A Programmable Router Architecture Supporting Control Plane Extensibility

Jun Gao   Peter Steenkiste   Eduardo Takahashi
Allan Fisher

March 2000

CMU-CS-00-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report is an expanded version of "A Programmable Router Architecture Supporting Control Plane Extensibility," *IEEE Communications Magazine*, (38)3:152-59, March 2000.

This research was sponsored by the Defense Advanced Research Projects Agency and monitored by NCCOSC RDTE Division, San Diego, CA 92152-5000, under contract N66001-96-C-8528 and by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518.

Keywords: Programmable Networks, Active Networks, Control Plane Extensibility, Mobile Code

Abstract

The Internet is evolving from an infrastructure that provides basic communication services into a more sophisticated infrastructure that supports a wide range of electronic services such as virtual reality games and rich multimedia retrieval services. However, this evolution is happening only slowly, in part because the communication infrastructure is too rigid. In this report, we present a programmable router architecture, in which the control plane functionality of the router can be extended dynamically through the use of *delegates*. Delegates can control the behavior of the router through a well defined router control interface, allowing service providers and third-party software vendors to implement customized traffic control policies or protocols. We describe Darwin, a system that implements such an architecture. We emphasize the runtime environment the system provides for delegate execution and the programming interface the system exports to support delegates. We demonstrate the advantages of using this system by presenting several delegate examples.

1 Introduction

The Internet has evolved from a basic bitway pipe to a more sophisticated infrastructure that supports electronic services. The services today are fairly primitive and are typically related to collecting information over the Web. Richer services such as high-quality videoconferencing, virtual reality games, and distributed simulation have been promised. Progress is slow in part because the infrastructure is inflexible. Routers are closed boxes that execute a restricted set of vendor software. Compute and storage servers are typically dedicated to support one type of service. An alternate architecture is to have an open infrastructure in which specific services can be installed and instantiated on demand, much like what we do on a PC today. One of the advantages of this approach is that it allows a larger community of people to develop services, which spurs innovation. We use some examples to motivate this approach.

The first class of examples addresses the customization of traffic control and management. Today, the range of traffic management options is fairly limited. While switches and routers increasingly have some support for classification and scheduling, these capabilities are often only used in simple ways, such as to filter out certain types of traffic, to do some simple prioritization of flows, or to implement standardized QoS mechanisms such as differentiated services. One could envision that users could employ these mechanisms to handle their traffic in specific ways. For example, one service provider could implement gold/silver/bronze service differentiation in a proprietary way, while another service provider implements communication services with stronger guarantees. Similarly, one could envision deploying a virtual private network(VPN) service, in which VPNs can use different traffic control policies or control protocols.

The second class of examples consists of value-added services, that is, services that require not only communication, but also data processing and access to storage. Examples include videoconferencing with video transcoding and mixing support, customized Web searching services, and application-specific multicast. While it is possible to deliver these services using a set of dedicated servers, it would be more efficient if services could deployed dynamically on servers leased on an as needed basis. This would allow the service to adapt to the demands and locations of the customers. Value-added services can also benefit from customized traffic management support. For example, a virtual reality game service provider may want to handle control, audio, and video traffic flows in different ways. This may require customized traffic control policies on the router.

As long as routers are closed boxes that are shipped with a set of standard protocols, it is unlikely that these examples will be realized. The above examples can best be supported by a programmable network infrastructure. Such a network will allow computing, storage, and communication resources to be allocated and programmed to deliver a specific service. Standards (e.g., ODBC, POSIX) exist to use storage servers (Web, file systems, databases) and compute servers. Routers (i.e., communicatin servers), however, are not programmable today. In this report we present a router architecture in which the control plane functionality of the router can be extended using *delegates*, code segments that implement customized traffic control policies or protocols. Delegates can affect how the router treats the packets belonging to a specific user through the router control interface (RCI). With this architecture, a broader community (e.g., third party software vendors or value-added service providers) can develop applications for routers.

The remainder of the report is organized as follows. We first define a network architecture in Section 2 in which network functions (e.g., QoS) can be selectively customized to meet the special needs of users. In Section 3, we describe Darwin, a specific instance of the above architecture. We focus on the runtime environment for delegates, code segments that can customize network control and management. In Section 4 we present some examples of how delegates can be applied to address a variety of resource management and traffic control problems, and we discuss security issues raised by the use of delegates in Section 5. Finally, we present related work in Section 6 and conclude the report in Section 7.

2 A Programmable Network Architecture

We first characterize the network programmability requirements and introduce the concept of a delegate. We then present a programmable network architecture that can support delegates.

2.1 Network programmability

We can distinguish between two types of operations on data flows inside the network. The first class involves manipulation of the data in the packets, such as video transcoding, compression, or encryption. Since most routers do not have significant general-purpose processing power, this type of processing will typically take place on compute servers (e.g., workstation clusters inside the network infrastructure). In the future, these tasks could also be performed on routers that have been specifically built to also perform data processing, besides the usual routing functions. The second class of operations on data flows changes how the data is forwarded but typically does not require processing or even looking at the body of packets. Examples include tunneling, rerouting, selective packet dropping, and changing the bandwidth allocation of a flow. The nature of these operations is such that they are best executed on routers or switches.

We call the code segments that perform these tasks *delegates* since they represent the owner of the data flows inside the network. Data delegates perform data processing operations and execute on compute servers or specially designed routers. Control delegates execute on routers and are involved in the control of data flows. This simple classification of delegates is somewhat artificial since some delegates may fall in between these two classes. For example, a delegate that is concerned with control may need to look at the packet body to decide how to handle packets. Similarly, a "mostly control" delegate may on rare occasions have to modify a packet. Nevertheless, the distinction is useful because the two classes of delegates impose very different requirements on the system on which they run. Control delegates require an environment that provides a rich set of mechanisms to control data flows, while data delegates must run on a platform with substantial computational power.

While nobody is likely to argue against the use of data delegates on compute servers for data processing, the need for control delegates is less obvious. One could imagine routers with fixed functionality, similar to today's routers, where users can control how their traffic flows are handled by passing parameters to the routers using a signalling protocol. The examples discussed in Section 1 provide some reasons that directly executing code (i.e., control delegates) on the routers may be a more effective way to customize traffic control and management. A first reason is that control delegates can respond much more quickly to changes in the traffic conditions; it would take an entity at the edge of the network at least one and more likely several round-trip times before it could first observe and then respond to a change in the network. Second, it seems impractical to identify all possible user requirements *a priori*, so that they can be addressed by the default router software; an architecture based on delegates is more flexible and extensible. Finally, supporting customization by extending router functionality may often be a more natural and thus less error-prone solution. For example, if a service provider wants to use a routing protocol that is optimized for its traffic, doing so from the edges of the network is likely to be unnecessarily complicated.

