An Active Networking Approach to Service Customization

Peter Steenkiste[+*], Prashant Chandra[*], Jun Gao[+], Umair Shah[+]
[+]School of Computer Science
[*]Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{prs,prashant,jungao,umair}@cs.cmu.edu

Abstract

Active networking is a powerful technology to insert new functionality into the networking. In this paper we look at how active networking technology can be used to customize network services. We observe that users often want slightly different versions of network services such as multicast and network quality of service. We propose to implement these services as a base service that provides the basic service functionality and a customization code modules that allows users to customize the service. The customization module uses a service-specific API to modify service behavior. We compare this architecture with the traditional active networking architecture based on execution environments and active applications. We also present several examples of customizable network services.

1 Introduction

Active networking has been proposed as a powerful mechanism for opening up networks [36, 35, 12]. The motivation is that traditional network devices, such as routers, are typically closed systems that implement a fixed set of functions. Software that runs on a router is supplied by the router vendor and the customer's control over router functions is limited to managing built-in functions. This type of router design slows down the deployment of new services since all changes or extensions to the router functionality have to be implemented by the vendors. Active networking opens up the router architecture by supporting the execution of software from a wide range of sources, thus allowing more rapid innovation. For example, third-party software vendors can implement diverse network Quality of Service (QoS) packages for service providers with different requirements. Similarly, active routers allow the deployment of VPN service that supports customized per-VPN QoS and network management [28].

Different active networking architectures have been explored. They range from active capsules [37], where the code to process a packet is included in the packet, to active extensions, where active code is either downloaded as needed or installed in the routers using a signaling protocol [15, 17]. However, all these architectures need very similar support on the active router, as described in [10] and [30]. Roughly speaking, the active router has to provide a runtime environment for the execution of the active code, plus a mechanism to control what packets are processed by the active code. Following the terminology of [10], we will call this runtime environment the Execution Environment (or EE) and the active code the Active Application (or AA). In the last few years, a wide range of EEs have been developed [37, 18, 4, 1, 5].

The Libra project is developing network support for hierarchical network services. The goal is to be able to build sophisticated network services such as distance learning, virtual reality, and distributed interactive simulation through composition of more primitive services, such as video streaming, multicast, and transcoding. Active networking is a key technology in realizing this goal since service components can be deployed on-the-fly when and where they are needed. The Libra project is developing both a set of service components that can be used to build richer services, and a runtime infrastructure that deploys and executes composite services.

While building a set of Libra service components, we observed that different users often need different versions of the component. For example, many users need QoS support, but the details of how bandwidth is managed are user-specific. Similarly, many applications need multicast, but their specific requirements in terms of for example congestion control and reliability are different. We decided that

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

an elegant way of supporting this diversity was to break the service component into two modules: a base service that implements shared functionality, and a customization module that allows users to "fine tune" the service to meet their specific needs. Note that the base service provides a useful service in its own right. For example, in the case of a QoS service, it may support simple bandwidth management based on the Differentiated Services Expedited Forwarding model. Customization is based on a runtime environment in which users can insert customization code. In our QoS example, the customization code could monitor traffic and dynamically adjust classifier or marker parameters.

The service architecture consisting of a base service combined with a customization module is very similar to the EE/AA architecture: the base service combined with a small runtime environment can be viewed as an EE and the customization code is the AA. The difference with the more traditional EE/AA view is that in our case, most of the functionality is provided by the service EE. Note also that while people often associate traditional EEs with a specific programming language and programming model (e.g. ANTs with Java capsules, ...), in our case the EE is characterized by its functionality (base service plus supported customizations) and language is a secondary consideration.

This paper explores the use of active networking as a way of customizing network services. We first describe our network model in the next section. We then sketch the general EE architecture and discuss how service customization fits in this framework. We present several examples of service customization from the Libra project in Sections 4 through 7. Finally, we discuss related work in Section 8 and summarize in Section 9.

2 Network services model

The Libra project assumes a network model in which the network services that are delivered to end-users include elements from multiple service providers. The service providers fall in two classes. Network service providers (NSPs) provide the resources necessary to deliver the service, i.e. bandwidth on network links, computing cycles and memory on network nodes, and possibly specialized devices. This infrastructure is supposed to be programmable, i.e. new services can be deployed on-the-fly by downloading code modules into programmable network elements. Application service providers (ASPs), sometimes also called value-added service providers [13], deliver more advanced services such as distance learning or backup services to end-users. Value-added services are built by combining a set of service components and by executing them on a set of resources that is leased from an NSP. Service components can be developed internally by the ASP or can be licensed from third-party software developers.

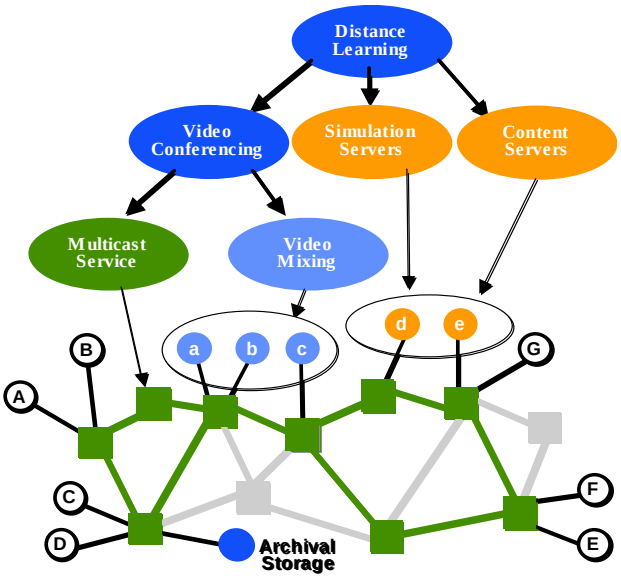

Figure 1. Service development through composition

This model assumes that network services will be developed and delivered in a competitive market. ASPs get paid by its customers (end-users and higher-level ASPs) for the value-added services they deliver; NSPs get revenue from the users of their infrastructure (ASPs and possibly end-users). Note that in practice the distinction between NSPs and ASPs may not be that clear. ASPs are likely to own some communication and computational resources, so they also play the role of NSP. Similarly, NSPs may deliver some value-added services.

As in any competitive market, service providers will want to be able to differentiate their products (services) from those of their competitors and they will want to bring services to the market quickly. The Libra project is developing two complementary technologies to meet this goal: the development of value-added services through composition of more primitive service components, and the customization of service components. Service composition allows the development of new services by combining existing components, as is illustrated for a distance learning service in Figure 1. It reduces the development effort and should allow the deploy of more sophisticated services, for example by building on components developed by specialized providers. Service customization is described in more detail in this paper.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

3 Execution environment architectures

In this section we take a closer look at the different types of EEs that have been developed. We then introduce service EEs as a new class of EEs and discuss how customizable services can be deployed.

