A Programmable Router Architecture Supporting Control Plane Extensibility

Jun Gao, Peter Steenkiste, Eduardo Takahashi, and Allan Fisher, Carnegie Mellon University

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 article 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 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 with two delegate examples.

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 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 supporting 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 packet 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 be deployed dynamically on servers leased on an as needed basis. This would allow the service to adapt to the demands and locations of 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 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., communication servers), however, are not programmable today. In this article 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 article is organized as follows. We first define a programmable network architecture. We describe Darwin, a specific instance of the above architecture. We present two examples of how delegates can be applied to address a variety of resource management and traffic control problems, and discuss security issues raised by the use of delegates. Finally, we present related work and conclude the article.

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.

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). 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. Nevertheless, the distinction is important 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 signaling protocol. The examples discussed previously 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.

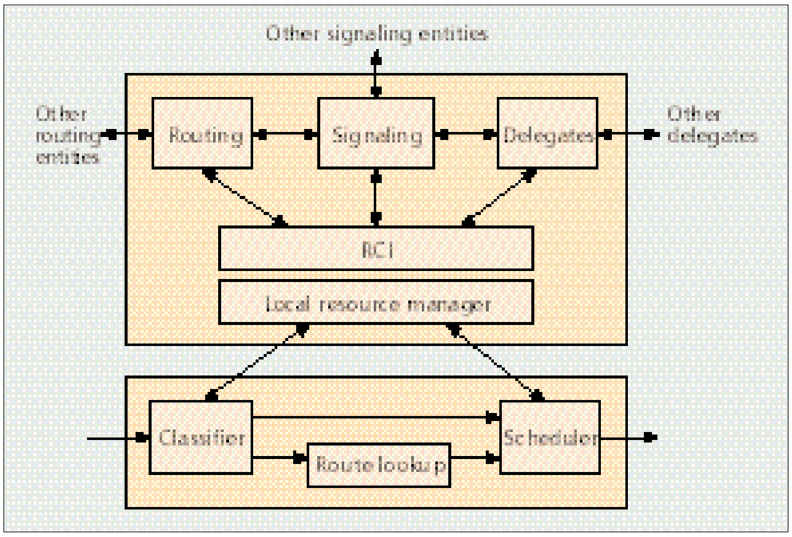


Figure 1. *Node architecture.*

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 signaling, 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 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*, 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 specifications to determine to which flow incoming packets belong.

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.

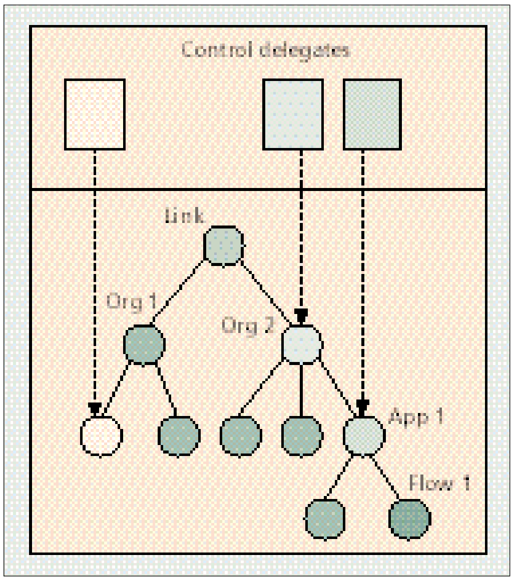


Figure 2. *Pairing customization in the control and data plane.*

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 and the more recent differentiated services model. The difference lies in this architecture providing programmability in the control plane through the use of delegates and the RCI programming interface.

Darwin Delegates

We give a brief overview of the Darwin [1] system, and describe the Darwin RCI and delegate runtime environment.

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-specific metrics to optimize resource utilization [2]. Delegates support customizable runtime resource management, as described above. Finally, Darwin uses a hierarchical packet scheduler that supports a wide range of service disciplines [3]. The activities of the three mechanisms are coordinated by a signaling protocol called Beagle [4].

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 different application identifiers. The application identifier is stored in the packet as an IP option, but other formats (e.g., the IPv6 flow ID) are also possible.

Second, Darwin manages resources in a hierarchical fashion. This means that the resource distribution of a link is represented by a resource tree (the bottom part of Fig. 2), with the root representing the link, leaf nodes actual data flows, and interior node 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 [5], 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.

