Mechanisms for Dynamic Application Management of Network Resources

Eduardo Takahashi, Peter Steenkiste, Jun Gao, and Allan Fisher
Departments of Computer Science and Electrical Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Contact: Peter Steenkiste, `prs@cs.cmu.edu`

Abstract

The deployment of advanced network services such as virtual reality games, distributed simulation, and video conferencing, will require sophisticated resource management support. The reason is that the quality of the delivered service will depend both on what resources are allocated to the user, and how these resources are managed at runtime. This problem is challenging because the definition of Quality of Service (QoS) is in general user specific, so hardwired resource management mechanisms will not be sufficient. To address the runtime resource management problem, we introduce the concept of a *delegate*, a code segment that applications or service providers inject into the network to assist in the management of the network resources that are allocated to them. This approach allows users to tailor runtime resource management to best meet their specific needs. Moreover, since delegates execute inside the network, they can easily collect information on changing network conditions, and can quickly adapt the resource allocations for the flows they are responsible for.

Delegates have been implemented in the CMU Darwin system, which provides an integrated set of customizable resource management mechanisms in support of sophisticated network services. In this paper we present the design of the delegate runtime system, focusing on the programming interface that delegates use to monitor the network and modify resource use. We describe how delegates are supported in Darwin, and we show how delegates can be used to deal with a number of problems such as congestion control for video streaming, tracking down non-adaptive sources, and balancing traffic load.

1 Introduction

Advanced services networks have become a very active area of networking research in recent years. These research efforts cover a broad spectrum of topics, ranging from per-flow service definitions like IntServ [5, 14] to service frameworks like Xbind [18], but they share the overall goal of evolving the Internet service model from a basic bitway pipe model to a sophisticated infrastructure capable of supporting complex advanced services, such as intelligent caching, video/audio transcoding and mixing, and virtual reality games. These services will combine traditional communication resources (link bandwidth) with storage and computation resources. The design of such a service-oriented network poses new challenges in several areas, such as resource discovery, resource management, service composition, and security. In this paper, we focus on the resource management architecture for such a network. Resource management plays a central role because what resources are allocated to an application, and how these resources are managed, will have a strong impact on the quality of the delivered service, and the efficiency with which it is realized. In particular, we focus on mechanisms that support application-customized, runtime resource management.

Resource management activities can be classified as startup or runtime operations. Startup activities typically happen when users join or leave the network, and they are executed relatively infrequently. Examples include the allocation of a set of resources in response to a request from a new application, or changes in the provisioning of network resources to accommodate a new organization. Several groups have developed and implemented resource management architectures that support startup activities in service oriented networks. They typically rely on resource brokers [8, 22] and a signaling protocol [6, 9] to identify and allocate resources.

Runtime activities, in contrast, take place on a shorter time scale and are more incremental in nature. For example, a video streaming application may adjust its frame rate based on network feedback, or a distributed simulation may add a nodes to accommodate increased computational requirements. Traditionally, fine grain, runtime activities are driven by the endpoints, but in a service-oriented network, most of the functionality is provided *inside* the network. Moreover, future networks will deploy mechanisms to directly control network resource, e.g. low level QoS support, but using these mechanisms effectively will require knowledge about how resources are used inside the network. Both arguments suggest that restricting runtime resource management to the endpoints may not be sufficient.

In this paper we introduce a mechanism for applications and service providers to inject into the network code segments that are directly involved in or affect the resource management decisions for the traffic belonging to that user. The motivation for this approach is that advanced services and applications will have service-specific notions of Quality of Service (QoS) that would be difficult to capture in any fixed QoS framework. Therefore, service providers must be able to influence how "their" resource are managed based on their own notion of service quality, and this is most directly achieved by having them provide code that implements their policies. We call these code segments *delegates* since they represents the interests of the users inside the network. Delegates have been implemented in the CMU Darwin system [7], which provides customizable resource management for value-added services. Several delegates have been developed, dealing with problems such as congestion control for video streaming, tracking down non-adaptive sources, and balancing traffic load.

In the remainder of the paper we first motivate the concept of delegates (Section 2). We then define a general architecture for delegates and describe how delegates have been implemented in the Darwin system (Sections 3 and 4). In Section 5 we present four examples of how delegates can be applied to address a variety of resource management problems. We discuss related work in Section 6, and summarize in Section 7.

2 Motivation

An important requirement for the successful deployment of sophisticated value-added services is appropriate resource management support so that services can identify, allocate, and manage the resources they need to meet their specific quality of service goals. The networking community has developed several mechanisms, per-flow QoS and link sharing, that begin to address these needs. Research in per-flow QoS has resulted in the definition of service classes with either strict mathematical guarantees, e.g. guaranteed service [25], or weaker guarantees, e.g. controlled load [33] and differentiated service [11]. Strict guarantees are needed for some real-time applications and can simplify development of many other applications by making network behavior more predictable, but and they can be expensive to support. Weaker guarantees can be supported more efficiently, but service is less predictable, and applications will in general still have to adapt to network conditions within some service window. Link sharing[16] is similar to per-flow QoS, but resource guarantees are provided for a large set of flows belonging to an organization instead of a single flow.