3.1 Classes of EEs

While there is general EE architecture that defines what support is needed to host an AA, there are several ways of deploying an EE in a network. For the purpose of our discussion, we will distinguish between two classes of EEs.

The first class of EEs is characterized by the fact that its primary purpose is to process the packets that flow through AAs hosted by the EE. Examples include ANTs and the PLAN/Switchlets. Per-packet processing range from relative simple local operations such as compression or error correction (e.g. [29]) all the way to complex AAs that in effect implement virtual routers or bridges [2]. One property of such EEs is that the AAs that they host have miminal interact with the rest of the router infrastructure. Their primary interaction with the rest of the router is the exchange of packets, although they may occasionally also collect status information or negotiate with the Node OS for resources. We will call such EEs *overlay EEs* since they typically add network functionality that is quite separate from that of the hosting network infrastructure.

The second class of EEs is characterized by the fact that the AAs they host modify the behavior of the router indirectly. For example, they may update routing tables or change the parameters of packet schedulers. Any packets handled directly by such AAs are typically control packets that may trigger actions by the AA. Examples of such EEs include the Darwin QoS delegates runtime environment [23, 34] and ASP [4]. A key design parameter for such EEs is the API they offer to their AAs [31, 34]. It determines what actions the AAs can take. We will call such EEs *control EEs* since their AAs typically control router functionality.

Figure 2 illustrates the difference between overlay and control EEs. The thick arrows represent the primary dataflow while the thin dashed arrows indicate control operations. APIs are represented by horizontal dashed lines.

Not surprisingly, control EEs will typically be implemented in the control plane of the router. Overlay EEs, on the other hand, are logically part of the data plane, although how they are executed depends on how expensive the AA processing is and on the architecture of the router. On high-end active routers, overlay EEs could execute on dedicated processing resources on the router port cards, e.g. plugins [18], while on low end platforms, slow path data processing may share resources with the control plane.

3.2 Customization

While building a set of service components for the Libra project, we observed that many components share the following property. While most of the service functionality is generic and is required by all users, some service features can be supported in many different ways. Users want to choose how these features are supported, since it has a big impact on the end-to-end service properties. Let us look at some examples:

- QoS: Many services can benefit from QoS support, for example so they can deliver more predictable services to end-users. However, the details of how, for example, reserved bandwidth should be managed will be different.

- Transcoding: Many users need to be able to translate video format or reduce video resolution. However, the precise formats needed or the required video resolution will differ and may change over time.

- Multicast: Basic multicast support, e.g. delivering a packet to many receivers, is useful for many applications. However, properties such as required reliability of data delivery and practical methods for doing congestion control will be different.

While it is of course possible to implement the service component as a series of AAs, each implementing a slightly different version of the service, this is inefficient since EEs may have to run many copies of very similar code. A more elegant solution is to break the service into two parts: a base service that implements shared functionality, and a customization code module that allows users to "fine tune" the service. The customization module is often very simple, since it only has to extend or modify the existing base functionality. This is a natural solution and this approach is commonly used on end-nodes. For example, many end-user applications such as text editors or spreadsheets provide ways of extending or customizing the capabilities of the application through programs or macros.

The overall structure of a customizable service component is very similar to the EE/AA architecture. The base service combined with a small runtime environment can be viewed as an EE and the customization code is the AA. There are also some significant differences. First, in a customizable service component, most of the functionality is provided by the service EE. In contrast, most traditional EEs primarily provide support for AA execution, e.g. language support, downloading and possibly caching of AA code modules, and installing of AAs. Another difference is that the function performed by the customization code is very focused and specific to the service being customized.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

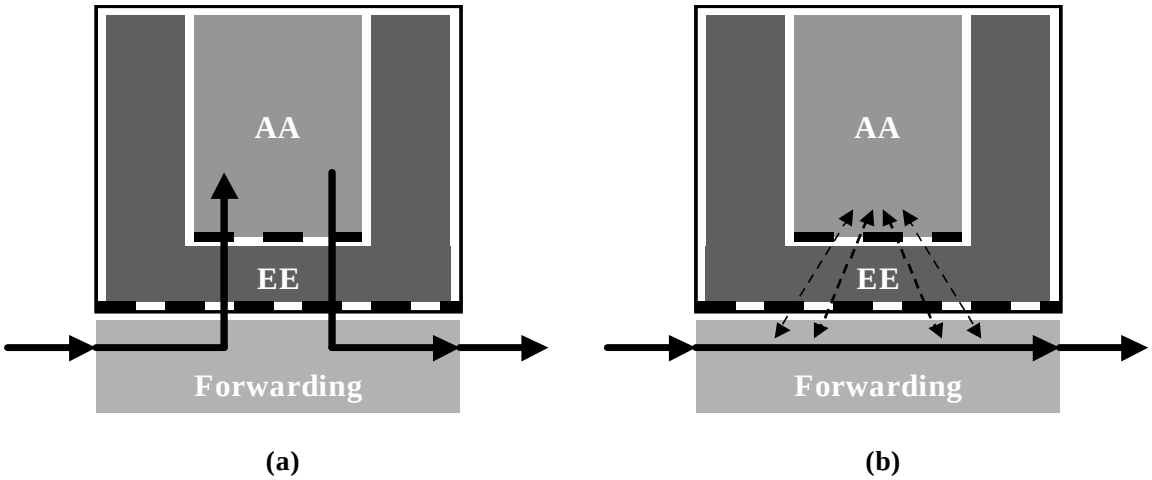

Figure 2. Classes of EEs: (a) overlay EE and (b) control EE

This second difference provides the motivation for including the base service functionality as part of the EE. Viewing just the runtime environment as the EE makes little sense since it has no useful function without the presence of the base functionality of the service. Note that customization applies to both overlay and control EEs, as is illustrated in Figure 3.

A customizable EE has two APIs (dashed lines in Figure 3. First, it has the API that it implements for its AAs. As we discussed above, we expect that API to be fairly narrow and highly service specific, and we provide some examples later in the paper. The second API defines how the EE interacts with the rest of the router. This API will typically correspond to the Node OS API and it is much more flexible and powerful than the first API. This difference in APIs suggests that installing a new customization AA will typically be a very lightweight operation, while installing a new customizable service is more heavy weight. This is not unlike the difference between installing an AA and an EE in, for example, the ABone. Installing a new EE requires special privileges, while any user can install an AA.

While we have introduced the use of customization EEs as a pragmatic solution to the problem of supporting services that differ only in certain features, this approach turns out to have another significant advantage. The API that the customization EE offers to its AAs is typically very focused and simple. This simplifies the security challenges that come with the flexibility of active networking. Specifically, checking the correctness of AA calls is typically simple and very lightweight. We will illustrate this point when discussing examples later in this paper.

