without the Pronto patch, Linux with the Pronto patch, and then the classifier installed but empty, and finally with the classifier and an active flow.

Figure 5 shows the throughput in packets per second. The throughput of Linux peaks at just about 53.5K packets per second, and then degrades monotonically as packet size increases. The S shape of the curve is caused as the CPU and the bandwidth trade places as the bottleneck resource. Most interestingly though, is that the performance of the router in all cases is practically identical. The curves in the figure can hardly be told apart. The maximum difference occurs with the classifier installed without a declared flow, with about 3% reduction in throughput. Interestingly, in this case with a declared flow the Pronto platform outperforms Linux by about 1%, due

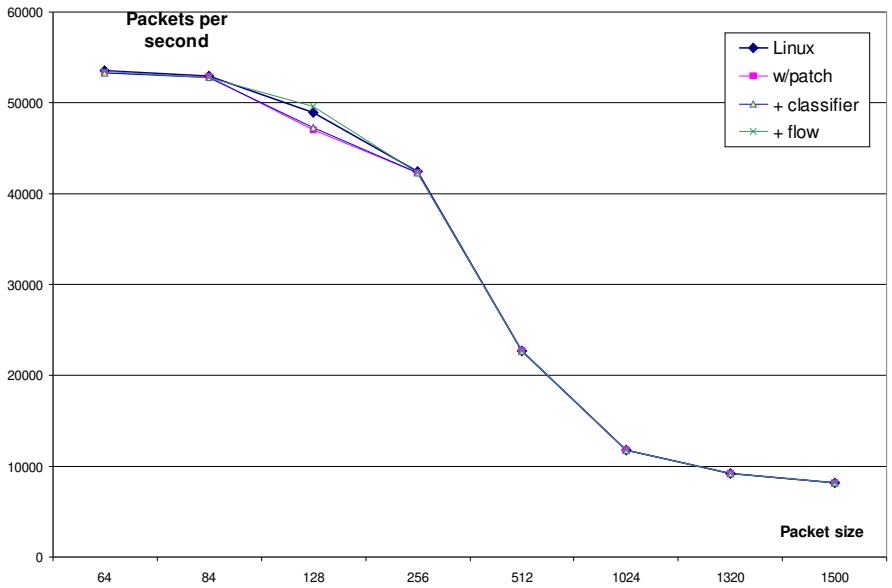


Figure 5 : Throughput in packets per second. As evident the performance with the Pronto platform is almost identical to that of Linux without it.

to slightly better implementation. For all other packet sizes, the difference is negligible or non-existent. We conclude that the Pronto platform performs identically to Linux without it.

Figure 6 shows the throughput in Mbps for various packet lengths. This figure is really the dual of Figure 5 and simply verifies that the Pronto platform does not reduce the performance compared to using Linux unchanged. The performance is pretty impressive, above 95Mbps with only 256 byte packets, and at 99Mbps at 1024 bytes. We conclude that even with moderate packet sizes we can afford some additional processing without impeding performance.

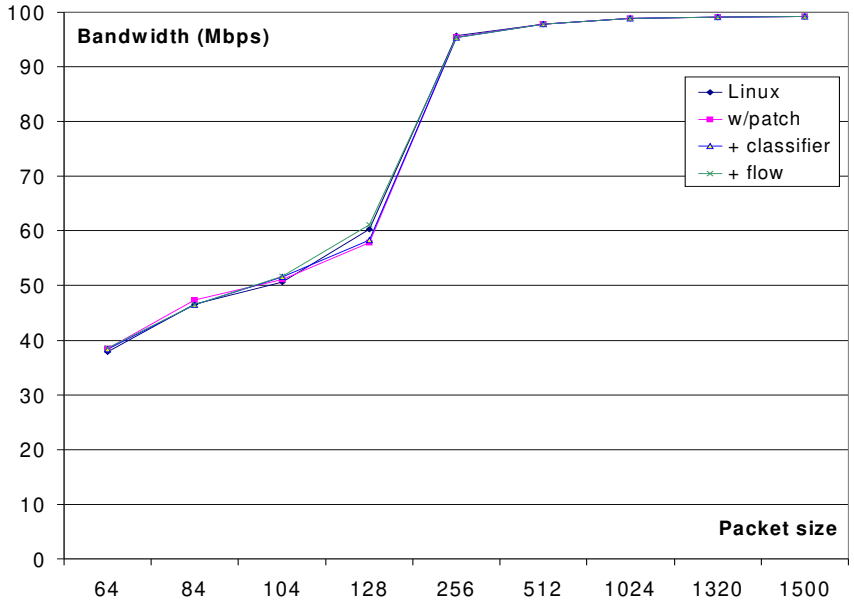


**Figure 6:** Throughput in Mbps. Even at 256 byte packets the link is filled almost to capacity, and completely at 1024 bytes.