These mechanisms, however, address only part of the problem. Advanced applications will have many flows that use a variety of resources in the network, and an important question is how low-level per-flow QoS mechanisms can be used to implement complex application-level definitions of service quality. In this paper, we focus on the runtime component of this problem; a resource allocation architecture that optimizes application-specific quality of service criteria is presented elsewhere [8]. Runtime resource management policies are needed in a number of situations. First, the availability of networking resources may change, forcing the application to change how it uses resources; this is especially important for applications that use best effort service or weak guarantees. Alternatively, the application may have to change its resource usage because its requirements have changed. Similarly, link sharing mechanisms will have to be controlled through resource management policies that are appropriate for user organizations.

Responsibility for adapting resource use has traditionally been pushed to the end points. One of the main motivations for this design is that it simplifies the core of the network. However, both network applications and the network itself are changing rapidly. Applications are becoming more complex (combining many end-points and flows) and sophisticated (actively involved in resource allocation and use). The network provides mechanisms for explicit resource control and is delivering more sophisticated services. As a result of these changes, having some resource management policies implemented by entities inside the network could have several advantages:

- Strategically placed entities in the network can more easily collect all the information that is needed to make resource management decisions. For example, they could monitor how all flows belonging to a user are using a congested link. An endpoint typically has information only on the flows it generates or receives.
- Entities in the network have immediate access to relevant information and can more quickly respond to changes. Adaptation policies implemented at endpoints have to deal with at least one roundtrip time worth of delay.
- Endpoints of course have to be involved in runtime adaptation. However, entities in the network can give specific feedback that may help in adapting. Without explicit feedback endpoints have to rely purely on implicit feedback, i.e. packet loss or measured delay; implicit feedback is often hard to interpret and often offers incomplete information.

Motivated by these potential benefits, Darwin provides explicit support for the implementation of application- or service-specific runtime management policies inside the network, as described in the remainder of this paper.

3 Delegate architecture

We describe the delegate runtime environment, focusing on the control API that delegates use to perform customized runtime resource management.

3.1 High level description

The most direct way of having applications and service providers involved in runtime resource management is to have them provide code that implements their adaptation policies. We use the term "delegate" for code segments that applications or service providers can inject into the network to implement customized resource management management of their data flows.

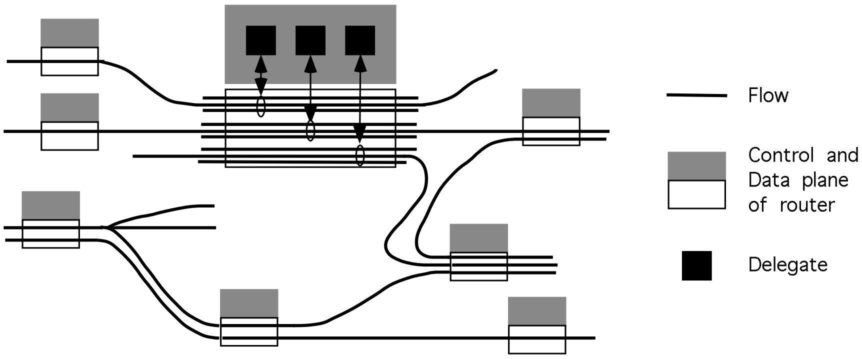


Figure 1: Delegate network model

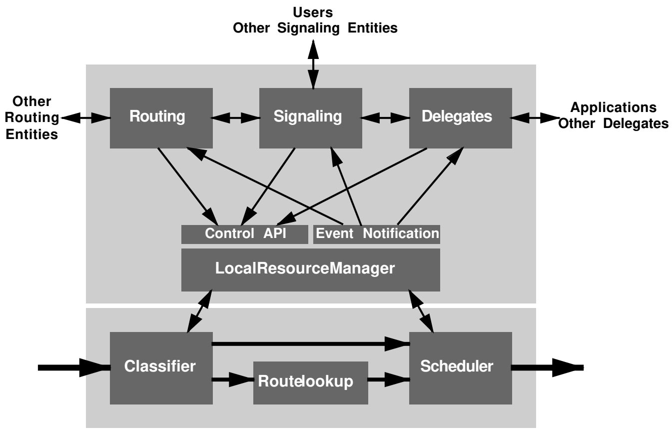


Figure 2: Node architecture

Delegates can broadly be classified as data or control delegates. Data delegates can be used to implement data manipulation operations such as video transcoding, compression, or encryption. We expect that data delegates will typically reside in compute servers (e.g. workstation clusters that are co-located with routers), although simple operations could potentially be supported on specially designed routers. Control delegates, on the other hand, perform resource management tasks that do not require processing or even looking at the body of packets, such as changing bandwidth allocations, selective packet dropping, or rerouting. Control delegates are part of the control plane, i.e. they execute on the control processor of routers or switches. While this partitioning into control and data delegates is somewhat artificial, e.g. some delegates could look at some of the data and fall in between these two classes, it adequately covers all the examples we have looked at so far. In this paper we focus on control delegates.

Control delegates execute on designated routers and can monitor the network status and affect resource management on those routers. The network model that forms the basis for the control API that delegates use is illustrated in Figure 1. The traffic in the network is viewed as a set of flows, a sequence of packets with a semantic relationship defined by applications and service providers. Flows are defined on each router using a flow spec [6], i.e. a list of constraints that fields in the packet header must match for that packet to belong to the flow. A packet classifier in the data plane of the router (shown in white in Figure 1) determines what flow each incoming packet belongs to (Figure 2). The delegates live in the control plane (shown in grey in Figure 1) and can monitor and change resource use in the data plane on a per-flow basis.