Using our definition of customization EE, a number of other active networking projects have developed EEs that are very similar to our customization design. An example is the Concast effort [11]: the base service is incast (the reverse of multicast), while the specific way of merging messages from the leaves to the root can be customized. We discuss related work in more detail in Section 8.

3.3 Deploying customizable services

The main characteristic of a customization EE is that it allows users to extend or control functionality that itself is active, i.e. it is installed when it is needed. Comparing Figures 2 and 3 also shows that the distinction between customization EEs and more traditional EEs will sometimes be vague. Specifically, some Node OSes may provide functions that are provided outside the Node OS on other platforms. When extending or controlling this functionality through AAs, we end up with the architecture of either Figure 2 or Figure 3.

Figure 3 also raises the issue of how the base service component is installed. An alternative to deploying the customizable service as an EE is to deploy it as an AA in a more general EE, as is suggested in Figure 4. Deploying the base service dynamically as an AA is more in the spirit of active networking. We have however not explored this option for any of the Libra services because the interactions between the customizable service and the router are often complex, making the design of the EE in Figure 4(b) and its API very difficult. Note that Figure 4(b) corresponds to a 2-level hierarchy of AAs. It is not clear that hierarchies deeper than two would ever make sense.

Which of the two options in Figure 4 is more attractive is a deployment question. The EE option is easier to implement since there are fewer interfaces and software modules. However, it will typically require a stronger trust relationship between the entity installing the EE (typically, a service provider) and the entity owning the router. The AA option

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

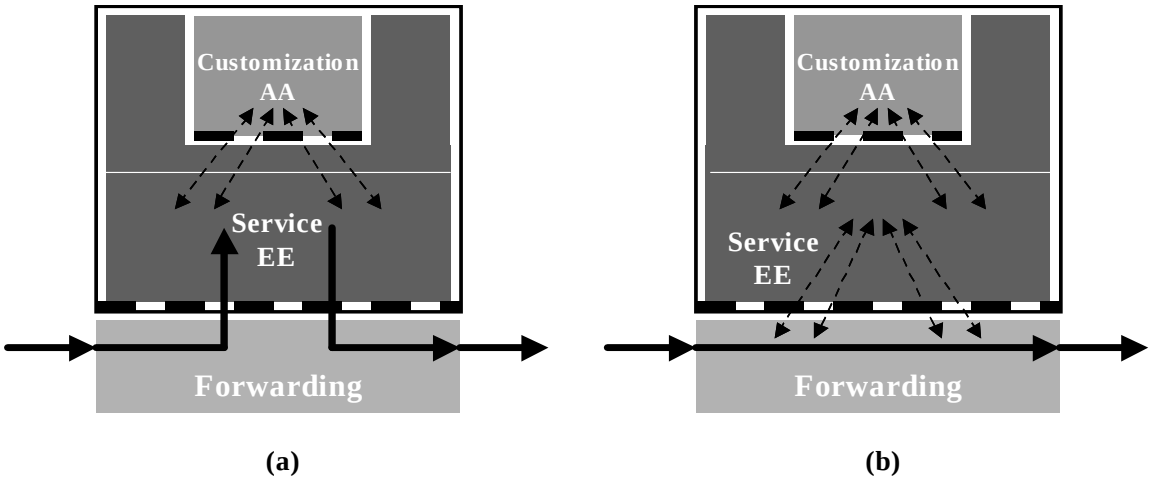

Figure 3. Active Service Customization for (a) overlay EEs and (b) control EEs

is more dynamic and flexibly. However, it will be harder to realize since more general EEs have to be available in the infrastructure.

3.4 Functionality versus language

The initial research in developing EEs often had a language focus. There are probably two reasons for this. First, opening up routers by executing third-party code raises significant security and safety questions, and the programming language is the first line of defense against malicious or buggy code. This explains the strong emphasis on using safe languages, e.g. Java [37], or CAML and PLAN [1]. A second reason is that initial active networking projects often explored the use of capsules, where clearly the EE selected has to match the language used for the capsule code.

However, given a better understanding of the safety and security issues, and given an extension-based approach to active networking, we believe that a focus on AA functionality is needed. The language is a complementary and in many cases a secondary concern. In fact, in a successful application of active networking, one should expect the same packet to be handled by AAs written in different language. Envision an experiment where active networking is used to do pre-standard testing of a new IP feature. Two groups, using different EEs may want to do interoperability testing by having a single flow of packets handled by their respective AAs implementing the new feature on different routers. In fact, one can envision AA repositories that store for each "functional AA" different code modules that different EEs.

In the following four sections, we provide some examples that illustrate the use of active networking for service customization. For each example, we will motive the need for service customization, describe our implementation and the customization API, and discuss security considerations.

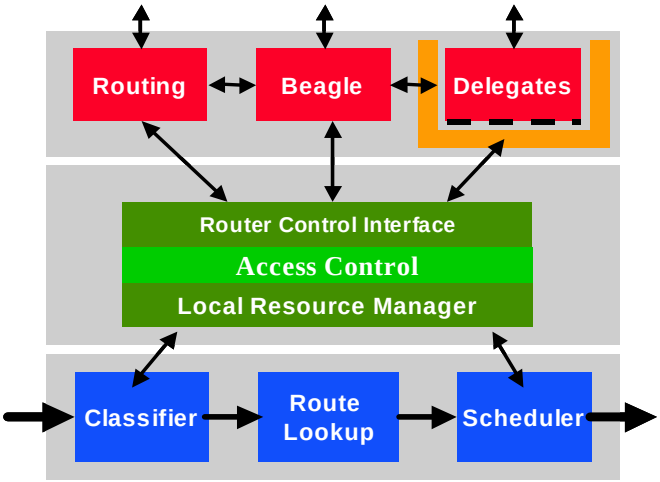

Figure 5. The Darwin delegate EE

4 QoS delegates

4.1 Motivation

The Darwin system provides router support for "application-specific" network QoS. Darwin consists of a set of data plane mechanisms (classifiers, packet schedulers) plus support in the control plane to set up and manage the data plane. Users (e.g. an ASP) can extend the control plane using "delegates", active code segments that can monitor the QoS of the user's flows and can change the QoS behavior if needed by adjusting the classifier and scheduler parameters. The Darwin architecture is shown in Figure 5.

