Dynamic Management for Enabling QoS in a Campus Network

Ashu Razdan & Gautam Barua
Computer Science and Engineering Dept
Indian Institute of Technology, Guwahati
Assam, India.
razdan@iitg.ernet.in & gb@iitg.ernet.in

Abstract

Bursty, high-bandwidth, real-time and mission-critical applications are driving the need for better differentiation of service in the campus network beyond that provided by traditional best-effort services. The ability to provide these services depends on the availability of cohesive management tools that can be used to provision and monitor sets of routers in a coordinated manner. To facilitate the development of such a management tool it is essential to first define a framework that aims at setting up generic QoS-enabling components across the campus network. In this paper we study such a scheme for enabling QoS in a campus network and discuss the design and implementation of a management tool that eliminates the need for device-by-device command streams. We also investigate the performance improvement gained by a proposed change in the three-tier mechanism.

1.0 Introduction

Originally designed for research networks, the protocols used in today's Internet have been mainly optimized to provide connectivity. In these networks, the main problem was to reach the destination even if transmission quality was poor. Due to this fact, the Internet and most IP based networks provide today a best-effort service, i.e. a context where the network does its best to transmit information as quickly as possible but does not provide any guarantee on the timeliness or even the actual delivery of this information.

In today's electronic trade context, there is a real demand for a minimum level of performance to be guaranteed to mission critical applications. This was to some extent supported in ATM networks which allow for rate based service categories. In general, with connection oriented networks, information transfers are allotted a guaranteed - yet unique - end to end service, but at the cost of complexity and execution overheads in network nodes. This demand of performance guarantees now reaches the Internet, which has evolved to an international, commercially operated network.

Corporate administrators are in need of simple and comprehensive mechanisms to deliver services. Introducing such mechanisms between users and the network means that, contrary to the current best-effort networks which handles all data equally, the network now needs to discriminate between various kinds of data traffic to provide multiple service levels. In this context, a data flow is conceived as a set of transfer units considered related to each others, according to some discriminating criteria on quantitative observations such as generating sources, users, applications, or destinations.

1.1 Enabling QoS

Until recently, ATM was the only technology designed for for providing effective QoS. However, ATM QoS needs an ATM network infrastructure, a rather unrealistic prerequisite at a time when connectionless frame-switched networks are omnipresent, especially in the campus [3]. Today new standards and technologies are continually being enhanced to provide QoS in pragmatic ways.

One such approach is the deployment of cohesive management tools that can be used to provision and monitor a set of routers in a campus network, in a coordinated manner. As this paper aims at defining and implementing such a conceptual model, it becomes imperative to clearly outline the issues that confront the deployment of such an architecture.

1.2 Problems in Campus Network Management

In the present scenario concerning campus networks, enabling QoS and managing it effectively are two of the most nagging problems faced. These problems can be solved effectively if the following two aspects are dealt with properly.

- Bandwidth Management : The state of the network needs to be monitored for performance measures. Appropriated actions also ought to be taken in times of failures or when performance has degraded below acceptable limits. The importance of Congestion Control can not be emphasized enough.
- Access Control : The traffic on the network needs to be regulated for security reasons, or efficiency, or both.

Our work mainly deals with issues concerning bandwidth management and aims at setting up an infrastructure that can essentially support both the aspects. Extending our work to enable the second mechanism is an interesting prospect for future work in this area.

The remainder of this paper is organized as follows. Section 2 starts with a discussion on the basic schemes that are presently being proposed for incorporating QoS, and concludes by choosing the architecture that is to be used in our implementation. Section 3 begins by sketching the high-level blocks of the required model and extends to list and detail all the components required for a successful implementation. This section also discusses the underlying transport infrastructure and network environment. The entire implementation is then discussed in Section 4, with Section 5 providing some insightful remarks about how the implementation was tested. This section also evaluates the performance improvement gained by a proposed change in the existing communication mechanism between the QoS components. The Conclusion and some further avenues for extension of this work are discussed in Section 6

2.0 Design Issues

2.1 Types of QoS

There are essentially two approaches to enabling QoS. These QoS protocols and algorithms are not competitive or mutually exclusive, but on the contrary, they are complementary.

- Reservation (integrated services) : Network resources are apportioned according to an application request, and subject to bandwidth management policy. RSVP provides the mechanism to do this.
- Prioritization (differentiated services) : Network traffic is classified and apportioned according to bandwidth management policy criteria. To enable QoS, classifications give preferential treatment to applications identified as having more demanding requirements.

These protocols can also be designed for use in combination to accommodate the varying operational requirements in different network contexts. However, this approach is out of the scope of our work.

2.2 Which model to use

RSVP (reservation based) provisions resources for network traffic, whereas DiffServ (priority based) simply marks and prioritizes traffic. The former is more complex and demanding than the latter in terms of router requirements, so can negatively impact backbone routers. DiffServ uses a simple and coarse method of providing differentiated classes of service, whereas to enable the reservation scheme, records corresponding to all the flows will have to be maintained in all the intermediate routers. A campus network usually supports the traffic generated by hundreds of customers. This motivates us to choose DiffServ as our basic QoS-enabling scheme.