Our distinction between control and data delegates is in part driven by our desire to achieve good performance using today's routers. Complex data manipulation operations are moved to compute servers, so the router data plane remains simple: it only has to perform classification and scheduling. In contrast, activities in the control plane happen on a coarser time scale, so there is more room in the control plane for customization and intelligent decision making using control delegates. However, even on different router architectures, e.g. routers that can support expensive data manipulation, the distinction will be useful, since the two types of delegates need different APIs, and raises different performance and security concerns.

Delegates can be viewed as an application of active networks [27]: network users can add functionality to the network. It is however a very focused application: delegate operations are restricted to traffic management. As is the case with active networking in general, delegates raises significant safety and security concerns, although the focused nature of delegates simplify the problem. Delegates are in general untrusted, so the router has to ensure that they cannot corrupt state in the router or cause other problems. This can be achieved through a variety of run time mechanisms (e.g. building a "sandbox" that restricts what the delegate can access) and compile time mechanisms (e.g. proof carrying code [23]). A related issue is that of security. At setup time, the router has to make sure that the delegate is being provided by a legitimate user, and at runtime, the local resource manager has to make sure that the delegate acts only on flows that it is allowed to manage.

A critical design decision for delegates is the definition of the control interface, i.e. the API that delegates use to interact with the environment. If the API is too restrictive, it will limit the usefulness of delegates, while too much freedom can make the system less efficient. The remainder of this section focuses on the API. The definition of the API is driven by the need to support resource management and it includes functions in three categories:

- **Collecting information:** Delegates can monitor network status, waiting for events such as congestion conditions or hardware failures, or just keeping track of traffic patterns and flow distributions. Querying output queue sizes, checking for connectivity, or retrieving bandwidth usage are methods that can be used to collect information local to a delegate.

- **Resource management actions:** Delegates can change how resources are distributed across flows: splitting and merging flows, changing their resource allocation and sharing rules. For instance, a subset of a flow may be isolated through a flow split, and assigning no resources to that subset implements a selective packet dropping mechanism. Delegates can also affect routing, for example to reroute a flow inside the application's traffic for load balancing reasons. Another example is to direct a flow to a data delegate on a compute server that will, for example, perform data compression to reduce bandwidth use.

- **Delegate communication:** Delegates can send and receive messages to coordinate activities with peers on other routers and to interact with the application on endpoints. Messaging between delegates allows the system to both gather global knowledge and perform global actions, as in the case of 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.

4 Delegates in Darwin

We give a brief overview of Darwin and describe how Darwin delegates are implemented.

4.1 Darwin System Architecture

The Darwin project is developing 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 time scales. A resource broker, called Xena, selects resources that meet application needs using application-specified metrics to optimize resource use [8]. 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 [26]. The activities of the three mechanisms are coordinated by a signaling protocol called Beagle [9].

Figure 3 shows how the Darwin components work together. Applications (1) running on end-points can submit requests for service (2) to a resource broker (Xena). The resource broker identifies the resources

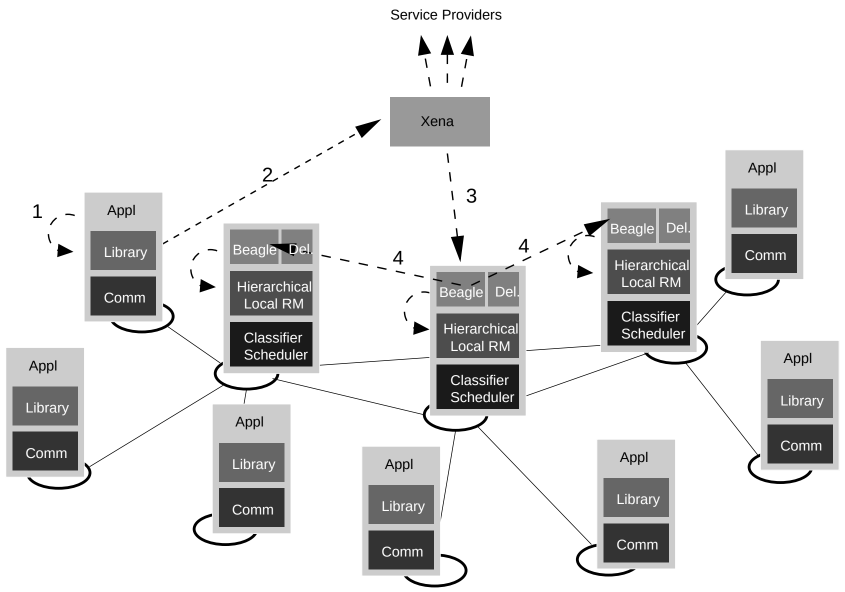


Figure 3: Darwin components

needed to satisfy the request, and passes this information (3) to a signaling protocol, Beagle (4), which allocates the resources. For each resource, Beagle interacts with a local resource manager (LRM) to acquire and set up the resource. The local resource manager modifies local state, such as that of the packet classifier and scheduler shown in the figure, so that the new application will receive the appropriate level of service. The signaling protocol also sets up delegates.