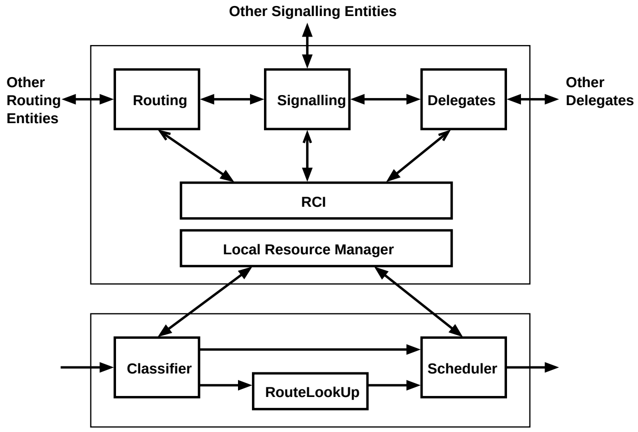


Figure 1: Node architecture.

2.2 Router architecture

Figure 1 presents a node architecture through which delegates can be added to a router. The architecture shows a control plane (top part) that executes control protocols such as routing and signalling, and a data plane (bottom part) responsible for packet forwarding. Control delegates execute in a special runtime environment that is part of the control plane. This design is motivated by both the intended use of control delegates (control and management of traffic flows) and practical design considerations (we do not want to add unnecessary complexity to the data forwarding path, where speed and simplicity are critical). On some routers it may also be possible to insert data delegates in the data forwarding plane.

Control delegates can change how traffic is handled in the data forwarding plane through the RCI. The RCI provides a set of operations on *flows*, sequences of packets with a semantic relationship defined by applications and service providers. Flows are defined on each router using a *flow spec* [5], a list of constraints that fields in the packet header must match for a packet to belong to the flow. The classifier uses the flow specs to determine what flow incoming packets belong to. Classification should take place early in the data forwarding path so that packets can be handled appropriately. Some flow-specific actions, such as tunneling and rerouting, are best executed on the input interface. Actions on the output interface are generally associated with QoS related parameters so that after classification the scheduler can schedule the packets belong to each class accordingly to meet their QoS requirements.

We can view the RCI as an instruction set that operates on flows as a basic data type. A critical design decision in the architecture is the definition of this interface (what functions should be exported to the delegates). The RCI should be broad enough to support both value-added services and network management applications. However since the RCI will be used by control delegates on routers, efficiency and security issues should also be taken into account. We will elaborate on a specific implementation of the RCI later.

Most of the components in the proposed router architecture can also be found in architectures designed to support quality of service (QoS) in the Internet. For example, the packet classification and scheduling modules are present in the Internet Engineering Task Force (IETF) integrated services model [4, 13] and the more recent differentiated services model [23], [2]. The difference lies in that this architecture providing programmability in the control plane through the use of delegates and the RCI programming interface, RCI.

3 Darwin Delegates

We give a brief overview of the Darwin [8] system and describe the Darwin RCI and delegate runtime environment.

3.1 Darwin delegate design

The Darwin project developed a set of *customizable* resource management mechanisms. Customizability allows applications and service providers to tailor resource management, and thus service quality, to fit their needs. Darwin includes three mechanisms that operate on different timescales. A resource broker, called Xena, selects resources that meet application needs using application-specified metrics to optimize resource utilization [9]. Delegates support customizable runtime resource management, as described above. Finally, Darwin uses a hierarchical packet scheduler that supports a wide range of policies and integrates per-flow QoS and link sharing in a single framework [24]. The activities of the three mechanisms are coordinated by a signalling protocol called Beagle [10].

Darwin delegates are based on the architecture outlined in the previous section, but the architecture is extended in two ways. First, in Darwin the classifier that identifies flows uses not only the standard fields in the IP and transport headers (IP addresses, port numbers, and protocol identifier), but also an optional application identifier. This allows services to define flows based on their own semantics. An example is layered video, where different layers are tagged with a different application identifier. In our current implementation, the application identifier is stored in the packet as an IP option. Other formats (e.g., the IPv6 flow ID) are possible. Every packet should normally be classified once, soon after it arrives on the router. In Darwin, for implementation convenience, we use two classification engines, one at the input port and the other at the output port. The extra classification adds negligible overhead in our system.

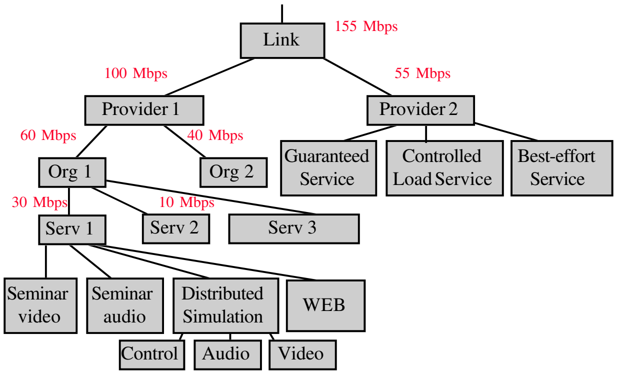


Figure 2: Example resource tree.

Second, Darwin managess resources in a hierarchical fashion. This means that the resource distribution of a link is represented by a resource tree (Figure 2), with the root representing the link, leaf nodes actual data flows, and interior nodes organizations, services or applications that control the flow or flow aggregate corresponding to their children. Resource allocation policies can be specified for both leaf and interior nodes, so both per-flow QoS and link sharing can be supported in the same framework. This can be viewed as a "divide-and-conquer" approach to resource management. The bandwidth of a link (root node) can be divided across a set of organizations (children of the root), each of which can manage its bandwidth share by constructing an appropriate subtree. Darwin uses the Hierarchical Fair Service Curve scheduler [24], which has excellent isolation properties; changes in the structure or policies of one subtree do not affect the way traffic controlled by other subtrees is handled.

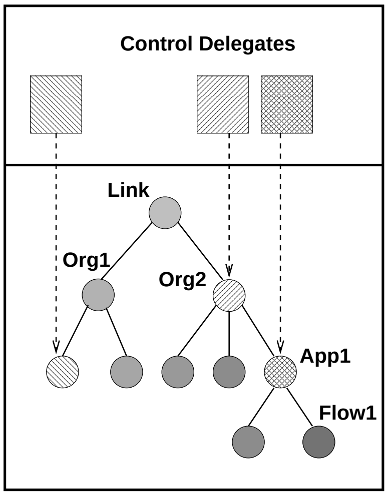


Figure 3: Pairing customization in the control and data plane.

The combination of delegates (in the control plane) and hierarchical resource management (in the data plane) provides an excellent framework for the customization of traffic control and management. We emphasized earlier that one use of delegates is to customize how a specific set of flows is handled. This is achieved by associating a delegate with a specific node in the flow hierarchy(Figure 3), so the delegate operates only on flows associated with that node and its subtree and cannot affect other flows. In other words, the hierarchical scheduler provides the isolation of network resources in the data plane and the matching hierarchical delegates provide the isolation of traffic management and control in the control plane. Note that some delegates may operate on the network as a whole; such delegates are logically associated with the root.