2.3 The DiffServ idea

The DiffServ [14] architectural model improves the scalability of QoS provisioning by pushing state and complexity to the edges of the network and keeping classification and packet handling functions in the core network as simple as possible. Briefly put, flows are classified, policed, marked and shaped at the edges of a DS (DiffServ) domain. The nodes at the core of the network handle packets according to a Per Hop Behavior (PHB) that is selected on the basis of the contents of the DS field in the packet header.

From the description of the behavior supplied by a PHB, it is intended that one can make a service description; at least that part of the service description that says what effect a service has. Although others are possible, there are currently two standard PHBs defined that effectively represent two service levels [5, 13]

- Expedited Forwarding (EF) : This has a single codepoint (DiffServ value). EF minimizes delay and jitter and provides the highest level of aggregate QoS. Any traffic that exceeds the traffic profile (which is defined by the local policy) is discarded.
- Assured Forwarding (AF) : This has four classes and three drop preferences within each class (so a total of twelve codepoints). Excess AF traffic may not be delivered with as high priority as the traffic "within profile", but is not necessarily dropped.

But the most distinctive feature of DiffServ remains the segregation of the traffic into classes. The concept behind this classification can be explained as follows.

2.4 Class of Service

The Class of Service concept divides network traffic into different classes and provides class-dependent service to each packet depending on what class it belongs to. While the strict QoS has some absolute measures for quality, the CoS has relative measures: At this time this class gets packet drop probability of 10^{-6} while on the other class packet drop probability is 10^{-3}. To differentiate the network traffic into the different classes, the differentiation must be based on some factor.

This section discussed the basic schemes that are presently being proposed for incorporating QoS. We also chose the architecture that is to be used in our implementation, and also reasoned out why we selected the same. The next section sketches the high-level blocks of the required model and extends to list and detail all the components required for a successful implementation. It also discusses the underlying transport infrastructure and network environment.

3.0 Design Overview

A typical picture of network resource control involves a management tool, a policy repository, a policy decision point and a policy enforcement entity. The network administrator uses the management tool to populate the policy repository with a number of policy rules that regulate access/use of network resources. These rules could specify for instance, the service category to be employed for a particular application, how much bandwidth is allocated to a particular flow or ToS category, etc.

The administrator-specified rules are stored in the policy repository in a well-understood format or schema. The decision entity downloads the policy rules. The enforcement entity is the router, which encounters packets flowing across the network. It queries the decision entity for specific actions that are to be applied in conditioning the packet stream.

This architecture is illustrated in Figure 1. This architecture supports the out-sourcing of policy decision making to a separate Policy Decision Entity resident on a specialized policy server. From a directory-oriented perspective, the network consists of the directory server function, and one or more directory clients which query/update the directory. Some of the directory clients are management tools, which populate and maintain the policy repository in the directory. Others are enforcement entities that apply policy rules by dropping, marking, reshaping or otherwise conditioning the packet stream. Examples of such clients include routers, firewalls, and proxy servers. We assume that policy rules are relatively static. The directory can be queried by the enforcement entity when needed, for instance, triggered by events such as an IP packet bearing a TCP connect request, etc.

The scenario that has been outlined above pertains strictly to the components within a domain, as the inter-domain issues are beyond the scope of our work.

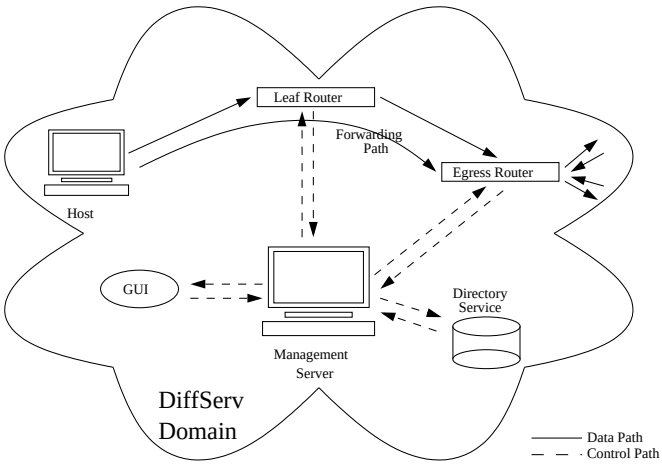

Figure 1. The Design

3.1 The QoS Components

The various QoS components that were to be implemented in our work consisted of the following:

1. DiffServ Enabled, Remotely Configurable Routers (Policy Enforcement Point)
2. Management Node (Policy Decision Point)
3. Directory Service (Policy Repository)

The implementation issues of these components are discussed in detail in the following sections.

3.1.1 QoS enabled Routers

The QoS enabled routers [14], or the Policy Enforcement Points, are where the policy decisions are implemented. As the QoS architecture we have adopted is DiffServ, this component must have the functionalities to support DiffServ. The Linux kernel v.2.2.14-12 has many built-in features like packet marking, classification and shaping. Due to the presence of these traffic control mechanisms, the component being discussed has been hosted in a Linux kernel.

Leaf routers are configured with a traffic profile for a particular flow based on its packet header. Figure 2 (left) shows what happens to a packet that arrives at the leaf router, before it is passed to the forwarding engine. All arriving packets must have both the EF-bit and the AF-bit cleared after which packets are classified on their header. If the header does not match any configured values, it is immediately forwarded. Matched flows pass through individual Markers that have been configured from the usage profile for that flow.