The motivation for making the management of QoS customizable is that the QoS requirements can be very diverse. Applications such as distance learning, distributed simula-

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

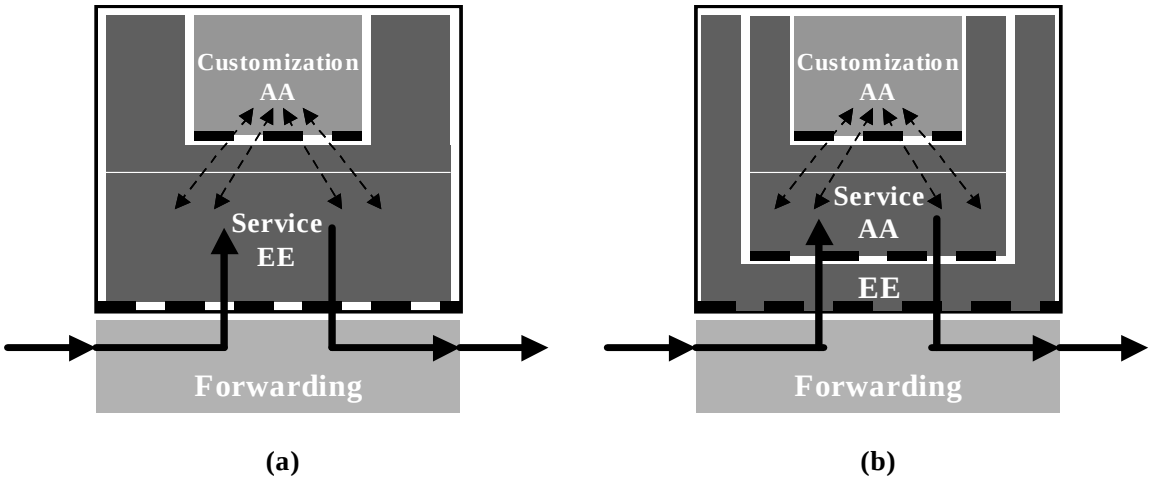

Figure 4. Deployment of a base overlay service: (a) as a service EE or (b) as an AA in a more general EE

tion, and visualization are quite different in terms of how bandwidth, delay, or loss-sensitive their flows are. In Darwin, an NSP can allocate a certain amount of bandwidth to an ASP, which can then manage that bandwidth as it sees fit using delegates. The delegates can be developed by the provider of the distributed application, or by third-party software vendors who specialize in network Quality of Service packages for service providers with diverse requirements. Another Darwin application is the deployment of customizable VPN services [28].

The Darwin architecture roughly follows the customizable services architecture of Figure 3. Delegates use an API called the Router Control Interface (RCI), shown by the dashed line, to monitor and control the QoS of their flows. Darwin is implemented in the context of a FreeBSD-based router and the delegate EE is implemented by a combination of user level and kernel software.

4.2 API

The RCI consists of a set of methods that support a rich set of operations on flows, where a flow is defined as a sequence of packets that belong together, as defined by a *flow spec* [9, 14]. RCI methods fall in three categories [24, 23]. The first category of methods enables delegates to manipulate flows by updating the classifier data structures. For example, a delegate can identify a new flow by providing a flow spec that characterizes the packets in the flow. The delegate can then apply further processing to this flow.

The second category of methods deals with the quality of service flows receive, i.e., it controls packet scheduling. Darwin uses a hierarchical scheduler [33], which means that the bandwidth distribution across flows is controlled using a resource tree. The RCI methods allow delegates to modify the resource trees for the output links (e.g., make, modify, or terminate bandwidth reservations) and to associate flows with nodes in the tree to provide bandwidth guarantees to a class of traffic.

The third category of RCI methods deal with more general packet processing or monitoring operations. For example, delegates can send and receive packets to coordinate their actions with other delegates or the user. Delegates can also collect traffic-related information about their flows, e.g. average bandwidth used, queue sizes, ...

4.3 Security and safety

While the RCI allows ASPs to manage their own traffic, it also opens the door for abuse. An ASP could for example steal bandwidth from a competing ASP. To prevent such abuse, we must limit a delegate's access to dataplane resources. Experience with traditional operating systems shows that access control lists (ACLs) are a simple and light weight mechanism for controlling access to the resources manages by an operating system. An ACL consists of three parts: a principal, an object and a permission string that defines what permission this principal has on this object. The principals in this context are delegates, the objects represent the abstract resources to be protected, and the permission bits define the possible operations on these objects. When a delegate has a permission bit set for an object, it is allowed to perform that operation.

In this section, we present the design of an ACL that is appropriate for the RCI. Given the operations supported by the RCI, the two resources that must be protected are link bandwidth and user data traffic.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

4.3.1 Protecting Bandwidth

Darwin uses a hierarchical packet scheduler since it supports both link sharing and finer grain bandwidth management. In particular, a hierarchical scheduler represents the division and sharing of bandwidth on a link in the form of a hierarchical resource tree. Each node in the resource tree corresponds to a portion of the bandwidth allocation and the subtree rooted at that node specifies how that bandwidth slice should be further partitioned. Hierarchical scheduling can for example be used to distribute the bandwidth of a link across a set of organizations, where within each organization, bandwidth can be further distributed across departments or applications. The hierarchical fair service curve (HFSC) scheduler [33] used in Darwin has the attractive property that bandwidth allocation decisions made in one subtree of the resource tree do not affect the QoS properties of traffic flows using other subtrees, i.e., there is good isolation between the subtrees, allowing them to be managed independently.

Delegates perform bandwidth management by manipulating the tree structure through RCI methods. For example, a delegate can reserve bandwidth by adding a new node to the tree via the call **create_node** *(parent_node_id, QoS_parameters)*. It is possible that a malicious delegate can steal bandwidth by adding nodes to a part of the tree that belongs to other users. It can also read, change or remove reservations made by others through manipulation of the resource tree.

Given the properties of hierarchical resource management, it is very natural to have the access control for bandwidth be based on the resource tree (see Figure 6). The node in the resource tree represents bandwidth so we use it as the object in the ACL. In order to execute bandwidth management decisions, a delegate must have the appropriate permission bit set for the corresponding node in the resource tree.

By examining the RCI methods related to resource nodes, we construct the following set of access rights to control delegate operations on nodes.

- create (c): The (c) permission on a node allows a delegate to create new nodes rooted at this node. This means that the delegate can sub-divide the bandwidth that this node represents.
- modify (m): The (m) permission on a node allows a delegate to modify the bandwidth distribution across its children.
- delete (d): The (d) permission on a node allows a delegate to delete child nodes.
- retrieve (r): The (r) permission on a node allows a delegate to retrieve the subtree structure rooted at this node.

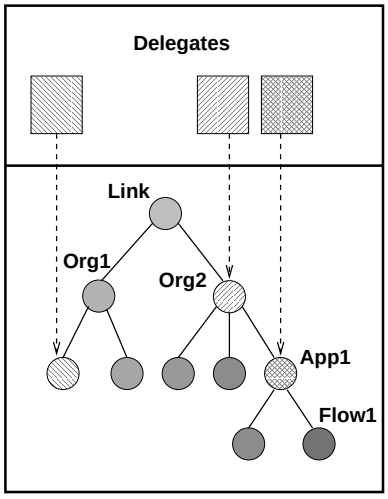

Figure 6. The association of delegates and resource nodes

- monitor (n): The (n) permission on a node allows a delegate to monitor bandwidth usage of the node.
- use (u): The (u) permission allows the delegate to use the resource represented by this node to provide QoS to traffic, as described below.