3.2 Router Control Interface

We describe five classes of functions that are necessary for the RCI to support a broad spectrum of delegates.

3.2.1 Flow manipulation methods

The RCI presents delegates with a flow-based programming model. This class of methods allows delegates to define and manage flows by updating the classifier data structures. For example, a flow defined with a *flow spec* (a list of header fields) can be added to the classifier through the **add_flow** call. A handle corresponding to this flow is returned. Subsequent treatment on this flow, such as specifying QoS parameters or traffic redirecting parameters, are then performed using this handle. Four methods are available for various flow manipulation.

add_flow (*interface_name*, *flow_spec*) creates a flow with *flow_spec* on the given interface and returns a *flow_id* to the caller;

delete_flow (*interface_name*, *flow_id*) removes the flow with *flow_id* from the specified interface;

modify_flow (*interface_name*, *flow_id*, *new_flow_spec*) changes the *flow spec* of the flow with *flow_id* to *new_flow_spec* and returns a new *flow_id*;

retrieve_flow_spec (*interface_name*, *flow_id*) returns the *flow spec* of the flow with *flow_id*.

3.2.2 Resource management methods

Delegates change how resources are allocated and distributed across flows by modifying states in the scheduler through the RCI. The scheduler relies on its hierarchical resource tree to schedule packets to meet each flow's QoS requirement. A node in the resource tree is identified by a node_id, and may correspond to multiple flows. Each resource tree node is assigned a set of QoS parameters to specify the service that the flows belonging to this node will receive. The precise nature of this class of functions depends on the scheduler, and the functions listed below in our implementation should be representative of most hierarchical schedulers.

create_node(*interface_name*, *parent_node_id*) creates a child node for the node with *parent_node_id* and returns a *node_id*. By default, the root node which corresponds to the total resource of an interface is created at the system start up time;

delete_node(*interface_name*,*node_id*) deletes the node with *node_id* from the resource tree;

reserve_service(*interface_name*,*node_id*, *service_parameters*) tries to reserve services specified in *service_parameters* for the node with *node_id*. This call is subject to resource admission control, and if it returns successfully, flows belonging to this node will receive the service reserved.

modify_service(*interface_name*,*node_id*, *service_parameters*) tries to change services for the node with *node_id* according to the service parameters specified in *service_parameters*. This call is also subject to resource admission control, and if it returns successfully, flows belonging to this node will receive the service defined by *service_parameters*;

add_flow_to_node(*interface_name*,*node_id*, *flow_id*) adds the flow with *flow_id* to the set of flows that belong to the node with *node_id*. This basically grants the flow with the service that the node has;

del_flow_from_node(*interface_name*,*node_id*, *flow_id*) removes the flow with *flow_id* from the set of flows that belong to the node with *node_id*. By calling this method, a flow's service can be revoked;

retrieve_tree(*interface_name*, *node_id*) returns the resource tree hierarchy rooted at the node with *node_id*;

retrieve_flows(*interface_name*, *node_id*) returns the list of *flow_id*'s associated with the node with *node_id*.

3.2.3 Flow redirecting methods

As opposed to the above class of methods, which have only "local" meaning, the RCI methods in this class have "global" meaning in that they may affect the traffic distribution in the network. Delegates use the **associate_action** method to bind certain actions with flows for flow redirecting purposes. For example, a delegate can reroute a flow's packets using a route other than the default to avoid hot spots in the network. While the resource management RCI calls typically control a router's output port functionalities, the flow redirection actions typically take place on the input port.

associate_action(*inteface_name*, *flow_id*, *action_data*) associates the proper action data required to the flow with *flow_id*.

Currently three actions are supported:

tunneling: the action data for tunneling is two encapsulation IP addresses: *encap_src_addr* and *encap_dst_addr*. The packets belonging to this flow will be prepended with an extra header using *encap_src_addr* and *encap_dst_addr* as source and destination addresses. By associating this action with a flow, a delegate can redirect the flow's packets to a selected destination;

rerouting: the action data for rerouting is *next_hop_addr*, which is the next hop address that this flow's packets will be forwarded to. This will enable the data plane of the router to skip the usual routing table look up and forward packets using the interface that corresponds to *next_hop_addr*;

dropping: drops a flow's packets at the input port.

The method **retrieve_action**(*interface_name*, *flow_id*) returns the action data that associates with the flow with *flow_id*.

Methods of direct operation on a router's routing table can be used by delegates that have superuser privileges. These methods do not operate on a per-flow basis.

get_route(*dst_addr*) retrieves next hop address from the routing table for *dst_addr*;

change_route(*dst_addr*, *next_hop_addr*) updates the next hop address in the routing table entry for *dst_addr* to *next_hop_addr*, and this requires superuser privilege.

3.2.4 Network status monitoring methods

This class of methods allows delegates to probe certain network states, e.g., queue occupancy, kernel statistics counters, error flags, etc. It can also involve posting requests for notification from the kernel of a set of specific events, such as crossing of a queue occupancy threshold and occurrence of a failure condition. By monitoring the network status using these methods, a delegate can intelligently take other proper actions in a timely fashion according to what it has observed. For example, a delegate may set up a threshold for a particular flow's queue using the **set** method, and later on can periodically query the queue length by calling the **probe** method. This can also be done by setting up an asynchronous notification with the kernel using the **request** method. This way the kernel can signal the delegate whenever the queue length is too long. As another example, a delegate can turn on the monitor mode in the kernel by using **monitor_on** method, and then retrieve packet counters in the kernel through **retrieve_data** method to compute the average bandwidth usage of a particular flow.

3.2.5 Support for delegate communication

Delegates can set up communication channels to coordinate activities with peers on other routers and interact with the application on endpoints. Messaging between them allows delegates to gather global information so that proper global actions may be taken, such as rerouting for load balancing. Interaction with applications on end-points increases the flexibility of the system, as adaptation to network events typically involves the sources. Inter-delegate communication is often application specific. We built the communication channels between delegates on top of standard communication methods.

3.2.6 Other methods

The above five classes of functions are likely to be appropriate for most routers. However, individual routers may have additional functionality on their data forwarding path and may allow delegates to control these functions. As an example, on a router that supports random early detection (RED), delegates may be able to change the thresholds used to trigger early packet drops. On a DiffServ edge router, delegates may need access to the control parameters for shapers and meters. Other routers may have algorithms to identify non-conformant flows and may allow delegates to get access to this information; we will give an example using this functionality in Section 4. Clearly, an interface standard like management information base (MIB) definitions for network management would have to be extensible, so new calls can be added as technology evolves.