The Darwin architecture is similar in many ways to traditional resource management structures. For example, the resource management mechanisms for the Internet that have been defined in the IETF in the last few years rely on QoS routing [30, 17] (resource brokers), RSVP [35] (signaling similar to Beagle), and local resource managers that set up packet classifiers and schedulers. The more recent proposals for differentiated service [24] require similar entities. The proposals differ in the specific responsibilities of the entities. Delegates are unique to Darwin: most other resource management architectures do not address runtime resource management, and rely exclusively on endpoints to perform this function.

4.2 Darwin network model

The delegate control API is based on flow-oriented network model, although the precise definition of a flow maybe network specific. The definition of flows differs from the definition used in the IETF IntServ working group in two ways.

First, the Darwin scheduler supports hierarchical resource management. This means that the resource distribution of the link is represented by a resource tree (Figure 4), with the root representing the link, leaf nodes actual data flows, and interior nodes organizations, services or applications that control the flow or flow aggregates 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. This means that the flows observed by delegates will be organized as a hierarchy. We sometimes refer to the resources belonging to an organization, i.e. the corresponding nodes for all resource trees in the network, as a virtual network.

Second, the classifier considers not only fields in the layer 3 and 4 header, but also an application ID. This allows applications to define flows based on application semantics. In the current implementation, the application identifier is stored in the packet as an IP option. Other formats, e.g. the IPv6 flow ID, are possible.

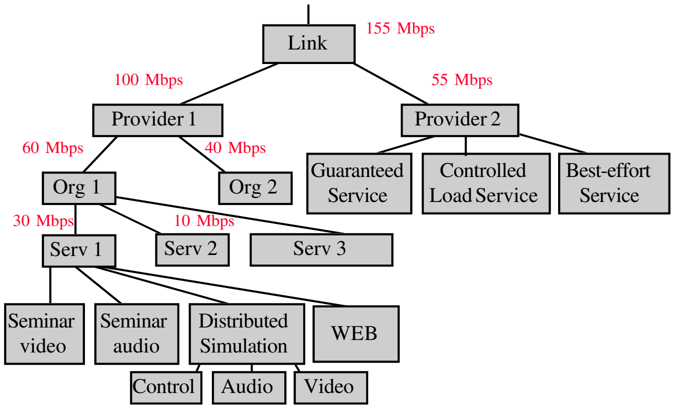


Figure 4: Example resource tree

4.3 Delegate runtime environment

Our current framework for delegates is based on Java and uses the Kaffe Java virtual machine [32], capable of just-in-time (JIT) compilation and available for many platforms. 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." Measurements of runtime performance show just around 50% slowdown when comparing the execution time of equivalent Java and C code in our environment. Currently, delegates can run with different static priority levels, although a more controlled environment with real-time execution guarantees is desirable.

The API that gives users access to resource management functions and event notification is implemented as a set of native methods that call the local resource manager, which runs in the kernel. Experiments to measure the overhead of the API 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 Java API calls are actually native methods, with a thin layer of indirection. More significant than the language employed in the runtime environment is the architectural decision of implementing it on user space. Experiments with machines in our testbed showed an overhead of around 5 microseconds in responding to system calls in an unloaded system. 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.

Table 1 presents the methods that implement the API to the packet classifier, scheduler and router. Communication was built on top of standard `java.net` classes. While this environment is sufficient for experimentation, it is not complete. It needs support for authentication and mechanisms to monitor and limit the amount of resources used by delegates. We also expect the API to evolve over time.

4.4 Delegate set up

Setting up a delegate involves a number of steps. First, we have to verify that the router has sufficient CPU and memory resources to support the delegate. The delegate may also need specific libraries or APIs that may or may not be available on all routers. Verifying that these conditions are met is a form of admission control, similar to what is done for bandwidth on link resources. Second, the delegate code has to be transferred to the router and installed. Finally, the delegate runtime environment has to be initialized. For example, the delegate has to be told what flows it is responsible for and has to be given authorization using the Beagle signaling protocol [9].

Beagle was designed to allocate network resources for structured applications. The resource needs of the application are specified in the form of a *mesh* that specifies the communication requirements plus the control and data delegates and their requirements. Delegates are characterized by their QoS requirements, runtime environment needed (for e.g., Java, Perl, VisualBasic script, etc.). Runtime type identifies the native library requirements of the delegate (for e.g., JDK 1.0.2, WinSock 2.1, etc.). In addition to delegate QoS

Methods	Description
add	Add node in scheduler hierarchy
del	Delete node from scheduler hierarchy
set	Change param. on scheduler queue
dsc_on	Activate selective discard in classifier
dsc_off	Deactivate selective discard in classifier
probe	Read scheduler queue state
reqMonitor	Request async. cong. notification
retrieve	Retrieve scheduler hierarchy
getrt	Get next hop's IP address for a specified destination
chgrt	Change the routing table entry for a specified destination
mmode_on	Turn on the monitor mode to monitor bandwidth and delay
mmode_off	Turn off the monitor mode
getdata	Retrieve bandwidth usage and delay data recorded in the kernel

Table 1: Methods available at the Delegate/Local Resource Manager API

and runtime requirements, the delegate setup message also contains a list of flow descriptors, which identify flows to be manipulated by the delegate at the execution node. At the execution node, when a delegate setup message arrives, Beagle locates the appropriate runtime environment, instantiates the delegate and passes its handles to LRM reservation state for all flows that are requested. Using these handles, the delegate can interact directly with the local resource manager to perform resource management for the flows during runtime.

5 Examples

We briefly describe the Darwin testbed, and present four examples of how delegates can be used to perform customized runtime resource management.