Figure 2 (left) a shows a schematic, and Figure 2 (right) shows the inner workings of the Marker. For both Assured and Premium packets, a token bucket “fills” at the flow rate that was specified in the usage profile. For Assured service, the token bucket depth is set by the Profile’s burst size. For Premium service, the token bucket depth must be limited to the equivalent of only one or two packets. When a token is present, Assured flow packets have their AF-bit set to one, otherwise the packet is passed to the forwarding engine. For Expedite-configured Marker, arriving packets that see a token present have their EF-bits set and are forwarded, but when no token is present, Premium flow packets are held until a token arrives. If a Premium flow bursts enough to overflow the holding queue, its packets will be dropped.

Thus the implementation of the forwarding path uses extended features, in addition to the QoS features available in the Linux kernel, to set up differentiated packet treatments.

The main function of the *Core Router* router used in our model is illustrated in Figure 3. This simple border router input interface is a Profile Meter constructed from a token bucket whose rate can be dynamically varied. The QoS Manager can configure this rate by communicating with the routers. After the QoS Manager has computed how much of the data should leave the domain, it sets the values for parameters like EF token rate & AF token rate. The EF token bucket size and AF

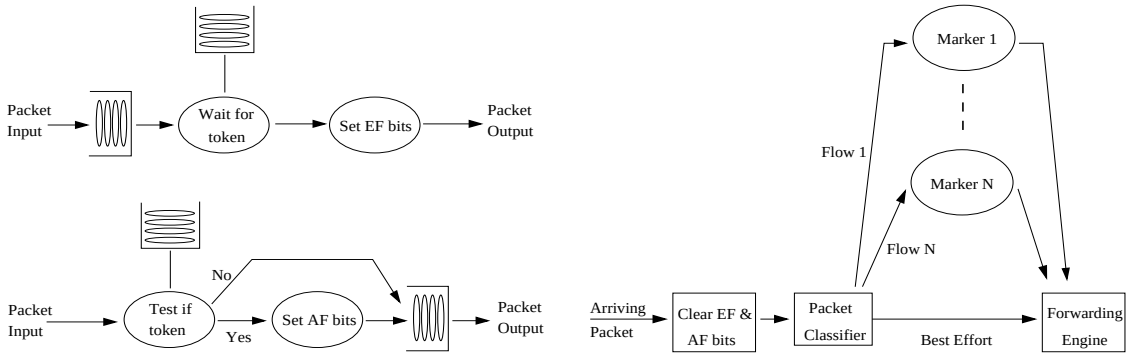

Figure 2. Packet Marking

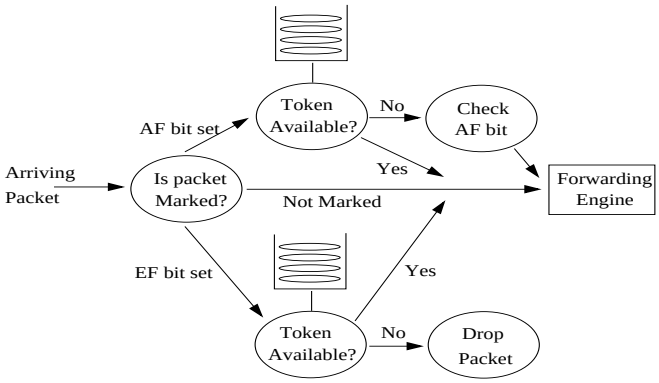

Figure 3. Processing at the Core Routers

token bucket size parameters can be kept static. (This is in contrast to models where every flow that crosses the boundary must be separately policed and/or shaped.)

One other function that the edge router has to perform is to give the QoS Manager a feedback on the amount of traffic that is being dropped so that the QoS Manager can, in turn, take corrective actions to recompute the appropriate values of the above-mentioned parameters so that network resources are not wasted.

After the initial configuration is set up in the router, it then performs the function of policing the traffic as shown in Figure 3. This policing ensures that the amount of traffic that is allowed to leave the domain conforms to the value that the QoS Manager has agreed upon.

3.1.2 The QoS Manager (Policy Decision Point)

The QoS Manager is a software entity that manages resources for IP QoS services. It is responsible for control decisions according to a policy database. Based on such decisions it configures the routers within the domain. The algorithms is shown below:

1. Traffic originating from an end-host reached the leaf-router for being forwarded.
2. The leaf-router finds that the priority information for the <IP, Application Type> pair and caches it.
3. The leaf-router contacts the QoS manager (through an RPC call, as would be clear after the reader is familiar with the Implementation discussed in subsequent sections), requesting for the priority information.
4. The QoS manager, in turn, contacts the Directory Service (Policy Repository) to query for the same.

5. After receiving the response from the directory, the QoS manager return the corresponding priority back to the leaf-router.

6. The leaf-router thus marks the traffic appropriately.

7. All the traffic with the same ToS field, is handled in a similar fashion at the core-routers, in conformance with the idea proposed in the DiffServ architecture.

This component is based on a client-server architecture. All the Routers, on startup, will connect to the QoS Manager to set up basic configurations and keep updating the information henceforth at regular intervals. The Manager then waits for incoming requests from the routers, mainly for priority resolution of the traffic.