3.3 Delegate implementation

Darwin delegates are based on Java and use the JDK1.2 Java virtual machine (JVM) from Sun Microsystems [16]. This environment gives us acceptable performance, portability, and safety features inherited from the language. Delegates are executed as Java threads inside the virtual machine "sandbox". A delegate is characterized by its QoS requirement (e.g., the amount of CPU and memory resources needed), and runtime environment needed.

Experiments to measure the overhead of the RCI calls from within the delegate runtime environment showed minimal difference between calls from Java delegates and calls from equivalent C programs. That is a reasonable result since RCI calls are actually implemented as native methods. The overhead of most delegates calls in an unloaded system is measured to be around 5 microseconds using machines in our testbed (see Section 4). As the system load increases (e.g., with packet forwarding) the system call latency becomes highly variable and unpredictable since our operating systems do not offer real-time guarantees.

Delegates are installed through the Darwin signalling protocol, Beagle, using a multi-step process. First, the application or service provider submits delegate Java bytecode, delegate resource and runtime requirements, together with a list of flow specs to Beagle. Second, Beagle transports this information to Beagle daemons running on the relevant routers. Third, each Beagle daemon performs local admission control for both the flows and (if necessary) any delegates. For delegates, this includes verifying that the delegate runtime environment has the required libraries to support the delegate. At this point, Beagle should also verify that the router has sufficient CPU and memory resources to accommodate the delegate, but since our environment (PC-based routers) can not explicitly manage these resources, this step is not implemented. Finally, if admission control succeeds, Beagle then sets up the flows by making appropriate calls to the classifier and scheduler, passes the bytecode to the delegate runtime environment to start up the delegate, and then passes the delegate a set of handles identifying the flows for which it is responsible. The interface that is used by Beagle to start delegates is described in more detail elsewhere [10].

Delegates written in C are also supported in our system but in a much more restrictive way. Data delegates that run on compute servers can be set up in a similar fashion.

Delegates are a very focused application of active networking [26]: they are installed asynchronously from the rest of the data traffic by a separate signalling protocol on a per-application or per-service basis; a delegate can only operate on the flows that it is associated with. As is the case with active networking in general, delegates raise significant safety and security concerns. In Section 5 we will discuss safety and security issues related to delegates in more details.

4 Examples

The Darwin system has been implemented on FreeBSD and NetBSD PC routers. Experiements were performed to test the system on a local testbed shown in Figure 4. The three routers, shown in boxes, are Pentium II 266 MHz PCs running the Darwin kernel which is built on top of FreeBSD 2.2.6. The end systems `m1` - `m9` are Digital Alpha 21064A 300 MHz workstations running Digital Unix 4.0. All links are full-duplex point-to-point Ethernet links configurable as either 100 Mbps or 10 Mbps. Unless specified, the links are configured as 100 Mbps. In this section, we present five delegate examples and show some experimental results obtained from this testbed. The first three examples demonstrate how application specific services can be added to the network through delegates to improve the quality of execution of these applications. The last two examples show that delegates can dynamically customize traffic control and management in a network.

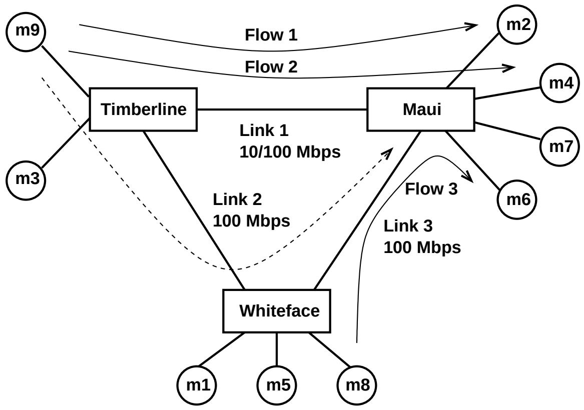


Figure 4: Darwin IP testbed topology.

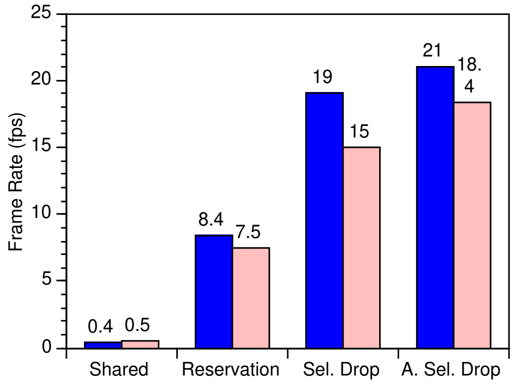


Figure 5: Video quality under four scenarios.

4.1 Selective packet dropping for MPEG video streams

MPEG video streams are very sensitive to random packet loss because of the dependencies between three different frame types: I frames (intracoded) which are self contained, P frames (predictive) which use a previous I or P frame for motion compensation and thus depend on this previous frame, and B frames (bidirectional-predictive) which use (and thus depend on) previous and subsequent I or P frames. Because of these inter-frame dependencies, losing I frames during transmission is extremely damaging, while B frames are the least critical. Under congestion, the received video quality is degraded due to packet loss but the quality can be enhanced if only B frames are dropped at the congested link instead of random frames (including crucial I frames) being dropped.

Delegates can be used to perform selective packet dropping on congested routers. Delegates are injected into routers that may potentially be congested, and these delegates are associated with MPEG flows in the data plane. In the resource hierarchy, a flow is set up to correspond to the MPEG application and three sub-flows differing from one another in the application identifier field in the flow spec are then added to differentiate the three different frame types. Packets belonging to different frame types are marked with specific application identifiers so that they will be classified into different sub-flows. Delegates monitor the queue length of the MPEG flow, and when congestion is detected (e.g., the queue length exceeds some preset threshold), delegates will instruct the data plane to drop the packets belonging to the B frame sub-flow, thus protecting I and P frames from being dropped due to queue overflow. As a result, the receiver will observe a gracefully degraded quality.

Experiments were performed on the testbed to demonstrate the effectiveness of selective packet discarding delegates. Three streams are sent over the Timberline-Maui link (bottleneck link) of the testbed: two MPEG video streams and one unconstrained UDP stream. The UDP stream causes congestion on this link. Both video sources send at a rate of 30 frames/second, and our performance metric is the rate of correctly received frames. Figure 5 compares the performance of four scenarios. In the first scenario, the video and UDP data packets are treated equally, and the random packet losses result in a very low frame rate, as expected. In the second case, the video streams share a bandwidth reservation which equals to the sum of the average video bandwidths. This improves performance, but the video streams are bursty, and random packet loss during peak transfers results in less than a third of the frames being received correctly. In the third scenario, we place a delegate on Timberline. As described earlier, the delegate monitors the length of queue used by video streams and drops the B frames when congestion is observed. Packet dropping is switched off when the queue size drops below a lower threshold. Figure 5 shows that is quite effective: the frame rate roughly doubles.