5.1 Testbed

The Darwin system has been implemented on a testbed of PCs, and throughout the paper we show the results of a variety of experiments conducted on a controlled testbed. The topology of the testbed is shown in Figure 5. The three routers are Pentium II 266 MHz PCs. Timberline and Whiteface are running NetBSD 1.2D and Maui is running FreeBSD 2.2.5. The end systems `m1` through `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 the experiments presented in this Section.

5.2 Selective packet dropping for MPEG video streams

MPEG video streams are very sensitive to random packet loss because of dependencies between three different frame types: I frames (intracoded) are self contained, P frames (predictive) uses a previous I or P frame for motion compensation and thus depend on this previous frame, and B frames (bidirectional-predictive) use (and thus depend on) previous and subsequent I or P frames. Because of these inter-frame dependencies, losing I frames is extremely damaging, while B frames are the least critical. In this example, we will show how delegates can be used to selectively protect the most critical frames during congestion.

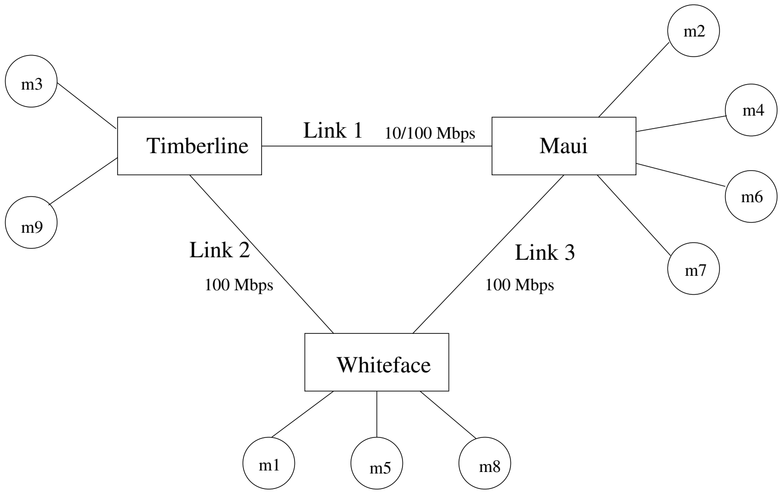


Figure 5: Darwin IP testbed topology

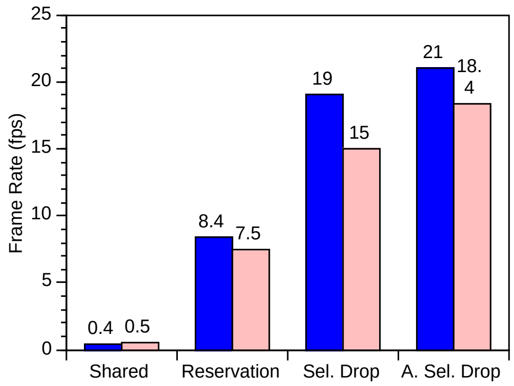


Figure 6: Video quality under four scenarios

To create congestion, we direct three flows over the Timberline-Maui link of the testbed: 2 MPEG video streams and an unconstrained UDP stream. Both video sources send at a rate of 30 frames/second, and our performance metric is the rate of correctly received frames. Figure 6 compares the performance of four scenarios. In the first scenario, the video and data packets are treated the same, 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 equal 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 also place a delegate on Timberline. The delegate monitors the length of queue used by video streams using the `probe` call. If the queue grows beyond a threshold, it instructs the packet classifier to identify and drop B frames. This is done by setting up the B frames as a separate flow using the `add` call (B frames are marked with an application-specific identifier), and then switching on selective discard for that flow using the `dsc_on` call. Packet dropping is switched off when the queue size drops below a second threshold. Figure 6 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

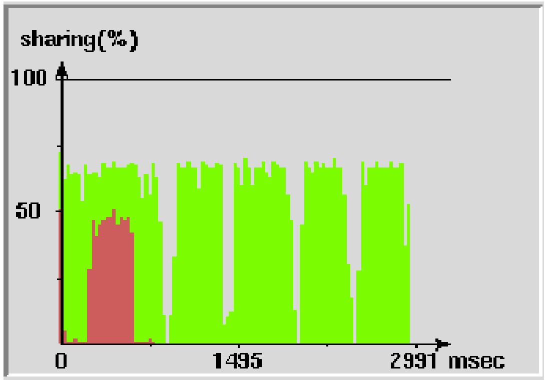


Figure 7: Bandwidth sharing between video and FFT streams.

delegates that should be degraded first can be customized. Scenario four in Figure 6 shows the results for a simple “time sharing” policy, where every a 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.

5.3 Dynamic control of MJPEG video encoding

An alternative to selective frame dropping for dealing with congestion is to use a video transcoder to compress, or change the level of compression, of the video stream. It is still possible to dynamically optimize video quality, as in the previous example, by using a control delegate to control the level of compression, as we illustrate in this example.

In this experiment, an application consisting of 2 MJPEG video streams and two bursty data streams is competing 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, and the remaining 30% is for the competing users. The application data streams belong to a distributed FFT computation. Since the FFT has alternating compute phases, when there is no communication, and communication phases, when the nodes exchange large data sets, the data traffic is very bursty. An important property of the hierarchical link-sharing scheduler used in this experiment 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.