3.1.3 The Directory Service

For storing and look-up of local policies, an LDAP system needs to be used for facilitating network management and account generation. The Policy Repository was visualized to be a Directory Services the policy information is not expected to change at a fast rate. As the number of updates are expected to be much less than the number of reads, a Directory Service fits the bill as they are optimized for reads.

Our goal is to use a preexisting directory service that will suit the needs of a campus. This is because we wish to augment a system that is flexible enough to be adapted for other purposes, and not just be used exclusively for the purpose of our project. Thus, it is recommended that a general, campus-wide architecture for a directory service be established that can encourage uniformity in use. The LDAP Directory-QoS Manager communication is done through the LDAP protocol. Java Naming and Directory Service (JNDI) APIs can also be used for the same.

The QoS Manager, which was discussed in the previous section, uses this information to allocate bandwidth to the prospective customer. For e.g., the faculty in a campus can be assigned a higher priority than the students. The attributes can be set in whichever manner the administrator feels would serve the purpose.

This section sketched the high-level blocks of the required model and gave a detailed analysis of all the components required for a successful implementation. It also discussed the underlying transport infrastructure and network environment. The next section deals with the implementation of the entire model.

4.0 Implementation

The various QoS components that were implemented in our work consisted of the following:

1. DiffServ Enabled, Remotely Configurable Routers (Leaf/Core/Edge)

2. Management Node

3. Policy Repository

4.1 Components at the Router

Although the role played by the different classes of routers (Leaf/Core/Edge) is very different, but the modules developed for them are quite similar in function. As can be realized easily, the need is for two basic functionalities at the routers. These can be broken up and relegated to two levels and various components as detailed below.

The first requirement is to be able to view the traffic statistics that are relevant to the management server. An example is SNMP. This protocol exchanges information, such as the drops at a particular network interface of a router, over the network. As expected, the router's kernel exports these statistics through a file (typically in the /dev directory, or the /proc filesystem). However, it was seen that a number of the existing exports were not relevant to the proposed architecture, and worse still, a number of the statistical information required had not been exported. This lead to the development of 2 software components (modules) which have the functionalities as detailed below.

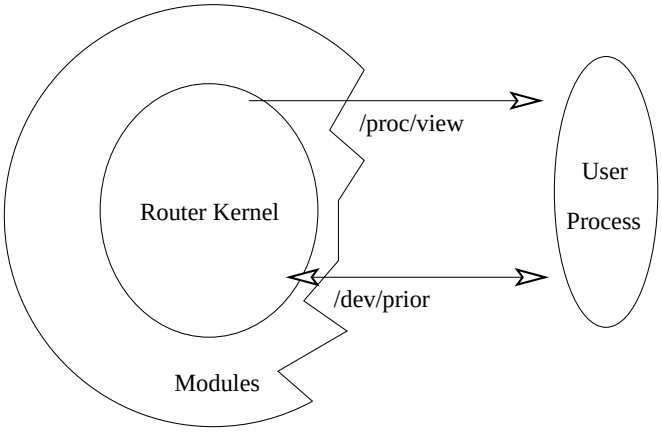

Figure 4. Components at the Router

4.1.1 The Kernel Modules

1. /proc/view All the traffic statistics extracted by the packets being forwarded by the router can be viewed through this file. The output is shown below. The output consists of three lines per entry. The format is:

   ```
   Line 1: <IP Address> <Priority> <Timestamp> <(R)esolved/(U)nresolved>
   Line 2: Labels, e.g. IP, ICMP, etc.
   Line 3: Corresponding statistics

   [root@jazz jazz]# cat /proc/view
   9.141.80.11 7 27521 U
   IP ICMP IGMP TCP UDP
   0   0    0   12   0
   9.141.80.14 7 27523 U
   IP ICMP IGMP TCP UDP
   0   0    0   22   0
   9.141.80.12 7 27523 U
   IP ICMP IGMP TCP UDP
   0   0    0   18   0
   9.141.80.13 7 27535 U
   IP ICMP IGMP TCP UDP
   0   0    0   18   0
   ```

2. /dev/prior As our main goal is bandwidth management through priority assignments (DiffServ), a 2-way file-level interface is required to:

   - Probe for traffic with unresolved priorities.
   - Subsequently resolving them

   This is done through the device file “prior”, resident in the “/dev” directory. At any instant, a list of all the IP addresses of the hosts with unresolved priorities can be acquired by reading this file. This interface is 2-way, and can also be used to resolve the priorities by writing in the appropriate values (sent back by the management server). An example is shown below.

```
[root@jazz jazz]# cat /dev/prior
189828361 (9.141.80.11)
```

```
240160009 (9.141.80.14)
206605577 (9.141.80.12)
223382793 (9.141.80.13)

[root @temp]# echo "9.141.80.14 3" \
> /dev/prior

[root@jazz jazz]# cat /dev/prior
189828361 (9.141.80.11)
206605577 (9.141.80.12)
223382793 (9.141.80.13)
```

It can be seen that the IP address for the entry resolved by the second statement does not figure in the content of “prior” anymore. This is because the internal tables of the router’s kernel were updated immediately after the execution if the second statement. If we look at the contents of the file “view”, we notice the symbol “R” in the entry for this IP due to the same reason. The simplicity of the second statement is challenged by the realization that any process running as root can tamper with the priorities. This problem can be taken care of by using a randomly generated session key, which can be configured/changed by the process that is actually supposed to interact with this information.

```
[root@jazz jazz]# cat /proc/view
...
9.141.80.14 3 27523 R
IP ICMP IGMP TCP UDP
0 0 0 22 0
...
```