For example, when a delegate has the (m) or (d) rights for a node, but not the (c) rights, the delegate can redistribute bandwidth among a node's children, but it cannot add new child nodes. Also, when a delegate has only (n) rights for a node, it can monitor the bandwidth utilized by the flows allocated to that node, but it cannot make any changes to the bandwidth allocation.

4.3.2 Protecting Traffic

As described above, Darwin provides QoS based on flows, a stream of packets that match a packet filter. By using the RCI primitives, delegates in Darwin can add filters to define flows and can assign flows to resource nodes to receive bandwidth guarantees. The RCI also supports a number of non-QoS operations on flows, e.g. flow redirection [23, 34]. Without proper control over which traffic can be accessed by which delegates, a malicious or faulty delegate can disrupt the traffic of other users. For example, it can define flow corresponding to the traffic of another user and then associate with a node in the resource tree that has no bandwidth associated with it. That is likely to result in a lot of packet loss. To prevent such security violations, a router must (1) constrain what kind of filters a delegate can install to control what traffic it can control, and (2) control the

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

services that can be applied to the traffic that a delegate is responsible for.

The ACL that is used to limits access to traffic is based on a "filter envelop". The filter envelop restricts what filters a delegate can install and what operations it can perform on them. A traffic envelop consists of one or more *envelop-spec* entries, where each envelop-spec entry has the following format:

```
[ traffic-spec, permission] ,
```

where traffic-spec specifies a class of filters that can be defined, and permission lists the permission bits for the traffic defined by those filters.

The traffic-spec consists of the following fields: Source IP address, Source IP address mask, Destination IP address, Destination IP address mask, Source port, Destination port, Protocol ID, and Application ID (IP option). Each of these fields defines a range of values that are allowed in a packet filter. For example, the source port could be restricted to the range `[ 81, 2000 ]`. The IP address and mask together define the range of possible IP addresses that can be used by a filter. For example: `[ 128.2.0.0, 255.255.0.0 ]` defines an IP subnet address.

When a delegate creates a new filter, the router checks whether this delegate is allowed to install such a filter based on the filter envelop that is associated with the delegate. A filter is defined by a *filter-spec*, which consists of the same set of fields as in a traffic-spec, except that the port numbers, protocol ID and application ID can only be specified as a single number instead of a set of numbers. A delegate can only create a filter if the filter-spec is within the scope one of the traffic-spec entries of the delegate's filter envelop, which means that all the filter spec's fields are inside the range specified in the traffic spec. For example, the source/destination IP address of the filter-spec must match the corresponding address specified in the traffic-spec using the longest prefix matching algorithm, e.g., [128.2.205.111 255.255.255.255] matches [128.2.0.0 255.255.0.0].

The second component of each envelop-spec in the filter envelop is the permission field. It specifies what operations a delegate can perform on a traffic flow defined by a filter within the envelop-spec's scope. For QoS-related operations, the delegate must hold the (q) permission. With the (q) permission, the delegate can assign the flow corresponding to this filter to a node in the resource tree, i.e. it will get the bandwidth guarantees associated with that node. Recall also that in order to use a resource, the delegate must have the (u) permission on that node in the resource tree, so two access checks are required to provide QoS, one regarding the traffic involved and one regarding the bandwidth allocated.

4.4 Design and implementation

The RCI is implemented as a Java library that delegates can call. This library forwards the RCI call to the LRM (Figure 5), which is implemented as a user-level process. The RCI library includes the delegate identifier in the request it forwards to the LRM. The Access Control Manager (ACM) is implemented as part of the LRM, and it intercepts each RCI call made by delegates. The ACM uses the delegate ID and the interface that the delegate is operating on to find the proper policy file for access checking. We also need a mechanism to manage the ACLs. This is done by the Router Security Manager, which sets up or modifies ACLs based on a policy maintained by an external policy manager, as is described in [22].

5 Temporal sharing in Beagle

5.1 Motivation

Traditional flow-based signaling protocols allocate resources for each flow independently. This is based on the underlying assumption that each flow in the network is independent of all other flows in terms of its resource utilization. However, most services with multiple flows exhibit temporal relationships in the way their flows utilize the resources allocated to them. In such cases, these "related" flows can share the same set of resources over time. We call this type of behavior *temporal sharing* and define it as *the sharing of resources among multiple flows with temporally interleaved resource usage*. We will call a group of flows that share resources a flow group. Temporal sharing forms a middle ground between independent flow-based allocation and periodic renegotiation by combining the low signaling overhead and predictable behavior of independent flow based allocation, with savings in resource consumption obtained using periodic renegotiation.

Temporal sharing was first introduced in the original RSVP design paper [38]. RSVP introduced the notion of resource reservation styles that allowed different senders to a multicast group to share the same set of resources. A subset of the styles introduced in the original paper is supported in the RSVP specification [9]. Temporal sharing has also been studied in the context of other signaling protocols like Tenet-2 [26] and ST2+ [19]. Although these signaling protocols represent an important first step in exploiting temporal sharing, the "one size fits all" approach they take limits their usefulness. The support they provide is mostly suited for conference style applications, i.e. applications where multiple sources can send data but where as a result of the structure of the application, only one sender can be active at a time.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

However, many other temporal sharing models exist. One example is Virtual Private Networks (VPNs) based on the *hose model* [20]. According to the hose-model, in a VPN with N sites, each site i is connected by an access link of bandwidth h_i called a "hose". Therefore, a hose limits the amount of traffic generated or received by the site to a value less than the traffic received or generated by the other sites. Another example is broadcast application with picture-in-picture capabilities. Such an application allows receivers to view one high quality video channel and optionally a second reduced-quality channel, so the bandwidth that has to be allocated to a receiver is limited to one high quality and one reduced quality flow even though many channels can potentially be selected. While it is possible to define a general representation for temporal sharing, it is complicated and calculating the bandwidth requirements for a set of flows based on the general representation is NP-complete [17, 16].

However, practical cases of temporal sharing can typically be supported much more efficiently. For example, calculating the resource requirements for the above VPN example can be done $O(n)$ time. This argues that instead of building a signaling protocol that handles temporal sharing in a general but expensive way, or that only handles certain cases of temporal sharing, instead, temporal sharing support in signaling protocols must be designed to be *extensible*. This will allow service providers to define and implement new sharing behaviors without having to modify the signaling protocol. This is based on the observation that temporal sharing is an optimization that closely depends on the behavior of the service and is therefore best performed using service-specific knowledge. In the remainder of this section we describe how the signaling protocol Beagle provides extensible support for temporal sharing.