This can be achieved by having a control delegate on Timberline monitor the FFT traffic, and adjust the level of compression of a transcoder (a data delegate) executing on Timberline. The transcoder takes in raw video and generates MJPEG. This allows the video flows to opportunistically take advantage of available bandwidth. Figure 7 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 8 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.

5.4 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

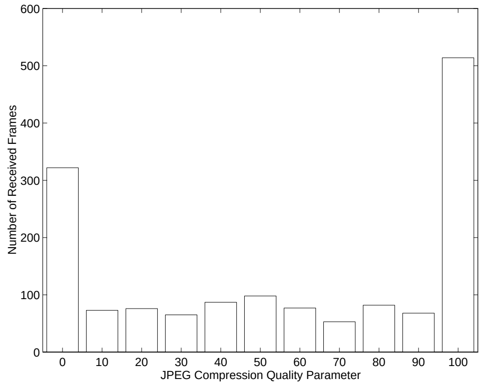


Figure 8: Distribution of received JPEG quality.

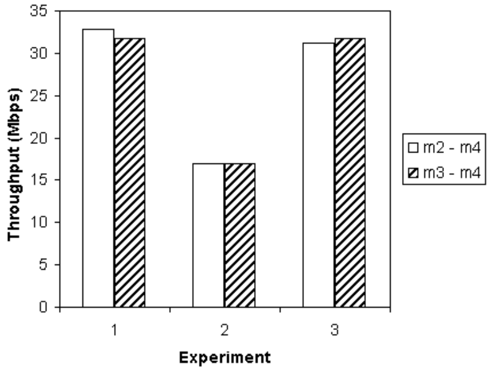
developed a variety of mechanism that try to protect conformant flows from non-conformant flows. They include Fair Queueing scheduling strategies that try to distribute bandwidth equally, and algorithms such as RED [15] and FRED [19] that, in case of congestion, try to selectively drop the packets of 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. The routers on that path may not respond to the non-conformant flow, either because they have no support for detecting non-conformant flows, or 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.

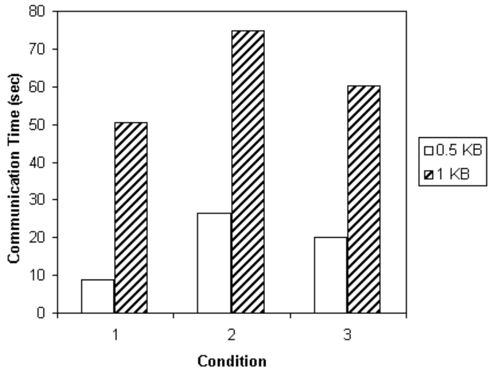
We implemented a simple version of this solution using delegates. A delegate locally observes current congestion status and tries to identify non-conformant flows. In our implementation, a flow is considered to be non-conformant if its queue is overflowing for an extended period of time; other tests could be implemented. 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 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.

In the second experiment, a bursty but conformant (using TCP) distributed FFT calculation has to compete with a non-conformant UDP stream. The FFT uses nodes 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



(a) TCP versus UDP



(b) FFT versus UDP

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

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.

5.5 Load-sensitive flow rerouting

Routing decisions in the Internet today are mostly load-insensitive and application-independent; in other words, the path taken by a packet does not depend on the load in the network or the application the packet belongs to. While this results in simple and stable routing protocols, it can also cause inefficient use of network resources. For example, in a client-server scenario, to handle multiple clients' requests, it may be necessary to have multiple servers. However, there are times that one server is overloaded by requests from clients for various reasons, and other servers are idle. In this case, it would make sense to redirect some requests to the lightly-loaded servers to achieve better overall performance. With the mechanisms described in Section 3, i.e., collecting information, communication with peer delegates and abilities of changing network resources, delegates are good candidates for this kind of task. Since delegates are considered part of an application, delegates should reroute only the flows that belong to one application.

We use a simple experiment to illustrate how delegates can balance an application's load by rerouting. The example application has multiple flows that use a virtual network that has 50% of the bandwidth on each of the three links between the routers. Flows originate from either m8 or m9 and the resource trees on Link 1, 2 and 3 are shown in Figure 10. On Link 1, Node 1 corresponds to this application and Node 2 corresponds to some other competing application. Node 3 corresponds to one specific flow of this application, m9 to m2. Node 4 corresponds to another flow of this application, m9 to m4, and is drawn in dotted lines, meaning this flow is not known to the scheduler and it will be classified to Node 1. On Link 2, Node 1 and Node 3 are the same as on Link 1, but there are no other applications that have reserved resources. On Link 3, Node 1 again corresponds to our application, and Node 2 represents some other competing application. Nodes 3 and 4 correspond to a flow from m9 to m2 and a flow from m8 to m6 respectively, and they are drawn in dotted lines, meaning that they do not have individual reservations.