All these statistics are maintained in the kernel’s internal tables at the router. The data-structures of one of these tables will give us an insight into the functioning of the entire algorithm.

```
/* a table entry.*/

struct phb_ent {

u_int32_t saddr;
u_int8_t priority;
long *traf_stats;
unsigned long now;
int flg;
struct pktq *q;
};
```

4.2 Yin, The QoS Client

As the management server runs centrally, a process/daemon must run on all the routers to communicate with it. This Process interprets the queries from the management server and responds with the required information after filtering all the statistics available on the router. This process can be visualized as one of the end-points of the RPC connection, the other being the central server.

The implementation details of the QoS client are shown in Figure 5. This component runs on four threads, each of which carries out a different function as detailed below:

1. The Resolve thread : This thread is responsible for resolving all the priority requests, the router’s kernel makes. This threads sleeps when there are no requests, and is woken up whenever any requests are made. It should be understood that this thread does not poll the /dev/prior file, and is awake only when there are some requests to be resolved.

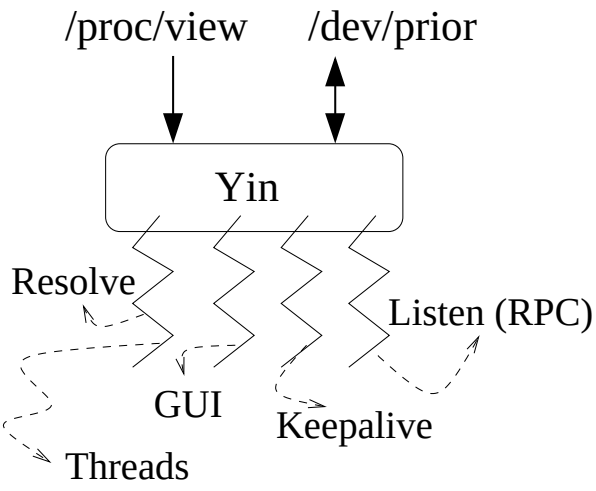

Figure 5. Yin, The QoS Client

2. The Listen thread : This thread is responsible for running the server module of the RPC communication between the QoS client and the manager. This is absolutely necessary to support calls originating from Yang.

3. The GUI thread : This component has been granted a GUI so that not only can router queries originate from the kernel, but some special requests can be supported got the local administrator (local to the router). This is to ensure maximum flexibility, even though the idea of having a GUI based process at the router may seem like a fancy idea.

4. The Keepalive thread : This thread keeps exchanging status information between the QoS client and the server. This mainly updates the traffic statistics of the router at the manager.

4.3 Yang, The QoS Manager

The QoS Manager runs centrally, and the client processes running on all the DiffServ enabled routers communicate with it.

Figure 6 (left) shows a screen-shot of the QoS Manager in action. The implementation details of the QoS Manager are discussed below. This component runs on two threads. They are:

1. The Listen thread : This thread is responsible for running the server module of the RPC communication between the QoS client and the manager. This is absolutely necessary to support calls originating from Yin.

2. The GUI thread : This thread supports the GUI for the manager. Unlike the GUI on the routers, having a GUI on the Manager is absolutely necessary, as the main aim of the project was to implement a centralized control mechanism for QoS policy deployment, and also eliminate device-by-device command streams.

The Keepalive and Resolve threads are absent from this implementation for obvious reasons. It is the responsibility of the the QoS clients (on the routers), and not the manager, to initiate the exchange of status information frequently. The GUI of this component has also been developed in Glade. This GUI currently supports a number of bandwidth management functionalities.

4.4 The Policy Repository

As mentioned in the previous section, the Policy Repository was visualized to be a Directory Service. This is because the Policy information is not expected to change at a fast rate. As the number of updates are expected to be much less than the number of reads, a Directory Service fits the bill as they are optimized for reads.