5.2 Design

A flow setup in Beagle is based on the standard *three-way handshake* mechanism realized by the exchange of three messages (SETUP_REQUEST, SETUP_RESPONSE and SETUP_CONFIRM) between neighboring routers along the path of the flow. The SETUP_REQUEST message carries information about the traffic carried by the flow and the QoS requirements for that flow. The Beagle entity at each router along the path processes this information, allocates resources required by the flow and forwards it to the next hop. In addition to this basic flow information, a SETUP_REQUEST message may also carry temporal sharing information if the flow is part of a flow group. This information is carried in the form of a `TemporalSharing` object, and it specifies the type of temporal sharing, the specific group that the flow belongs to, and parameters that are specific to the type of temporal sharing.

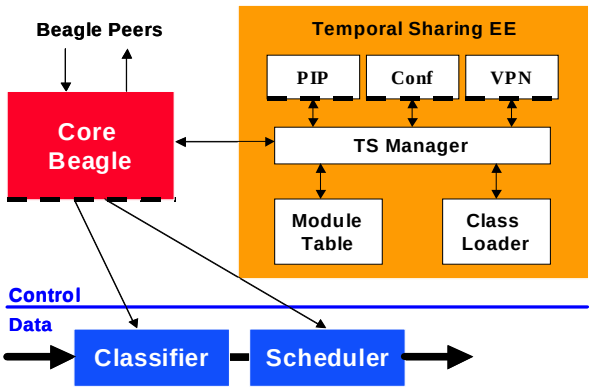

Figure 7. Beagle extensible temporal sharing architecture

The information in the `TemporalSharing` object is interpreted by dynamically downloaded AAs that implement support for a particular style of temporal sharing (such as conference, VPN, etc.). These code modules execute inside a *temporal sharing execution environment (TSEE)* which is responsible for interacting with the Beagle entity at that router to setup resources for flows with temporal sharing behavior. Figure 7 shows the design of the temporal sharing execution environment. The core Beagle module provides basic signaling support. The Temporal Sharing (TS) manager supports customizable temporal sharing optimization through the execution of temporal sharing AAs.

Core Beagle uses the TS manager to optimize the bandwidth allocated to flow groups. When it receives a request that includes a temporal sharing object, it passes the request to the TS manager. It will use the appropriate sharing module to calculate the aggregate resource requirements for the flow group, based on information stored in the request and its internal state. Let us look at a simple example of conferencing application with N sources sending video streams of bandwidth V. At any given time, any receiver only listens in to one sender. When a router receives a first request for a flow from this flow group, the conference sharing module will return a bandwidth requirement for the group of V. Later flow setup requests for this group will not change the bandwidth requirements for the group, so the conference sharing module will continue to return a bandwidth requirement of V.

5.3 API

The *Beagle Temporal Sharing Interface (TSI)* defines the interface between Beagle and the active sharing modules. Every sharing module must implement this interface. The primary calls are an "Add Flow" and "Delete Flow" call; Beagle issues these calls in response to a flow setup or tear

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

down request. In both cases, Beagle passes the temporal sharing object that is part of the flow setup and tear down request to the sharing module. Both calls return the aggregate resource requirements for the flow group to which the flow belongs.

The design of temporal sharing support in Beagle is driven by the goal of keeping the active sharing modules as simple as possible. Therefore, most of the functionality required to implement temporal sharing such as the definition of group instances, allocation of resources for group instances and arbitration of the shared resource during runtime are all implemented in the core non-extensible part of Beagle. The active sharing modules need only be concerned with calculating the aggregate resources for a particular group instance. This isolates the active sharing modules from the details of having to deal with the traffic control entities and simplifies the implementation of new sharing behaviors.

5.4 Security and safety

The TSI was designed to provide robust and predictable behavior in the presence of failures in the temporal sharing execution environment by providing a very simple interface that restricts the scope of actions that can be performed by the active sharing modules. Specifically, if the temporal sharing module does not respond or returns an error or an unacceptable bandwidth value (e.g. higher than the user is allowed to allocate), Beagle ignores the temporal sharing option and falls back on independent flow-based allocation.

5.5 Implementation

The active sharing modules are implemented use the Java programming language based on Java's support for safe-execution of downloaded code modules, support for implementing security policies, and wide-spread popularity. The TSEE is implemented as a Java virtual machine process using JDK 1.1. The Beagle daemon is itself implemented in C and allocates resources for a flow using the router control interface (RCI) at a router.

The main thread of control in the TSEE is the TS manager. The TS manager maintains a module table that has references to downloaded sharing modules of a particular type. The module table can also be used to implement caching strategies. The TSEE also implements a class loader that can dynamically load classes that implement a particular sharing module given the code URL associated with that module.

The TSI is specified as a Java interface specification. Each active sharing module must define a class that implements this interface. Each sharing module can create multiple threads. The TS manager thread can control how much CPU is allocated to the sharing module by enforcing thread priorities. The TS manager acts as an intermediary between the Beagle daemon and the active sharing modules. It implements a serialization protocol across the TCP connection to the Beagle daemon that provides support for each TSI call. Each TSI call by the Beagle daemon causes the TS manager to invoke the corresponding method of the sharing module of that particular type. The values returned by the method invocation are serialized and passed back to the Beagle daemon.

The temporal sharing module of the Beagle daemon implements the other end of the serialization protocol between the Beagle daemon and the TS manager. It provides an interface for the rest of the Beagle daemon to utilize services provided by the active sharing modules. Each TSI call is handled as a request-response transaction over the TCP connection. The temporal sharing module is also responsible for dealing with all the error conditions that might occur during any TSI transaction.

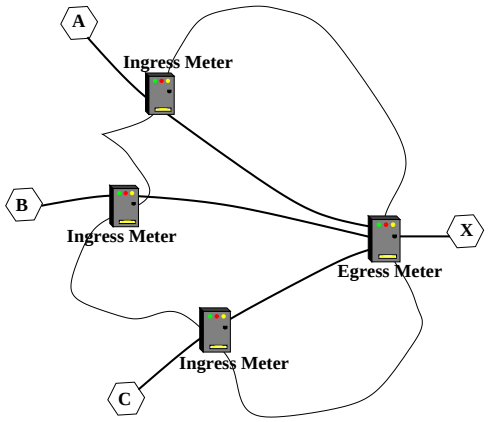

Figure 8. Multiple ingress domains (A, B, C) sending to single egress X

6 Dynamic SLAs for differentiated services

6.1 Motivation

The Differentiated Services (DiffServ) framework [8] supports network quality of service using a simple network core that treats packets belonging to one of a small number of service classes in the "same way". Traffic is policed at the entry points to the network according to *service level agreements* (SLAs). The SLA [8, 7, 3] between the customer (user or another service provider) and the service provider defines the traffic contract and the guarantees that the customer should receive from the network based on the customer's needs and the provider's policies.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

Simple SLAs require only static enforcement of the traffic contract. An example is point-to-point SLA where the traffic enforcement need only be done at the ingress. The more interesting case of managing an SLA is when it involves multiple ingress nodes (Figure 8). In such an SLA, the user (typically an ASP) might want to control the traffic distribution across the ingress points and customize the distribution dynamically. For example in Figure 8, an ASP may have multiple customers at sites A, B, and C, and it may want to change the bandwidth distribution based on how many customers are active at each site.