## 8. Conclusion

We have described the Pronto platform for active and programmable networking, providing a kernel level node OS support without sacrificing the benefits of a commercial OS. The platform consists of a *set of kernel modules*; a small kernel patch for a commodity version of Linux (2.2.2 and higher), and a user level library, with a defined API to access these facilities. A key difference between our platform and other Linux based node OSes is that our changes of the kernel proper are small enough to be version independent allowing us to exploit the latest version to the Linux OS at any given time. Our performance measurements demonstrate how putting the basic facilities at the kernel level results in negligible impact on forwarding performance. The platform is highly modular with very little coupling between individual components. Using a commodity OS allows other researchers interested in higher level issues and novel service creation to exploit the platform without sacrificing the services and support of a commercial OS, significantly minimizing their platform risk.

We have experimented extensively with the Pronto platform and provided three examples of applications using it. Our performance measures show that the base performance of the Pronto platform is the same as that of Linux. The platform is developed as part of the Pronto router project, and is available from the author now, and will be publicly available within weeks.

## 9. References


1. A. Bestavros, M. E. Crovella, J. Liu, and D. Martin, "Distributed Packet Rewriting and its Application to Scalable Server Architectures," in *Proceedings of the 1998 International Conference on Network Protocols* (ICNP '98), October 1998.

2. Mohit Aron, Gaurav Banga, Vivek Pai, Michael Svendsen, Peter Druschel, Willy Zwaenepoel and Erich Nahum “Locality-aware Request Distribution in Cluster-based Network Servers,” in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII). San Jose, California, October 1998.

3. Nikos Anerousis and Gísli Hjálmtýsson, "Service Level Routing on the Internet," to appear at *Globecom'99*.

4. Elan Amir, Steven McCanne, and Randy Katz, “An Active Service Framework and its Application to Real-time Multimedia Transcoding,” *SIGCOMM '98*, Vancouver, Canada, August 1998.

5. D. Wetherall, J. Guttag, and D. L. Tennenhouse, “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols,” *IEEE OPENARCH'98*, San Francisco, CA, April 1998.

6. Gísli Hjálmtýsson and Samrat Bhattacharjee, “Control on Demand - An Efficient Approach to Router Programmability,” in *IEEE JSAC*, Vol. 17, No. 9, September 1999.

7. Gísli Hjálmtýsson and Ajay Jain, “Agent-based Approach to Service Management - Towards Service Independent Network Architecture,” in *the Proceedings of the fifth IFIP/IEEE International Symposium on Integrated Network Management - IM97*, pp. 715-729, San Diego, California, May 1997.

8. Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., “Router Plugins - A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers,” *Washington University Tech Report WUCS-98-08*, February 1998

9. Samrat Bhattacharjee, Ken Calvert and Ellen W. Zegura. “An Architecture for Active Networking,” *High Performance Networking* (HPN'97), White Plains, NY, April 1997.

10 D. Scott Alexander, Marianne Shaw, Scott M. Nettles and Jonathan M. Smith, “Active Bridging,” *Sigcomm 1997*, Cannes, France, September 1997.

11. Larry Peterson, Scott Karlin, and Kai Li, “OS Support for General-Purpose Routers,” *HotOS Workshop*, March 1999.

12. Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek, “PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems,” In *proceedings of OpenArch '99*, pp. 78-89. New York, New York. March 27, 1999.
13. The Pronto project: http://www.research.att.com/~gisli/pronto
14. Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr., “Exokernel: an operating system architecture for application-level resource management,” in the *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995, pp. 251-266.
15. AN Node OS Working Group, “NodeOS Interface Specification,” Edited by Larry Peterson, February 1999
16. D. L. Tennenhouse and D. J. Wetherall, “Towards an Active Network Architecture,” *Computer Communication Review*, 1996
17. Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, Hui Zhang, “Darwin: Resource Management for Value-Added Customizable Network Service,” *Sixth IEEE International Conference on Network Protocols* (ICNP'98), Austin, October 1998.
18. J.E. van der Merwe, S. Rooney, I.M. Leslie and S.A. Crosby "The Tempest - A Practical Framework for Network Programmability," *IEEE Network*, Vol. 12, No. 3, May/June 1998, pp.20-28.
19. Pawan Goyal and Harrick M. Vin, “Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks (Extended Version),” accepted for publication in *IEEE/ACM Transactions on Networking*. Previous version *in the Proceedings of SIGCOMM'96,* Stanford, CA August 1996.
20. V. Vaishampayan “Application of Multiple Description Codes to Image and Video Transmission over Lossy Networks,” *Proceedings of the 7th Intl. Workshop on Packet Video*, pp. 55-60, Brisbane, Australia, 18-19 March, 1996.
21. Jennifer Rexford and Don Towsley, “Smoothing variable-bit-rate video in an internetwork,” in *IEEE/ACM Transactions on Networking*, April 1999, Vol. 7, No. 2, pp. 202-215.
22. M. Yajnik, J. Kurose, D. Towsley, “Packet Loss Correlation in the MBone Multicast Network,” *IEEE Global Internet Conference*, London, November 1996.
23. S. Kasera, G. Hjálmtýsson, D. Towsley, J. Kurose, “Scalable Reliable Multicast Using Multiple Multicast Channels,” submitted *to IEEE/ACM Transactions on Networking*, Jan 1999.