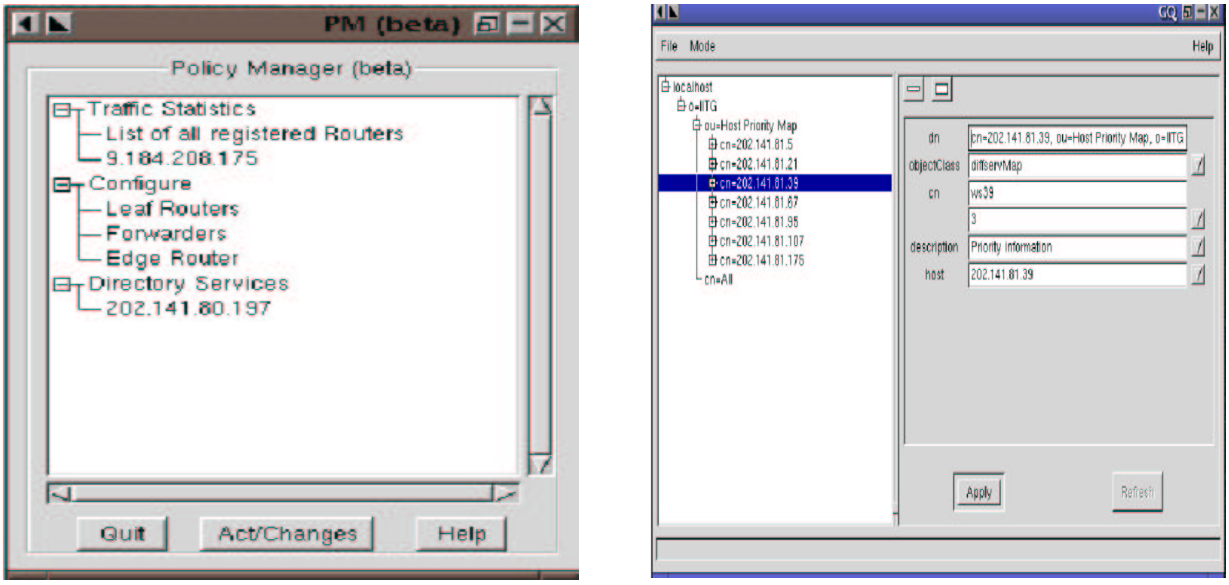

Figure 6. The QoS Manager (left), and the Policy Schema (right)

4.4.1 The Policy Schema

As can be seen from Figure 6 (right), the most important information that is being stored in the directory is the priority information. This is because the current implementation mainly delays with bandwidth management through service differentiation. The “priority map” incorporated in the schema keeps track of two types of mappings:

1. Source IP Address to ToS field map
2. Source Application Type to ToS field map

In its simplest mode of functioning, the Management tool is expected to allocated bandwidth to the hosts by assigning different priorities to different hosts. This means that all the traffic originating from a particular host is given the same priority, irrespective of the bandwidth requirements of the application that is generating the traffic. This seems to be an ad-hoc solution, and thus arose the need for the application-ToS map.

As is evident from the previous discussion, there was an urgent need for upgrading the schema to support a mapping between Application Types and the ToS field to be used in the underlying DiffServ architecture. This takes care of allocating higher bandwidth for applications that need more of it. For e.g., HTTP traffic should, intuitively, be given a lower priority than FTP traffic.

4.4.2 Who should change the Policy Information?

This is a very pertinent question that has to be answered before we can be absolutely sure that the model does achieve it’s ambitious goal. As one of the objectives of the project is to eliminate the need for device-by-device command streams, expecting the Network administrator to, painstakingly, change each of the priorities when needed, definitely goes against the very purpose of the project. Thus emanates the need for automation of the maintenance of policy information.

4.4.3 The Policy Scheduler

The need for the Policy Scheduler can be realized only after some real life cases are considered. Imagine a situation in which three different groups of an organization are given access to a common computing facility. The groups would be using the same hosts, at different instants of time. Now suppose that each of the groups is at a different level in the organizational hierarchy, and thus has been alloted a different priority for network bandwidth usage. As the host IP addresses remain unchanged through the whole exercise, there is a need to change the policy data that is resident in the Policy Repository.

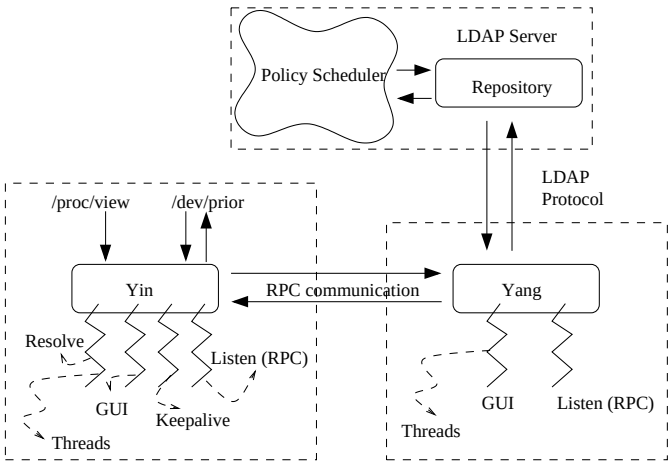

Figure 7. QoS Components

And this would have to be re-done many times in a single day, and that too for just one computing facility. Thus we have to automate the entire process of changing the policy data. This would be done by the *Policy Scheduler*. The design of a Policy Scheduler, however, is out of the scope of our work. This is a possible extension of our work.

4.5 Component Integration

The previous sections discussed the working and implementation issues pertaining to the individual QoS components. This section provides the exposition on how these components were integrated, or rather glued together. Figure 7 shows the various components that were implemented. As explained in the previous sections, they are listed as Yin (The QoS Policy Client), Yang (The QoS Management Tool), and the Policy repository. These components have well defined interfaces through which they communicate with each other. These inter-component interfaces are detailed below:

1. The Yin-Yang interface : The QoS Policy Client and Server, namely Yin and Yang, are connected through two RPC (Remote Procedure Call) interfaces, to provide a 2-way communication infrastructure. The calls of the type Yin->Yang, i.e. calls originating from the Policy Client, currently are used to resolve the priorities. The calls in the other direction (Yang->Yin), are used mainly to receive information regarding the traffic statistics at the Core router, and also to execute bandwidth management functions in case of Edge Routers in cases where the total outgoing/incoming traffic needs to be limited.