When the distribution of traffic across ingress nodes is dynamic, using a static allocation of bandwidth to each ingress point would require over-allocation and would lead to wasted bandwidth. Being able to use an SLA that allows the *dynamic* assignment of shares across ingress points based on user's current needs, is more efficient. This can be done in a number of ways. One solution is that the user can specify a set of rules that the NSP should use to determine the bandwidth distribution. Rules would be of the form: if condition X holds, then give weight Y to ingress node Z. This however raises the question of how rich the language for specifying the rules should be. Also, it is not clear how easily user-specific factors, e.g. how many customers are active, can be incorporated. We propose an alternative solution where users control the bandwidth distribution directly.

6.2 Design

We support traffic sharing across ingress nodes under user control by allowing users to download active modules onto the ingress routers. These active modules, which we will call ingress meters, can monitor the traffic and collect other statistics, and then decide how the user's traffic share should be distributed across the ingress routers. This results in a set of weights, one for each ingress router. Coordination across ingress meters can be done in a centralized or in a distributed fashion. Given that the number of ingress routers will typically be small, a centralized design will typically suffice. In a centralized design, ingress meters periodically report relevenant statistics to a meter coordinator, which calculates ingress shares, and distribute them to the ingress meters.

In this case, the base service is DiffServ, which has been extended by an EE that can host ingress meters. There are many ways of deploying such a service. A first option is that the router has native DiffServ support, in which case it should be possible to implement a simple EE that supports dynamic SLAs. An alternative is that DiffServ is supported through a service EE (e.g. Figure 4(a)) or a service AA (e.g. Figure 4(b)). Our implementation uses the service EE option, using Darwin as a starting point(see below).

6.3 API

The API that ingress meter AAs need from their execution environment has three types of calls. First, they must be able to collect traffic statistics. Second, They must be able to communicate with the meter coordinator (centralized design) or other ingress meters (distributed design); they may also have to be able to communicate with other control entities, for example to learn about external events that may affect the traffic distribution. Finally, they must be able to specify the bandwidth share on for that router in an SLA. This is a simple call that must specify the SLA and the bandwidth share.

6.4 Implementation

We have implemented DiffServ and dynamic SLAs in the context of the Darwin system [14]. We first added DiffServ traffic conditioning support to the input ports in the dataplane, i.e. token bucket meter, a marker, and a dropper. Next we extended the RCI so that control plane extensions can set up traffic conditioning blocks and control their operation.

The DiffServ EE is implemented as a variant of the Darwin delegate EE. Ingres meter AAs use the Darwin support for collecting traffic statistics and for communication. Detailed traffic statistics can be selected by instructing the packet classifier to forward certain classes of packets to a traffic statistics module, which calculates and stores the information; the ingress router can then periodically retrieve the traffic statistics information. The call that changes this router's share of an SLA's bandwidth is implemented by a simple method that translates share into a new set of parameters for the data plane traffic conditioning blocks.

6.5 Safety

The safety of calls for collecting traffic statistics and for communication is inherited from Darwin. It is very easy for the NSP to verify that the weights specified by the ingress meters on all the ingress routers satisfy the SLA. Each ingress router could periodically report all the shares for all SLAs to a central control station, which could verify that users are not violating the SLA. Note that compliance with an SLA can be more complex than just verifying that the aggregate bandwidth is below the SLA capacity. The SLA can also include constraints such as "no ingress router can use more than 50% of the SLA capacity".

7 Other examples

Besides the examples discussed in the previous three sections, we are working on several other examples of cus-

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

tomizable services. One example is End System-based Multicast (ESM) [27]. ESM provides multicast service based on a overlay that connects end-points and proxies running on active routers in the network. One of the advantages of this approach compared with traditional IP multicast is that intermediate ESM nodes can perform packet processing on the multicast flows. We are exploring how this capability can be used to support reliability and congestion control in a customizable way. We are also developing customizable services for video transcoding.

In the customization examples presented in this paper, the API available to customization AAs ranges from extremely narrow (Beagle temporal sharing, management of dynamic SLAs) to fairly broad (Darwin delegates). An interesting question is how powerful the API "should be". There is of course no simple anser to this question. For a particular service, the capabilities exposed by the customization API will depend very strongly on how much flexibility the provider is willing to give its customers; a related issue is how much leverage the customer has in demanding control over the service. On a more technical note, we found that the more narrow APIs were much easier to implement and secure. They are also more elegant and it is easier to justify why they are needed.

8 Related work

There are three areas of related work. The first area consists of research in the architecture of active nodes. Work in this area was discussed in Section 3. The second area consists of research in customizing network functionality using active networking. There are a number of projects that have looked at customization.

The CANES project [5, 6] defines an active networking architecture in which users can select the set of router functions that can process their packets. Moreover, users can provides parameters to these packet processing functions. The CANES functions are in effect (local) service components, which users can use to build more complex services. The parameterization can be viewed as a restricted form of customization as discussed in this paper. Concast [11] is a network service allows users to deal in intelligent with reverse path communication in multicast. User can specify code that defines how packets that flow back to the multicast source are combined. One can view this as a customizable service following the architecture defined in this paper. The base service is basic Concast with a default merging policy.

A number of groups have looked at using active networking to support the monitoring and managing of networks. One example is the Smart Packets project [32] at BBN. Packets can carry diagnostic programs written in a special-purpose language to diagnose the system. One can view the smart packets as customizing diagnostics using an extensible diagnostic service. Alternatively, one can look at the extensible diagnostic support as being part of the basic active node infrastructure (Node OS).

The ISI ASP project [4] has similar goals to the Darwin delegates discussed in Section 4. It allows users to dynamically install new control plane protocols. The equivalent of the Darwin RCI is the PPI. There are also several differences between ASP and Darwin delegates. The Darwin project focused on providing customizable QoS while ASP focused on more general control plane protocols, e.g. routing protocols and RSVP. Also, Darwin delegates require specific data plane support (for example, some of the security features require a hierarchical packet scheduler), while ASP was designed to run on a generic Unix-based router.

A final area of related work is that of using active networking to support composable network services. Example projects in this area include the Panda project [21] and the Ninja project [25]. Both projects developed support for path-based services, where multiple services could operate on a flow along the path between the sender and the receiver. We are not aware of any work on service customization in this context.