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(Fig. 2), 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.

The Router Control Interface

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

- 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 specification (a list of header fields) can be added to the classifier through the `add_flow` call. A handle corresponding to this flow is returned. Operations on this flow, such as specifying QoS parameters and rerouting, are then performed using the handle. Methods for deleting and modifying flow specifications are also available.

- 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. RCI methods of this category include adding nodes to and deleting nodes from the resource tree, modifying the service parameters of a node, and retrieving the resource hierarchy. The precise nature of this class of functions depends on the scheduler, and the functions in our implementation should be representative of most hierarchical schedulers.
- 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. For example, delegates use the `reroute` method to route a flow's packets through a route other than the default to avoid hot spots in the network. Packet tunneling and selective dropping methods are also implemented. Methods of direct operation on the routing table can be used by delegates that have superuser privileges.
- Traffic monitoring and queue management methods: Delegates monitor network congestion status by examining queue lengths. The hierarchical scheduler implements a fairly sophisticated queuing discipline, and RCI exports basic queue management methods to delegates. For example, the method `probe` returns the queue size of a flow. The method `retrieve_data` enables delegates to retrieve bandwidth usage and delay data of each flow.
- 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. Interdelegate communication is often application-specific. We built the communication channels between delegates in our examples on top of standard communication 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. Clearly, an interface standard like management information base (MIB) definitions for network management would have to be extensible, so new methods can be added as technology evolves.

DELEGATE IMPLEMENTATION

Darwin delegates are based on Java and use the JDK1.2 Java virtual machine (JVM) from Sun Microsystems. 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 delegate calls in an unloaded system is measured to be around 5 μs using machines in our testbed. As the system load increases, the system call latency becomes highly variable and unpredictable since our operating systems do not offer real-time guarantees.

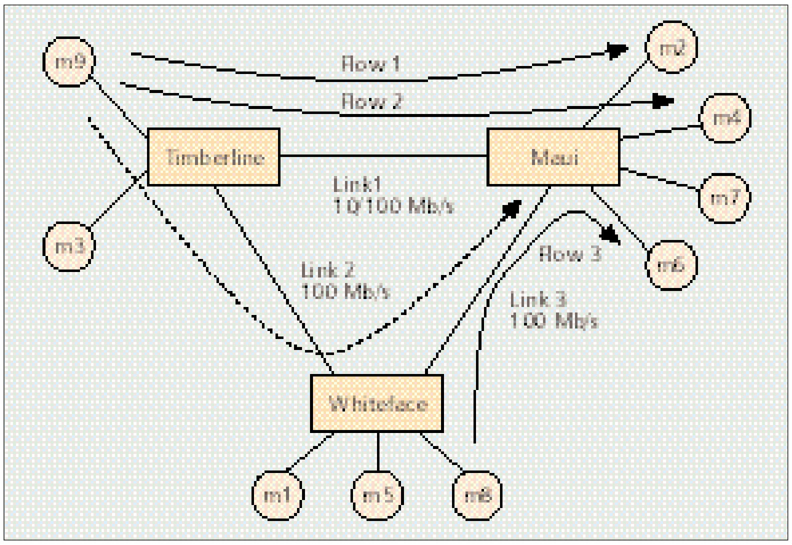
Delegates are installed through the Darwin signaling protocol, Beagle, using a multistep process. First, the application or service provider submits delegate Java bytecode, delegate resource and runtime requirements, together with a list of flow specifications to Beagle. Second, Beagle transports this 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) cannot 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 byte code 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 used by Beagle to start delegates is described in more detail elsewhere [4].

Delegates are a very focused application of active networking [5]: they are installed asynchronously from the rest of the data traffic by a separate signaling protocol on a per-application or per-service basis, and can operate only on the flows with which they are associated.

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.