2. The Yang-Repository interface : This interface exports the policy information from the Repository to the Management Server. As the Repository is implemented as a Directory Service, the LDAP (Lightweight Directory Access Protocol) is used by the Management Server to query the Directory to get the policy information.

It was noted that the RPC interface gave way to some small memory leaks, which occasionally lead to Management Server crashes. Although some minor testing was done at the component level, the full Integration Testing was also done and is documented in the succeeding section. The implementation was thus seen to work quite efficiently.

4.6 Linux QoS support

Linux offers a rich set of traffic control functions. The traffic control framework available in recent Linux kernels offers all of the DiffServ related functionality we required in our implementation. We therefore closely followed the existing design and felt no need to add new components.

Our implementation makes use of the CBQ (class based queuing) functionality to facilitate traffic handling at the core routers. The incoming traffic is classified according to the ToS field in the IP header (iph->tos), and enqueued accordingly. By setting the queue lengths and assigning different classes to different flows, based on the ToS field, we accomplish the goal of handling the packets at the “Policy Enforcement Points”, i.e. the core routers.

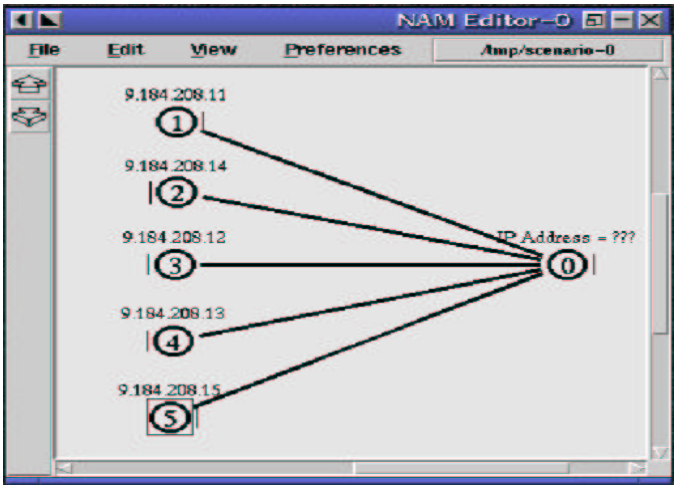

Figure 8. The Scenario

This section discussed the implementation of the entire architectural model. However, as the components are spread across the network, it becomes important to not only test the individual components, but also to carry out the integration testing. This exercise was carried out successfully and is explained in the next section. The performance improvement gained by a proposed change in an existing algorithm is also evaluated.

5.0 Emulation

This section describes the entire process used to test the implementation explained in Section 4, and to evaluate the performance of an algorithm that is proposed later in this section. To make sure that the various components being used to enable QoS behave in the way that they should, and do not incur performance losses, a test-bed was set up. This test-bed uses the traffic generation facility of ns. The entire exercise is explained in the following sections.

The *Network Simulator* (Ns) is an object-oriented software that is specialized in carrying out network based simulations, though the functionalities are not restricted to simulation only. Ns [11] can also be used for emulation purposes. Emulation refers to the ability of the simulator to inject traffic into a live network. Special objects within the simulator are capable of injecting traffic from the simulator into the live network and introducing live traffic into the simulator.

5.1 The Scenario

The endeavor is to introduce traffic into the network, and understand how it is being “handled” by the QoS components. To emulate conditions of a real campus network, an interesting scenario is adopted with the following assumptions.

- The number of source hosts is *exponentially distributed* with suitable boundary values.
- The destination address of the traffic generated by a particular node is generated randomly, and remains the same for that session.
- The nature of traffic generated in a particular session is *pareto distributed*.

Although the last of the three assumptions is true only for HTTP traffic, the same has also been assumed for other kinds of traffic. It should be understood, however, that other distributions can also be incorporated at the cost of increased complexity. A snapshot of the scenario can be seen in Figure 8.

The entire testing exercise may also be used to measure the performance of the chosen architecture to enable QoS. The next section discusses a variant of the algorithm that is employed at the leaf-routers to resolve the priority information. The algorithm under consideration incorporates a minor change that makes the resolution process more efficient.