While delegates provide an elegant way of selectively dropping B frames, the same effect could be achieved by associating different priorities with different frame types. In scenario four we use a delegate to implement a more sophisticated customized drop policy. In scenario three, either all or none of the B frames are dropped. By dropping the B frames of only a subset of the video streams, we can achieve finer grain congestion control. The advantage of having a delegate control selective packet dropping is that choice of applications that should be degraded first can be customized. Scenario four in Figure 5 shows the results for a simple "time sharing" policy, where every few seconds the delegate switches the stream that has B frames dropped. This improves performance by another 10-20%. Policies that differentiate between flows could similarly be implemented.

4.2 Dynamic control of MJPEG video encoding

An alternative approach to selective frame dropping in video applications to deal with congestion is to use a video transcoder to compress, or change the level of compression, of the video stream depending on the available bandwidth. We use both control and data delegates in this example to illustrate how delegates can control compression levels for video quality optimization via flow monitoring, flow redirecting and inter-delegate communication. A control delegate is set up on the router before the bottleneck link on the route of the video application. The control delegate alters the original route that the video stream will take by first redirecting the flow to a data delegate which resides on a compute server next to the router. The data delegate functions as a transcoder in that it takes in raw video and generates MJPEG frames using different compression levels. The MJPEG frames are then fed back to the bottleneck router, and will only then be forwarded to the originally intended receiver.The control delegate monitors the bottleneck link's congestion status. When facing congestion, the control delegate directs the data delegate to use a higher compression level for less bandwidth usage. At other times, when the control delegate sees abundant bandwidth on the bottleneck link, it can prompt the data delegate to deploy a lower compression level for better video quality. This allows the video flow to opportunistically take advantage of available bandwidth.

In the experiment, an application consisting of two MJPEG video streams and two bursty data streams compete for network bandwidth with other users, modeled as an unconstrained UDP stream. All flows are directed over the 10 Mbps Timberline-Maui link. The application has 70% of the bandwidth, 20 for video and 50 for data; the remaining 30% bandwidth of the link is for the competing users. The application's data streams belong to a distributed fast Fourier transform

(FFT) computation. The data traffic is very bursty, since FFT alternates between compute phases, when there is no network usage, and communication phases, when the nodes exchange large data sets. An important property of the hierarchical link-sharing scheduler is that the video flows have priority on taking bandwidth not used by the FFT flows. This means that video quality can be improved significantly during the compute phases of the FFT, if the video can make use of the additional bandwidth.

A control delegate is placed on router Timberline and a data delegate is installed on server m9. The video traffic received by Timberline is forwarded first to m9 for data processing; the generated MJPEG frames are sent back to Timberline. Timberline then sends these frames out on the Timberline-Maui link. Figure 6 shows a screen shot of the bandwidth used by the video flows (light grey) and FFT (dark grey). During FFT bursts, bandwidth is limited (20% of the link) and video quality is low, but between FFT bursts the video can use almost 70% of the link, resulting in increased video quality. Figure 7 shows a histogram of the received frame quality. We see that the majority of frames are received with either maximum quality of 100 (received when the FFT is in its computation phase) or with the minimum quality of 0 (when FFT is in its communication phase). Frames received with other quality settings reflect the ramp up and ramp down behavior performed by the control delegate as it tracks the available bandwidth.

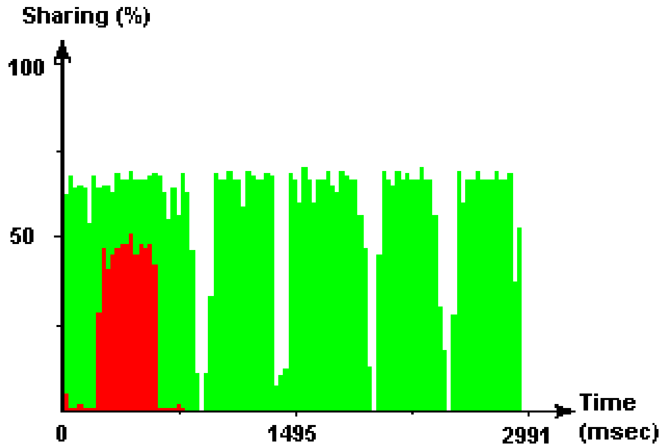


Figure 6: Bandwidth sharing between video and FFT streams.

4.3 Load-sensitive flow rerouting

Routing decisions in the Internet today are mostly load-insensitive and application-independent. While this results in simple and stable routing protocols, it can also cause inefficient use of network resources. Discovering a lightly-loaded route and using it to reroute an application's flow may significantly improve the application's performance. Similarly in a client-server scenario, there are times that one server is overloaded, and in the meantime, other servers are idle. In this case, it would make sense to have a mechanism inside the network to redirect some requests to the lightly loaded servers to achieve better overall performance.

By using the RCI, delegates can reroute a flow's packets or even redirect a flow to a different destination. We will now use a simple example to illustrate how flow rerouting can be done using multiple delegates to improve the application's network throughput.

The example application has three TCP flows m9-m2, m9-m4, and m8-m6, which are shown as

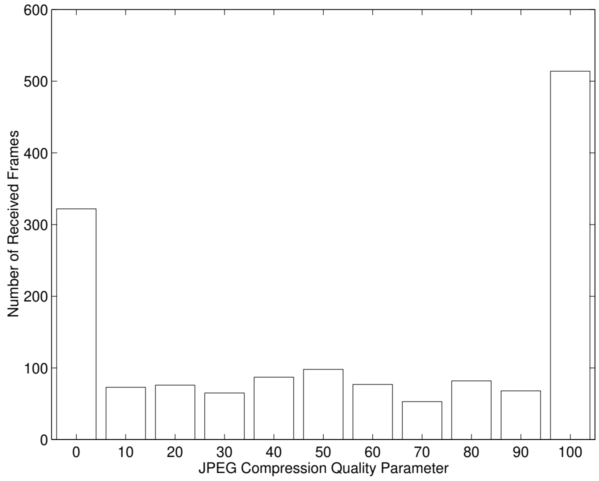


Figure 7: Distribution of received JPEG quality.

Flow 1, 2 and 3 respectively in Figure 4. The application reserves 50% of the bandwidth on Link 1, 2 and 3. Delegates are installed on each router to monitor and optimize the throughput of these flows. By default, the route for Flow 1 and Flow 2 passes routers Timberline and Maui only (the shortest path). An alternative route using all three routers is shown as a dotted line in Figure 4.

The delegate on Timberline uses the following algorithm to reroute a flow when necessary: periodically, the delegate retrieves bandwidth usage of Flow 1 and 2 from the data plane, and queries the delegate on Whiteface to get the available bandwidth for this application on Link 3. When the available bandwidth on the path consisting of Link 2 and 3 is higher than the bandwidth being used by an active flow (Flow 1 or 2), the delegate will reroute Flow 1's packets to Link 2 to avoid the competition between Flow 1 and 2. When the default route has higher available bandwidth, the delegate will then resume Timberline's default forwarding behavior.