EXAMPLES

The Darwin system has been implemented on FreeBSD and NetBSD PC routers. Experiments were performed to test the system on a local testbed, shown in Fig. 3. 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. 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 Mb/s or 10 Mb/s. Unless specified, the links are configured as 100 Mb/s. In this section we present two delegate examples to demonstrate how application-specific services can be added to the network through delegates to



■ **Figure 3.** *The Darwin IP testbed topology.*

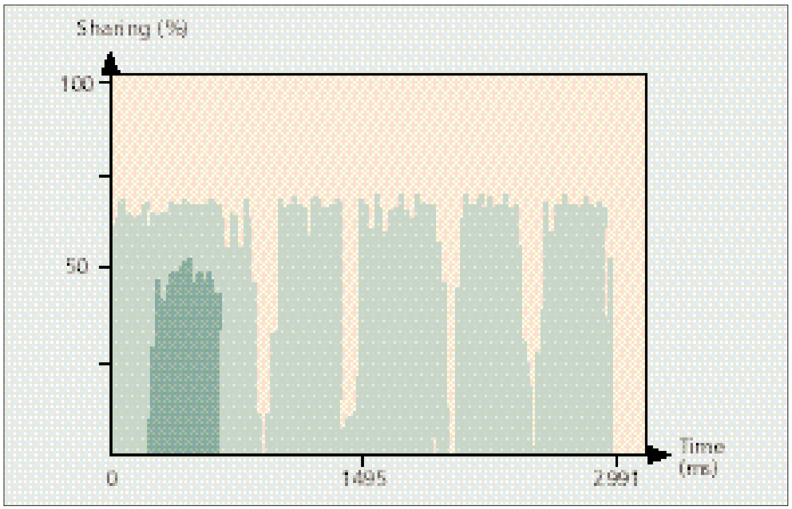
improve the quality of execution of these applications. For more examples and experimental results, refer to [6] by the same authors.

DYNAMIC CONTROL OF MJPEG VIDEO ENCODING

An approach to dealing with congestion in video applications 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 redirection, 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 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 Mb/s Timberline–Maui link. The application has 70 percent of the bandwidth, 20 for video and 50 for data; the remaining 30 percent of the link is for 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 to take 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 4 shows a screen shot of the bandwidth used by the video flows (light) and FFT (dark). During FFT bursts, bandwidth is limited (20 percent of the link) and video quality is low, but between FFT bursts the video can use almost 70 percent of the link, resulting in increased video quality. Figure 5 shows a histogram of the received frame quality. We see that the majority of frames are received with either maximum quality, 100 (received when the FFT is in its computation phase), or minimum quality, 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 4.** *Bandwidth sharing between video and FFT streams.*

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

The delegate on Timberline uses the following algorithm to reroute a flow when necessary: periodically, the delegate retrieves bandwidth usage of flows 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 links 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 flows 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; Fig. 6 shows the throughput of these flows. Initially, only flow 1 is active; it uses link 1, and its throughput is about 50 Mb/s. 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 Mb/s. 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 Mb/s) is higher than flow 1's current throughput (about 25 Mb/s). Flow 1 is routed with the default route, which uses link 1. As can be seen, the throughput of flow 1 goes back to about 50 Mb/s. In the meantime, flow 3 receives the full reservation on link 3, and its throughput is improved to about 50 Mb/s. In summary, with rerouting delegates help the application's flows adapt to the route that has larger available bandwidth in a timely fashion.

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-service attacks, and privacy and security concerns. 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.