9 Conclusions

Active networking a powerful technology that supports the insertion of new functionality into the networking. In this paper we looked at how active networking technology can be used to customize network services.

We observed that users often want slightly different versions of network services such as multicast and network quality of service. We propose to implement these services as a base service that provides the basic service functionality and a customization module that allows users to customize the function of the service. We compare this architecture with the traditional active networking architecture based on execution environments and active applications. We also present several examples of customizable network services.

Acknowledgements

This research was sponsored in part by the Defense Advanced Research Project Agency an monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518.

The authors would like to thank the members of the Libra group, specifically Yang-Hua Chu, An-Cheng Huang, Sanjay Rao, Srini Seshan, and Hui Zhang, for the many insightful discussions that helped in shaping this paper.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

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*, May/June 1998.

[2] S. Alexander, M. Shaw, S. Nettles, and J. Smith. Active Bridging. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, pages 101–111. ACM, September 1997.

[3] Y. Bernet, D. Durham, and F. Reichmeyer. Requirements of Diff-serv Boundary Routers. Internet Draft, November 1998.

[4] S. Berson, R. Braden, T. Faber, and B. Lindell. The ASP EE: An Active Network Execution Environment. In *Paper Collection DARPA Active Networks Conference and Exposition*. IEEE CS Press, 2002.

[5] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. An Architecture for Active Networking. In *High Performance Networking (HPN'97)*, White Plains, NY, April 1997.

[6] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Congestion Control and Caching in CANES. In *Proceedings of ICC '98*, Atlanta, GA, 1998.

[7] M. Biegi, R. Jennings, S. Rao, and D. Verma. Supporting Service Level Agreements using Differentiated Services. Internet Draft, work-in-progress, November 1998.

[8] D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. Network Working Group, RFC 2475, December 1998.

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

[10] K. Calvert. Architectural Framework for Active Networks, December 2001. Version 1.1. Available from the web through URL http://www.dcs.uky.edu/ calvert/arch-1-0.ps.

[11] K. L. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: Design and Implementation of an Active Network Service. *IEEE Journal on Selected Areas in Communications*, 19(3):–, March 2001.

[12] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. Villela. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communication Review*, 29(2):7–23, April 1999.

[13] P. Chandra, Y.-H. Chu, A. Fisher, J. Gao, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable Resource Management for Value-Added Network Services. *IEEE Network Magazine*, 15(1):22–35, January/February 1998.

[14] P. Chandra, A. Fisher, C. Kosak, T. S. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable Resource Management for Value-Added Network Services. In *Sixth International Conference on Network Protocols*, pages 177–188, Austin, October 1998.

[15] P. Chandra, A. Fisher, and P. Steenkiste. Beagle: A Resource Allocation Protocol for an Application-Aware Internet. Technical Report CMU-CS-98-150, Carnegie Mellon University, August 1998.

[16] P. Chandra, P. Steenkiste, and A. Fisher. Extensible Signaling for Temporal Sharing. *IEEE Journal on Selected Areas in Communications*, 19(3):–, March 2001.

[17] P. R. Chandra. *A Signaling Protocol for Value-added Network Services*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[18] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of the ACM SIGCOMM '98 conference*, pages 229–253. ACM, August/September 1998.

[19] L. Delgrossi and L. Berger. Internet Stream Protocol Version 2 Protocol Specification - Version ST2+, August 1995. Internet RFC 1819.

[20] N. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. Ramakrishnan, and J. van der Merwe. A Flexible Model for Resource Management in Virtual Private Networks. In *Proceedings of ACM SIGCOMM '99 conference*, pages 95–108, Cambridge, September 1999.

[21] V. Ferreria, A. Rudenko, K. Eustice, R. Guy, V. Ramakrishna, and P. Reiher. Panda: Middleware to Provide the Benefits of Active Networks to Legacy Applications. In *Paper Collection DARPA Active Networks Conference and Exposition*. IEEE CS Press, 2002.

[22] J. Gao and P. Steenkiste. An Access Control Architecture for Programmable Routers. In *2001 IEEE Open Architectures and Network Programming (OPENARCH'01)*, pages 15–24, Anchorage, April 2001.

[23] J. Gao, P. Steenkiste, E. Takahashi, and A. Fisher. A Programmable Router Architecture Supporting Control Plane Extensibility. *IEEE Communications Magazine, special issue on active and programmable networks*, pages 152–159, March 2000.

[24] J. Gao, P. Steenkiste, E. Takahashi, and A. Fisher. A Programmable Router Architecture Supporting Control Plane Extensibility. CMU technical report, CMU-CS-00-109, March 2000.

[25] S. Gribble, M. Welsh, R. von Behren, E. Brewer, and D. C. et. al. The ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473–497, March 2001. Special issue on Pervasive Computing.

[26] A. Gupta, W. Howe, M. Moran, and Q. Nguyen. Resource Sharing in Multi-Party Realtime Communication. In *Proceedings of INFOCOM 95*, pages 1230–1237, Boston, MA, Apr. 1995.

[27] Y. hua Chu, S. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM'01 conference*, pages –. ACM, August 2001.

[28] L. K. Lim, J. Gao, T. E. Ng, P. Chandra, P. Steenkiste, and H. Zhang. Customizable Virtual Private Network Service with QoS. *Computer Networks*, page to appear, 2000.

[29] W. Marcus, I. Hadzic, A. McAuley, and J. Smith. Protocol Boosters: Applying Programmability to Network Infrastructures. *IEEE Communications Magazine*, 36(10):79–83, Oct. 1998.

[30] L. Peterson. Node OS and Interface Specification, January 2001. Available from the web through URL http://www.cs.princeton.edu/nsg/papers/nodeos.ps.

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE

[31] G. Phillips, B. Braden, J. Kann, and B. Lindell. ASP PPI: An Active Execution Environment's Protocol Programming Interface, May 1999. Available at URL http://www.isi.edu/active-signal/ARP/DOCUMENTS/PPI.ps.

[32] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, D. Rockwell, and C. Partridge. Smart Packets for Active Networks. In *1999 IEEE Open Architectures and Network Programming (OPENARCH'99)*, pages 90–97, New York, March 1999.

[33] I. Stoica, H. Zhang, and T. S. E. 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.

[34] E. Takahashi, P. Steenkiste, J. Gao, and A. Fisher. A Programming Interface for Network Resource Management. In *1999 IEEE Open Architectures and Network Programming (OPENARCH'99)*, pages 34–44, New York, March 1999.

[35] D. Tennenhouse, J. Smith, D. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Networking Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[36] D. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[37] 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.

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

Proceedings of the DARPA Active Networks Conference and Exposition (DANCE'02)
0-7695-1564-9/02 $17.00 © 2002 IEEE