In the experiment, the three flows are turned on and off at different times; Figure 8 shows the throughput of these flows. Initially, only Flow 1 is active; it uses Link 1, and its throughput is about 50 Mbps. When Flow 2 is turned on, Flow 1's throughput drops dramatically due to sharing. The delegate on Timberline then changes the route of Flow 1 to use Whiteface. Flow 1's throughput recovers back to about 50 Mbps. When Flow 3 is turned on, Flow 1's throughput again drops to about half, because of the competition on Link 3. When Flow 2 ends, the available bandwidth on Link 1 (50 Mbps) is higher than Flow 1's current throughput (about 25 Mbps). Flow 1 is routed with the default route, which uses Link 1. As can be seen, the throughput of Flow 1 goe s back to about 50 Mbps. In the meantime, Flow 3 receives the full reservation on Link 3 and its throughput is improved to about 50 Mbps. In summary, with rerouting, delegates help the application's flows adapt to the route that has larger available bandwidth in a timely fashion.

4.4 Selective dropping of non-adaptive flows

Applications that do not use appropriate end-to-end congestion control are an increasing problem in the Internet. These applications do not back off when there is congestion, or they back off less aggressively than users that use correct TCP implementations, and as a result, they get an unfair share of the network bandwidth. We will refer to such flows as non-conformant. In response to this problem, researchers have developed a variety of mechanisms that try to protect conformant flows from non-conformant flows. These include Fair Queueing scheduling strategies that try to

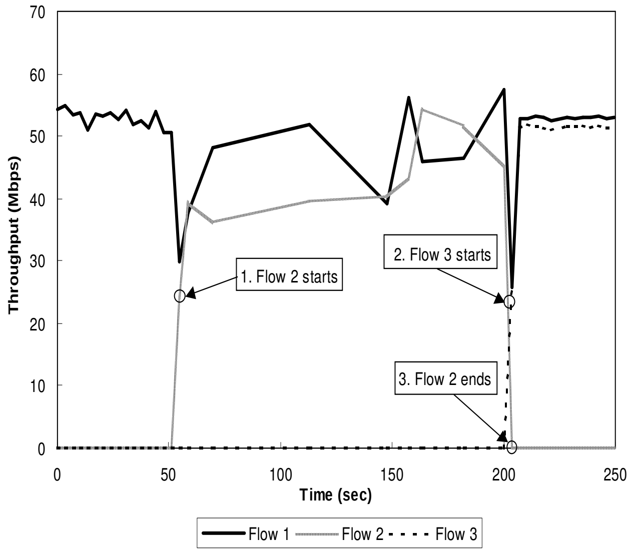


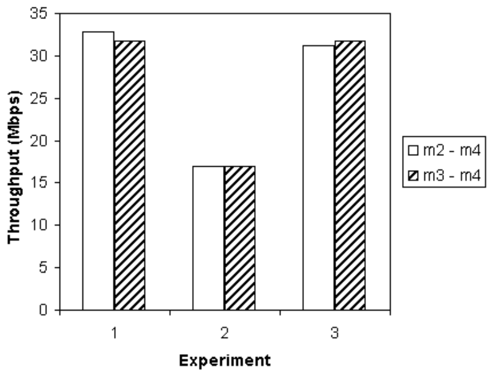
Figure 8: Load-sensitive rerouting results.

distribute bandwidth equally, and algorithms such as RED [14] and FRED [20] that, in case of congestion, try to selectively drop the packets of non-conformant flows.

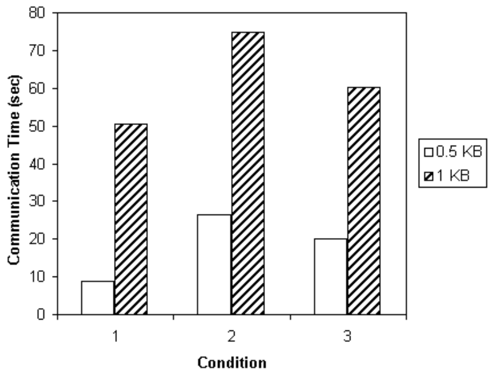
While, once deployed, these mechanisms will improve the fairness of bandwidth distribution at the bottleneck link, they address only part of the problem since they are designed to work locally. The problem is that non-conformant flows still consume (and probably waste) bandwidth upstream from the congested link. Upstream routers may not respond to the non-conformant flow, for example because they have no support for detecting non-conformant flows, or because the flow cannot be detected (e.g. because of aggregation in a core router), or because the flow appears to be conformant (e.g. does not cause congestion). This problem can be addressed by having routers propagate information on the non-conformant flows upstream along the path of those flows. This approach can also deal with denial-of-service attacks on servers: the server can report the attack to the router it is attached to, and the network can then track down, and selectively drop, that flow potentially all the way to the source.

We implemented a simple version of this solution using delegates. A delegate locally monitors the congestion status and tries to identify non-conformant flows among the flows it is responsible for. In our implementation, a flow is considered to be non-conformant if its queue is overflowing for an extended period of time; more sophisticated mechanisms would be needed in a production version of the system. Once a "bad" flow has been identified, the delegate enables selective packet dropping for the flow, and sends the flow's descriptor to a peer delegate on the upstream router. When a delegate receives a report of a "bad" flow, it verifies that the flow indeed has a high bandwidth consumption and enables selective packet dropping, if possible, and forwards the message to the upstream router. Clearly, many alternative policies could be implemented; for example, only a certain percentage of the packets could be dropped to reduce its bandwidth instead of dropping all packets as in our implementation.

We conducted two experiments to show the effectiveness of these delegates. In the first experiment, two TCP streams (m2-m4 and m3-m4) compete with a non-conformant UDP stream (m5-m4). The bottleneck link is Maui-m4. Throughputs of the TCP flows under different conditions are shown in Figure 9(a). In the first case, the UDP stream is switched off, and the two TCP flows share the link bandwidth evenly. In Experiment 2, the UDP flow is present in the background and the TCP flows' performance is greatly reduced because of the UDP flow. In Experiment 3, the delegates are running on all three routers. The UDP flow from m5 to m4 does not cause congestion on the link between Whiteface and Maui, and therefore the delegate on Whiteface does not react. However, there is congestion on the Maui-m4 link, and Timberline correctly identifies the UDP flow as non-conformant. It then informs the delegate on Whiteface through the delegate communication channel. Both routers then drop the UDP flow's packets, and as a result, the two TCP flows recover their throughputs to about the same level as in Experiment 1.



(a) TCP versus UDP



(b) FFT versus UDP

Figure 9: Conformant TCP and FFT competing with non-conformant UDP.