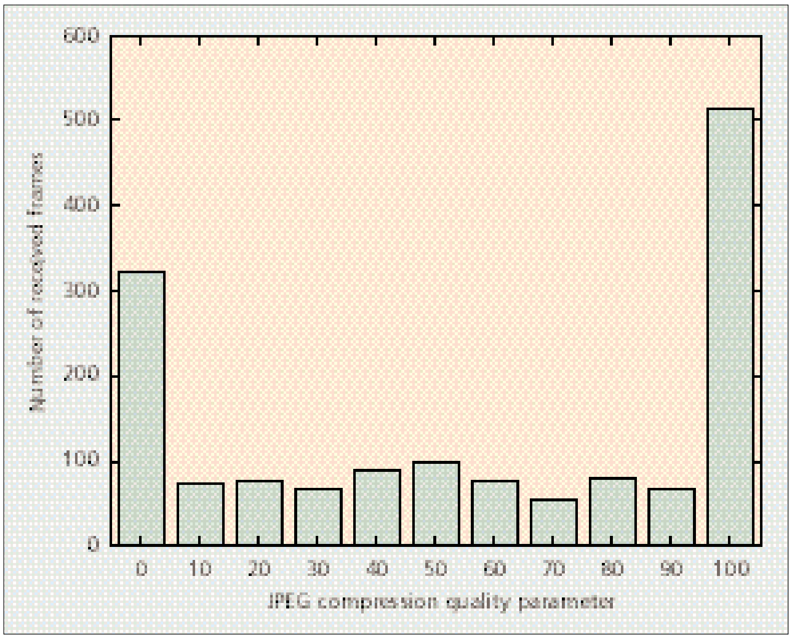


Figure 5. *Distribution of received JPEG quality.*

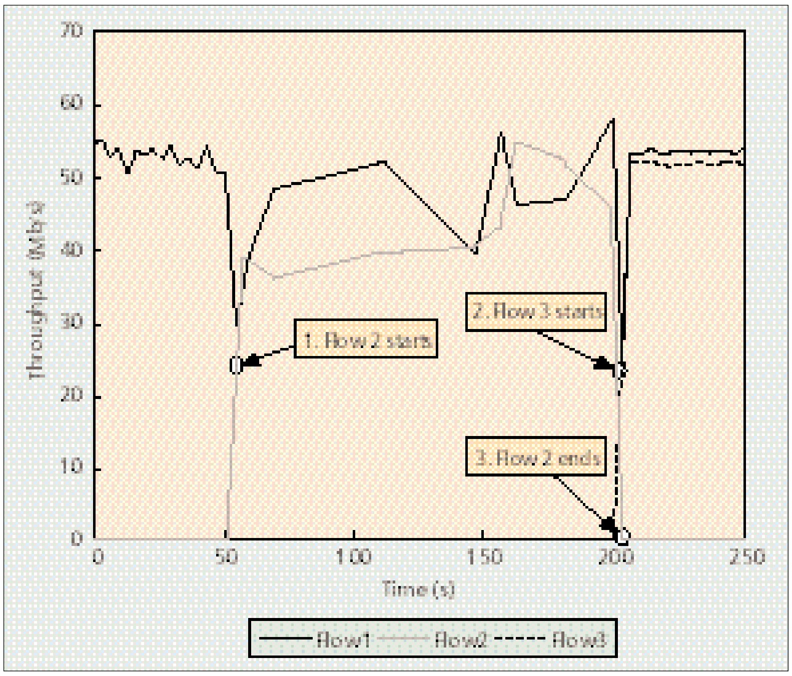


Figure 6. *Load-sensitive rerouting results.*

The first class of delegates can control local resource allocation only, that is, only modify the classifier and scheduler states. The primary risks are:

- A delegate manipulates traffic flows for which it is not responsible.
- A delegate changes 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 Fig. 2. When Beagle sets up flows and delegates, it provides the node operat-

The key insight to address these new problems is to constrain the delegate's 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.

ing 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 delegate 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 considered so far, it is 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, and 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 are not theirs or change parameters associated with other flows. The solution using access control mechanisms can prevent a delegate from manipulating other flows. However, a new set of problems comes into the 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 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 in addressing these new problems is to constrain the delegate's 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 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 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 signaling 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.

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 which should be used to handle the packet, or by dynamically installing extensions on the routers [5]. The Darwin delegate facility is based on active extensions for performance and efficiency reasons. 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 (signaling protocol required)than with active packets.

The Defense Advanced Research Projects Agency (DARPA)-sponsored Active Net Node OS working group defined an active node (AN) architecture [7]. 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. One difference is that we have focused on the router programming interface, which is not explicitly present in the more general AN architecture.

The Active Reservation Protocol (ARP) project [8] is developing a framework for fast implementation and dynamic deployment of complex network control functions using an AN approach. A similar programming interface, called the protocol programming interface (PPI), is defined for control plane protocols to control the data forwarding path. Protocols in ARP are networkwide: their actions are not restricted to specific sets of flows. The Pronto [9] project provides a platform to support network programmability. One difference from Darwin is that some Pronto calls imply a stronger coupling between data and the 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 IEEE P1520 working group [10] is working toward standardization of the interface for programmable networks. Clearly, this effort is similar to Darwin's RCI, and we hope that some of our results can feed into this process. We are also looking at the broader question of how to structure and build a system that uses this interface effectively.

Conclusion

In this article we present 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 applications for routers to a broader community, including third-party software vendors and value-added service providers.