The delegate on Timberline is responsible for one particular flow, m9 to m2, of this application. It knows the bandwidth usage by this flow on Links 1 and 2 by directly monitoring them, and it queries the delegate on Whiteface to get the available bandwidth for this application on Link 3. Initially, the route for flow m9 to m2 passes router Timberline and Maui only (the shortest path). Since the application has a 50% reservation, this flow gets about 50 Mbps throughput. When another flow, m9 to m4, which belongs to this application joins, they share the bandwidth reserved by the application, i.e, each gets about 25 Mbps. At this

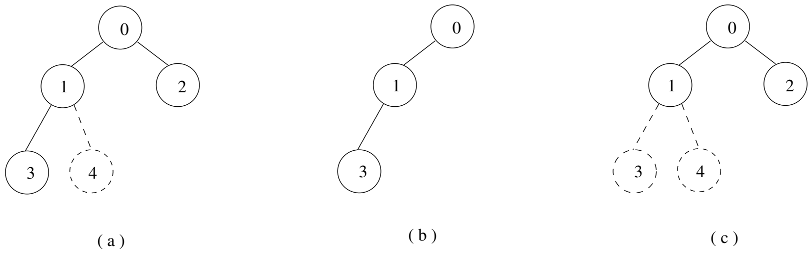


Figure 10: Resource trees used in the experiment in Section 5.5. (a) Resource tree on Link 1 (b) Resource tree on Link 2 (c) Resource tree on Link 3

time, the delegate on Timberline knows that 50 Mbps are available on Link 2 and, by querying the delegate on Whiteface, it understands that the available bandwidth for this application on Link 3 is 50 Mbps. The minimum of these two numbers is larger than what flow m9 to m2 is using, so the delegate on Timberline makes the decision to reroute flow m9 to m2 through Whiteface. Later, when another flow, m8 to m6, which also belongs to the application starts, flow m9 to m2 still goes through Whiteface until flow m9 to m4 finishes, making more bandwidth available on Link 1. At that time, the delegate changes the route for flow m9 to m2 back to its initial route.

The results are shown in Figure 11. The series annotated with “Node 3, Link 1” and “Node 3, Link 2” show the throughput of flow m9 to m2 when it passes only Link 1, and when it passes Links 2 and 3 respectively. “Node 4, Link 1” represents the throughput of flow m9 to m4 and “Node 1, Link 3” represents the throughput of flow m8 to m6. We can see that flow m9 to m2 adapts to the link that has larger available bandwidth in a timely fashion.

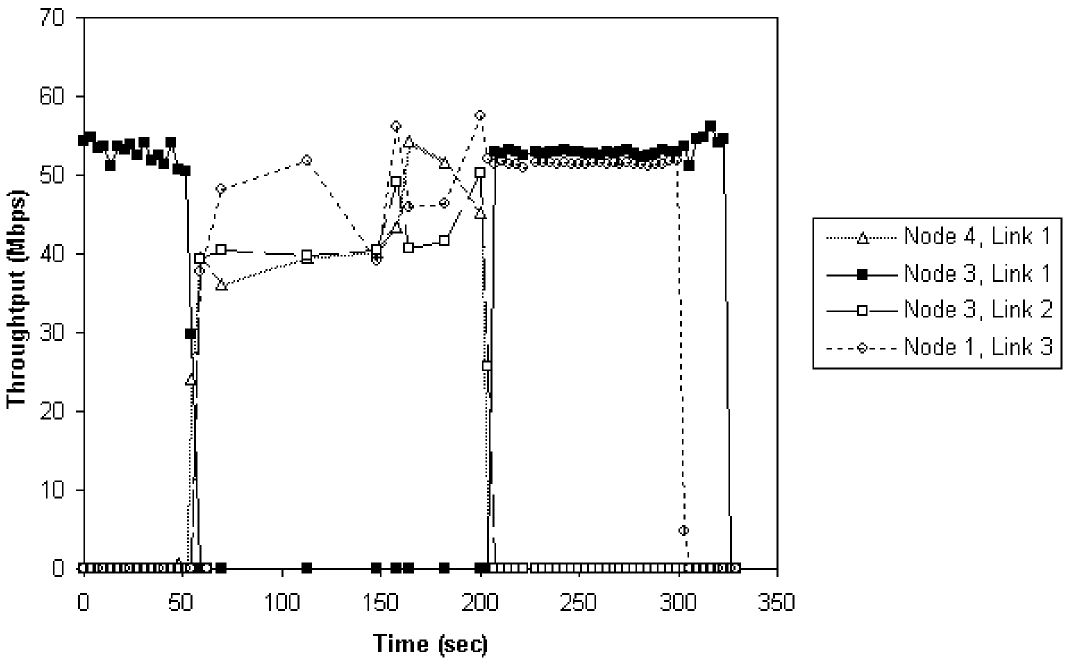


Figure 11: Load-sensitive rerouting results

6 Related work

There has recently been a lot of work as part of the Xbind [18, 34] and TINA [13, 28] efforts to define a service-oriented architecture for telecommunication networks. There are several differences between them and Darwin. First, services envisioned by Xbind and TINA are mostly telecommunications-oriented. Darwin and delegates target a broader set of services. Second, while the focus of both TINA and Xbind is on developing an open object-oriented programming model for rapid creation and deployment of services, the focus of Darwin is on developing specific resource management *mechanisms* that can be *customized* to meet service-specific needs. While Xbind and TINA have so far primarily been used as a development framework for traditional ATM and telecommunication network management mechanisms, they could potentially also be used as a basis for the development of customizable resource management mechanisms.