In the second experiment, a bursty but conformant (using TCP) distributed FFT computation competes with a non-conformant UDP stream. The FFT uses endpoints m2, m3 and m4, and the UDP stream uses nodes m5 and m4, as before. In Figure 9(b), we show how the FFT communication time is affected by the UDP flow under different conditions. We present results for two FFT data sizes, 0.5K and 1K. As a base case, in Experiment 1, the UDP flow is idle. In Experiment 2, the UDP flow is active and as a result, the FFT communication times are dramatically increased. Finally, in Experiment 3 we deploy the delegates, which correctly identify the UDP flow and drop its packets. This reduces the communication times for the FFT compared to Experiment 2.

4.5 A Virtual Private Network Service

A Virtual Private Network(VPN) [12] is an overlay network that is created within an existing physical network. It is virtual in that there is no separate physical infrastructure for this network and multiple virtual networks can run simultaneously within one network. It is private in that the data traffic belongs to one virtual network must not be observed by other virtual networks that are sharing the same infrastructure.

In a related project [19], we used Darwin as the basis for the development of a VPN service that allows per-VPN customization of control protocols and QoS support. A basic prototype has been running since October 1999. The per-VPN customizability relies on both the delegates and hierarchical resource management. Specifically, hierarchical resource management makes it possible to isolate the resources and traffic of different VPNs and to realize per-VPN QoS by having each VPN define its own subtree in the resource hierarchy. Delegates, on the other hand, allow control plane protocols specific to a VPN to be deployed dynamically.

To invoke the VPN service, users specify the topology of their VPN (e.g., virtual links and routers, bandwidth information, client networks that should be served by the VPN) and they specify the control plane protocols. The VPN service uses this information to determine what resources it needs in the network, and it then formulates a request for Beagle to set up VPN links and the control plane protocols (as delegates). One of the more difficult problems raised by this application is how to best deal with routing, given that different VPNs might use different routing protocols. We simply use a different table for each VPN in our current implementation [18]. Each VPN routing table is maintained by a routing daemon that implements RIP. The routing daemon currently coded in C is viewed as a special routing delegate. Once the VPN has been established, the user is notified and it can start sending data. Data packets originated from end-users belonging to a VPN are identified at the edge routers using our classifier. The edge router inserts a VPN ID into the packet header, and inside the network, core routers use flow classification based on the VPN ID to identify which VPN the packet belongs to. The data forwarding plane can then handle the packet appropriately, e.g., in terms of routing and QoS. IPsec is used for privacy.

5 Delegate Security

The programmable nature of an active network brings legitimate safety and security concerns. The safety issues brought to the routers by delegates include general code safety concerns, various types of denial-of-services attacks, and privacy and security concerns. For example, a malicious or badly-implemented delegate can pose various threats to the router ranging from router state corruption to machine crash. This problem is being addressed by many research groups, and solutions from a variety of runtime mechanisms (e.g. Java sandboxing) to compile time mechanisms (e.g. proof carrying code [21]). In this section, we focus on security issues related to traffic management and control. Examples of threats include unauthorized use of bandwidth allocated to other flows, and redirecting or dropping traffic belonging to other users. We explain these problems by examining three classes of increasingly more powerful delegates.

The first class of delegates can control local resource allocation only, that is, only modify the classifier and scheduler states. The primary risks are: (1) A delegate manipulates traffic flows for which it is not responsible; (2) A delegate changes the scheduler parameters for resources belonging to other users. Our solution is based on associating delegates with specific flows and nodes in the resource tree, as shown in Figure 3. When Beagle sets up flows and delegates, it provides node operating system (OS) with information about what flows and resources a delegate can control. The node OS stores this information in the form of an access control list. At runtime, for each RCI call invoked by a delegaate to manipulate flows or access resources, the node OS checks whether the call is permitted for this delegate by consulting the access control list. While a simple all-or-nothing access control mechanism is sufficient for the examples we have considered so far, it is probably useful to have finer-grained access control over the range of actions a delegate can take. Resource management operations can be subdivided into more levels, such as monitoring traffic only, modifying the QoS parameters, changing the structure of a subtree (adding, deleting nodes). This is similar to UNIX file system access control.

The second class of delegates can affect flow forwarding by changing the states in the forwarding engine. We still have the above-mentioned concerns: delegates manipulate flows that they are not theirs or change parameters associated with other flows. The solution using access control mechanisms can be used to prevent a delegate from manipulating other flows. However, a new set of problems comes into picture even when a delegate only operates on the flows it is in charge of: it can use its own flows to pose potential threats to other routers or end-hosts in the network.

For example a delegate can launch a denial-of-service attack by redirecting its own flows using tunneling to a victim server; a delegate can reroute its flow to critical links to cause congestion; or IP spoofing can occur if a delegate is allowed to add arbitrary tunnel headers to its flows.

The key insight to address these new problems is to constrain the delegate actions within the *virtual network* that corresponds to the service being deployed. The virtual network consists of the links and routers that will be used by this service. Beagle reserves link resources and installs delegates on these routers. Beagle has a global view (the list of output inferfaces on the routers, the client networks, etc.) of the virtual network for each service. To make sure that the delegates actions are within the virtual network, at setup time Beagle passes its global view to the node OS of individual routers. With this information, the node OS is then able to verify whether or not the actions to be taken by delegates conform to the global view.

So far we have assumed that all delegates are installed by Beagle, and that Beagle can be trusted to provide the appropriate access control information. In a richer delegate model, delegates can create other delegates under certain conditions. This would make it possible, for example, to deploy a signalling protocol that can create delegates as a delegate. This would allow the VPN service to be used to create hierarchies of VPNs. A similar but simpler example is that a delegate is allowed to transfer authority to another delegate. Supporting this model will require a richer specification of the delegate authorities than in the earlier models.

6 Related Work

In an active or programmable network, the functionality of the network can be extended on the fly, either through the use of active packets that carry the code that should be used to handle the packet, or by dynamically installing extensions on the routers [26]. The Darwin delegate facility is based on active extensions for performance and efficiency reasons: we do not want to pay the penalty of invoking customer code for every packet or even a very large number of packets. Instead, delegates execute out-of-band relative to the data flow and are executed asynchronously in response to specific events. As the examples in this report show, many routing functions are well-suited to this style of invocation. The drawback of this approach is that there is a higher cost associated with installing active code (signalling protocol required) than with active packets. This suggests that Darwin delegates may be less efficient than, for example, code groups in ANTS, for short-lived, "datagram-like" applications. A number of good overviews of active and programmable networking research are available in the literature [7, 25].

The Defense Advanced Research Projects Agency (DARPA)-sponsored Active Net Node OS working group defined an active node (AN) architecture [6]. The AN architecture supports an execution environment (EE) that can execute active applications (AA), which can be either active packets or extensions. The delegate runtime environment can be viewed as an EE executing in the control plane of the router. It handles "control" packets, while regular "data" packets follow the "cut through" channel, that is, they bypass the EE. While Darwin delegates follow the AN architecture, there is one difference. We have focused on the router programming interface, RCI, which is not explicitly present in the more general AN architecture.