The node architecture was implemented in the Darwin system; we describe two of the many delegate examples that were executed on our testbed to demonstrate that a range of applications can receive benefits via such a system. 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 extend our work in the directions of generalization of the architecture, performance evaluation in wide area networks, and dealing with delegate security issues.

References

[1] P. Chandra *et al.*, "Darwin: Resource Management for Value-Added Customizable Network Service," *Proc. 6th IEEE ICNP*, Austin, TX, Oct. 1998, pp. 177–88.

[2] P. Chandra *et al.*, "Network Support for Application-Oriented Quality of Service," *Proc. 6th IEEE/IFIP Int'l. Wksp. Quality of Service*, Napa, CA, May 1998, IEEE, pp. 187–95.

[3] I. Stoica, H. Zhang, and T. S. E. Ng, "A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service," *Proc. SIGCOMM '97 Symp. Commun. Architectures and Protocols*, Cannes, France, Sept. 1997, pp. 249–62.

[4] P. Chandra, A. Fisher, and P. Steenkiste, "Beagle: A Resource Allocation Protocol for an Application-Aware Internet," Tech. rep. CMU-CS-98-150, Carnegie Mellon Univ., Aug. 1998.

[5] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Comp. Commun. Rev.*, vol. 26, No. 2, Apr. 1996, pp. 5–18.

[6] E. Takahashi *et al.*, "A Programming Interface For Network Resource Management," *Proc. 1999 IEEE OPENARCH*, New York, NY, Mar. 1999, pp. 34–44.

[7] "Architectural Framework For Active Networks," Aug. 1998; available at http://www.cc.gatech.edu/projects/canes

[8] B. Braden, "Active Reservation Protocol (ARP)," Dec. 1998; abstract at http://www.isi.edu/div7/ARP

[9] G. Hjalmtysson, "The Pronto Platform — A Flexible Toolkit for Programming Networks using a Commodity Operating System," to appear, OPENARCH 2000.

[10] J. Biswas *et al.*, "The IEEE P1520 Standards Initiative for Programmable Network Interfaces," *IEEE Commun. Mag.*, vol. 36, no. 10, Oct. 1998, pp. 64–70.

Biography

JUN GAO (jun.gao@cs.cmu.edu) received B.S. degrees in engineering physics and computer science from Tsinghua University, Beijing, China, an M.S. degree in nuclear engineering from the University of Virginia, and an M.S. degree in computer science from Carnegie Mellon University, in 1995, 1997, and 1999, respectively. He is currently a third year Ph.D. student in the Computer Science Department at Carnegie Mellon University. His research interests include QoS provisioning, resource and traffic management for the Internet, and programmable and active networking.

PETER STEENKISTE (prs@cs.cmu.edu) is an associate professor in the School of Computer Science and the Department of Electrical and Computer Engineering at Carnegie Mellon University. He received a B.S. degree in electrical engineering from the University of Ghent, Belgium, in 1982, and M.S. and Ph.D. degrees in electrical engineering from Stanford University in 1983 and 1987. His research interests are in the areas of networking and distributed systems. More information can be found at http://www.cs.cmu.edu/~prs.

EDUARDO S. C. TAKAHASHI (takahasi@cs.cmu.edu) received the M.S. degree in Electrical and Computer Engineering from Carnegie Mellon University in 1999, M.S. degree in Electrical Engineering from Universidade de Sao Paulo, Brazil, in 1995, and B.S. degree in Electrical Engineering from Instituto Tecnologico de Aeronautica, Brazil, in 1989. His areas of interest include mobile, active, ubiquitous, wireless, pervasive networking, QoS provisioning, and resource management.

ALLAN L. FISHER (alf@cs.cmu.edu) received an A.B. degree at Princeton University in 1978, and a Ph.D. in computer science from Carnegie Mellon University in 1984. He has been on the faculty at Carnegie Mellon since then, conducting research on high-performance computers and networks, as well as establishing Carnegie Mellon's undergraduate computer science program. He is currently on leave while serving as president of Carnegie Technology Education, a subsidiary of the university that provides online courses and teaching support in software development to educational institutions worldwide.

The IEEE P1520 working group is working toward the standardization of the interface for programmable networks. Clearly, this effort is similar to Darwin's RCI, and we hope that some of our results can feed into this process.