Over the past decade much work has gone into defining QoS models and designing associated resource management mechanisms for both ATM and IP networks [10, 14, 29]. This has resulted in specific QoS service models both for ATM [2] and IP [5, 33, 25]. This has also resulted in the development of QoS routing protocols [4, 21, 20, 30] and signaling protocols [3, 4, 35]. A closely related issue being investigated in the IP community is link sharing [16], the problem of how organizations can share network resources in a preset way, while allowing the flexibility of distributing unused bandwidth to other users. Both Darwin and the delegate mechanims build on this research. Darwin provides resource management mechanisms that support application-level notion of resource management and quality of service that are build on top of these mechanisms. Delegates, specifically, provides runtime support that complements the above startup mechanisms.

The idea of "active networks" has recently attracted a lot of attention. In an active network, packets carry code that can change the behavior of the network [27]. The delegate mechanism is closely related to the concept of active networks, in that code specific to an application can be dynamically loaded onto and executed by network nodes. Similar facilities are provided by several experimental active networks, notably ANTS[31], SwitchWare[1] and NetScript[12].

The design of the Darwin delegate facility is distinguished primarily by its concern for run-time efficiency. Rather than requiring the invocation of custom code for each packet, or even each packet in a designated flow, delegates can arrange to be invoked asynchronously by events on the network node or by a periodic clock. As the examples of this paper show, many traffic-related functions are well-suited to this style of invocation. Further, delegates are installed by the Beagle signaling facility, rather than inline with data. For short-lived, "datagram-like" applications, this suggests that Darwin delegates may be less efficient than, for example, code groups in ANTS. However, for structured applications that involve many flows, i.e. for the services that we expect to create the greatest demand for runtime management, Beagle's globally planned signaling methods can be significantly more efficient.

7 Conclusion

In this paper we introduced the concept of a *delegate*, a code segment that applications or service providers inject into the network to assist in the runtime management of the network resources that are allocated to them. Our delegate architecture was driven by two requirements. First, users should be able to tailor runtime resource management so they can optimize their notion of quality of service. Second, since delegates execute inside the network, they can quickly respond to changes in the network conditions. We described the programming interface that delegates can use to monitor the network conditions, e.g. queue status and bandwidth of the flows they are responsible for, and to modify resource use, e.g. changing reservations, selective packet dropping or rerouting.

Delegates have been implemented in the CMU Darwin system, and we described a number of delegates addressing problems such as congestion control for video streaming, tracking down non-conformant traffic sources, and balancing of traffic load. While some delegates operate in a purely local fashion, others required coordinated actions by delegates running on multiple routers. While none of the examples provides necessarily the best, or even a complete, solution to these problems, they do illustrate that our programming interface is rich enough to support a broad range of resource management actions. Future research will compare the benefits of being able to make customized resource management decisions inside the network, with the additional complexity delegates introduce.

Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency monitored by Naval Command, Control and Ocean Surveillance Center (NCCOSC) under contract number N66001-96-C-8528.

References

[1] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunder, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network*, in press 1998.

[2] ATM Forum Traffic Management Specification Version 4.0, October 1995. ATM Forum/95-0013R8.

[3] ATM User-Network Interface Specification. Version 4.0, 1996. ATM Forum document.

[4] Private Network-Network Interface Specification Version 1.0, March 1996. ATM Forum document - af-pnni-0055.000.

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

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

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

[8] 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.

[9] 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.

[10] D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings of ACM SIGCOMM'92*, pages 14–26, Baltimore, Maryland, August 1992.

[11] D. Clark and J. Wroclawski. An approach to service allocation in the internet, July 1997. Internet draft, draft-clark-diff-svc-alloc-00.txt, work in progress.

[12] S. da Silva, D. Florissi, and Y. Yemini. Composing active services in NetScript. In *DARPA Active Networks Workshop*, March 1998.

[13] F. Dupuy, C. Nilsson, and Y. Inoue. The tina consortium: Toward networking telecommunications information services. *IEEE Communications Magazine*, 33(11):78–83, November 1995.

[14] 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.

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

[16] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[17] R. Guerin, D. Williams, T. Przygienda, S. Kamat, and A. Orda. QoS Routing Mechanisms and OSPF Extensions. *IETF Internet Draft <draft-guerin-qos-routing-ospf-03.txt>*, March 1998. Work in progress.

[18] 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.

[19] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, volume 1, Cannes, August 1997. ACM.

[20] Qingming Ma and Peter Steenkiste. On path selection for traffic with bandwidth guarantees. In *Fifth IEEE International Conference on Network Protocols*, pages 191–202, Atlanta, October 1997. IEEE.

[21] Qingming Ma and Peter Steenkiste. Quality of service routing for traffic with performance guarantees. In *IFIP International Workshop on Quality of Service*, pages 115–126, New York, May 1997. IFIP.

[22] Klara Nahrstedt and Jonathan M. Smith. The QoS Broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.

[23] 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.

[24] K. Nichols, L. Zhang, and V. Jacobson. A Two-bit Differentiated Services Architecture for the Internet, November 1997. Internet draft, draft-nichols-diff-svc-arch-00.txt, Work in progress.

[25] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service, September 1997. IETF RFC 2212.

[26] 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.

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

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

[29] J. S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24(10):8–15, October 1986.

[30] Z. Wang and J. Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE JSAC*, 14(7):1288–1234, September 1996.

[31] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH '98*, April 1998.

[32] Tim Wilkinson. KAFFE - A virtual machine to run Java code. http://www.kaffe.org/.

[33] J. Wroclawski. Specification of the Controlled-Load Network Element Service, September 1997. IETF RFC 2211.

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

[35] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Communications Magazine*, 31(9):8–18, September 1993.