The Active Reservation Protocol (ARP) project [3] is developing a framework for fast implementation and dynamic deployment of complex network control functions using an active network approach. Within this framework, new and modified services can be dynamically installed. A similar programming interface called the the protocol programming interface (PPI), is defined for control plane protocols to control the data forwarding path. So far ARP has focused on managing versions of traditional control protocols for routing and signaling, and not so much on support for control protocols that are customized for specific users. Also, protocols in ARP are networkwide: their actions are not restricted to specific sets of flows.

The Pronto [15] project provides a platform to support network programmability. Pronto supports a number of active networking models, including the one emphasized in Darwin: control plane programmability. One differences from Darwin is that some Pronto calls imply a stronger coupling between data and control plane. For example, Pronto achieves frame dropping by having the control plane identify the packets that should be dropped, while Darwin relies on a classifier in the data plane to identify those packets. The Darwin interface is also slightly richer, although there appears to be no reason why the additional calls could not be incorporated into Pronto.

There has recently been a lot of work as part of the Xbind [17, 28] and TINA [11, 27] efforts to define a service-oriented architecture for telecommunication networks. While these efforts have similar goals to Darwin, there are also differences. First, services envisioned by Xbind and TINA are mostly telecommunications-oriented, while Darwin and delegates target a broader set of services. Second, while the focus of both TINA and Xbind is on developing an open service framework, the focus of Darwin is on developing *mechanisms*, e.g., for resource management, that can be *customized* to meet service-specific needs.

In part as a result of efforts such as XBind, a standardization effort was started to define an industry standard for an interface for programmable networks. This work is done in the context of the IEEE P1520 working group[1]. Clearly this effort is similar to the Darwin RCI, and we hope that some of our results can feed into this standardization effort. Darwin is also looking at the broader question of how to structure and build a system that uses this interface effectively. Another related standards effort of a control interface for router is GSMP[22], but it is much more narrow in scope.

7 Conclusion

In this report we presented a programmable network architecture, in which the control plane functionality of the router can be extended using delegates, code segments that implement customized traffic control policies or protocols. Delegates affect how routers treat the packets belonging to a specific user through a well-defined programming interface, the router control interface (RCI). This open architecture offers opportunities to develop applicatins for routers to abroader community, including third-party software vendors and value-added service providers.

The node architecture was implemented in the CMU Darwin system; we describe a number of delegate examples that were executed on our testbed to demonstrate a range of applications can receive benefits via such a system. The applications include congestion control for video streaming and load balancing for client-server applications. Examples on tracking down non-conformant traffic sources and dynamically deploying VPN services gave a flavor of specific network control policies can be created using such a programmable infrastructure. While the examples do not necessarily provide the optimal, or even a complete, solution to these problems, they do illustrate that a rich set of traffic control and management services can easily be deployed through the system we built. We plan to exten our work in the directions of generalization of the architecture, performance evaluation in wide area networks and dealing with delegate security issues.

References

[1] J. Biswas, J. Huard, A. Lazar, K. Lim, S. Mahjoub, L. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein. The IEEE P1520 Standards Initiative for Programmable Network Interfaces. *IEEE Communications Magazine*, 36(10):64–70, October 1998.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service, December 1998. IETF RFC 2475.

[3] Bob Braden. Active Reservation Protocol (ARP), December 1998. Abstract at URL http://www.isi.edu/div7/ARP/.

[4] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, June 1994. Internet RFC 1633.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource Reservation Protocol (RSVP) – Version 1 Functional Specification, September 1997. IETF RFC 2205.

[6] Architectural framework for active networks, August 1998. Available over the net through URL http://www.cc.gatech.edu/projects/canes/arch/arch-0-9.ps.

[7] Andrew Campbell, Herman De Meer, Michael Kounavis, Kazulo Miki, John Vincente, and Daniel Villela. A survey of programmable networks. *Computer Communications Review*, 29(2):7–23, April 1999.

[8] Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Customizable Resource Management for Value-Added Network Services. In *Sixth International Conference on Network Protocols*, Austin, October 1998. IEEE Computer Society.

[9] Prashant Chandra, Allan Fisher, Corey Kosak, and Peter Steenkiste. Network Support for Application-Oriented Quality of Service. In *Proceedings Sixth IEEE/IFIP International Workshop on Quality of Service*, pages 187–195, Napa, May 1998. IEEE.

[10] Prashant Chandra, Allan Fisher, and Peter Steenkiste. Beagle: A resource allocation protocol for an application-aware internet. Technical Report CMU-CS-98-150, Carnegie Mellon University, August 1998.

[11] F. Dupuy, C. Nilsson, and Y. Inoue. The TINA Consortium: Toward Networking Telecommunications Information Services. *IEEE Communications Magazine*, 33(11):78–83, November 1995.

[12] P. Ferguson and G. Huston. What is a VPN? In *OPENSIG'98 Workshop on Open Signalling for ATM, Internet and Mobile Networks Workshop*, Toronto, October 1998.

[13] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[14] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[15] The pronto platform - a flexible toolkit for programming networks using a commodity operating system, October 1998. Unpublished draft.

[16] The Source for Java(TM) Techonology. http://www.javasoft.com/.

[17] A. Lazar, Koon-Seng Lim, and F. Marconcini. Realizing a foundation for programmability of atm networks with the binding architecture. *IEEE Journal on Selected Areas in Communication*, 14(7):1214–1227, September 1996.

[18] Keng Lim. Design and Implementation of a Customizable VPN Service with QoS. Master's thesis, Information Networking Institute, Carnegie Mellon University, January 2000.

[19] Keng Lim, Jun Gao, Eugene Ng, Peter Steenkiste, and Hui Zhang. Implementing Quality of Service Virtual Network Service(VNS). Carnegie Mellon University, Work in progress.

[20] Dong Lin and Robert Morris. Dynamics of Random Early Detection. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, pages 127–137, Cannes, August 1997. ACM.

[21] George Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 229–243. Usenix, October 1996.

[22] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. Ching Liaw, T. Lyon, and G. Minshall. Ipsilon's General Switch Management Protocol Specification Version 2.0, March 1998. RFC 2297.

[23] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet, July 1999. IETF RFC 2638.

[24] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, pages 249–262, Cannes, September 1997. ACM.

[25] David Tennenhouse, Jonathan Smith, David Sincoskie, David Wetherall, and Gary Minden. A survey of active networking research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[26] David Tennenhouse and David Wetherall. Towards and active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[27] Tina consortium. http://www.tinac.com.

[28] Project X-Bind. http://comet.ctr.columbia.edu/xbind.