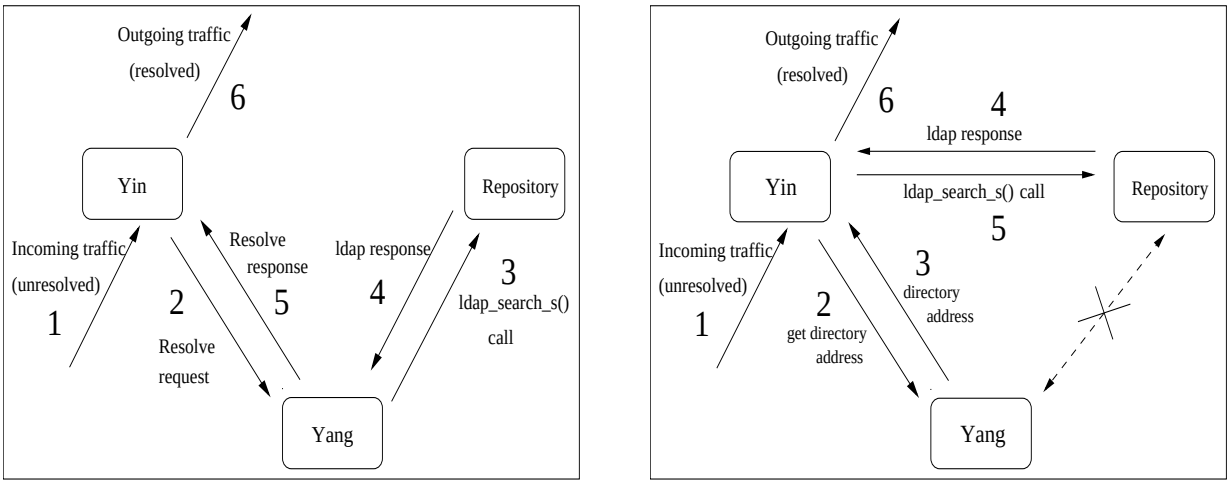

Figure 9. Two Priority Resolution Mechanisms

5.2 The Algorithm

The specifications [8] delineate a three-tier architecture, like the one adopted by our work. However, they do not specify whether the communication pattern, as shown in Figure 9 (left), ought to be followed at all times. An important limitation of this pattern of communication is that the QoS clients (the routers) are able to contact the Policy repository only through the Management node, and not directly. In cases where heavy imports of policy information need to be made by the routers, this mechanism is clearly not very efficient.

We propose an alternative methodology in such cases. According to the new mechanism, the routers contacts the Management node and requests for the address of the policy repository. The Management node responds with the network address of the directory service. All further policy imports are made by the routers directly, and not by the Management node on their part, thus reducing the delay. The entire mechanism is illustrated in Figure 9 (right).

It should be understood, however, that successful deployment of the altered mechanism is contingent upon the following assumptions.

1. The routers must be aware of the existing schema in the directory service, to some extent. This is because policy information can be imported only if corresponding are applied successfully in the LDAP queries made to the policy repository. This introduces some complexity in the implementation of the router utilities. However, the earlier algorithm (Figure 9. (left)) is not constrained by this assumption as the Management node has some prior knowledge of the policy schema.

2. The superior performance of the new mechanism is evident only if the policy information is being imported at a high rate. In cases where the load on the policy repository is low, i.e. the query rate is low, both the variants would perform equally well.

3. A *call-back* mechanism should be present. This would be important in cases when the network address of the policy repository changes. This problem could also be solved by setting up a time interval after which the routers would refresh all the control data, including the address of the repository. Thus whenever such an event, however rarely, occurs the status in all the policy-enforcement-points is updated.

5.3 Results

Figure 10 shows a plot between the time elapsed in processing of a query for policy information, versus the request number. The plot clearly shows that the proposed algorithm performs much better than the existing specifications. The graph indicates that there is an improvement of approximately 0.5 ms per request, on an average. This is a considerable reduction in the request delays.

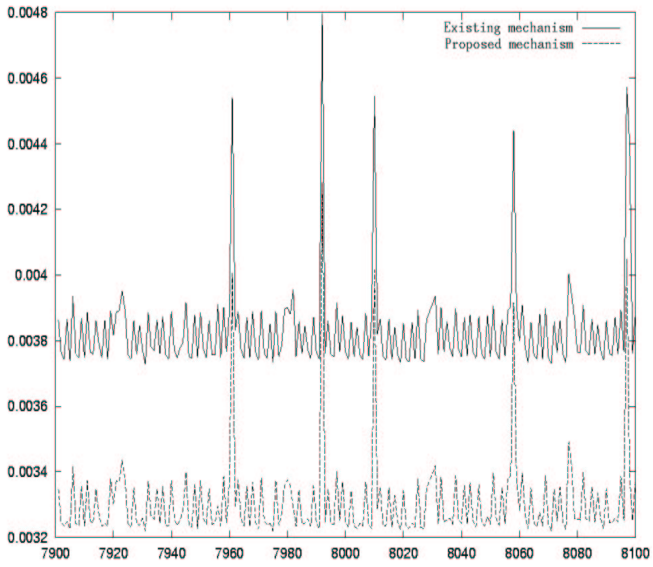

Figure 10. *query time vs request number*

The test-bed mentioned in this section may also be used to ensure that there are no memory leaks in the implementation. This is because quite a lot of traffic can be made to be handled by the components. If during the testing phase, a particular component breaks down, it is quite possible that the same might have occurred due to memory leaks in that component, as was experienced in our work.

6.0 Conclusion and Future Work

Enabling of QoS in campus networks [4] is a challenging endeavor. By deploying a management system, like the one that has been delineated and implemented in this work, much of the solutions can be achieved. The current implementation mainly supports Bandwidth Management. As the exposition in Section 1 suggests, we can extend these to support access control based management.

As the communication between the Management server and the process running on the routers is 2-way, thus providing all the infrastructure to move on to support more sophisticated management techniques.

Some architectures are currently being proposed for Inter-Domain Policy support. Our work pertains to enabling QoS in a single domain. To support Inter-domain policies, propose the Bandwidth Broker architecture wherein a Broker negotiates the amount of traffic to be permitted from one domain to the other, through the edge routers. In such a scenario, a communication channel must exist between the Bandwidth-Brokers, the Management Servers, and the Edge routers.

A framework is also being defined for mapping the priorities assigned to hosts in one domain, to the priorities they would eventually receive in the other domain. Though mapping priorities between domains does not seem challenging in cases where both the domains use the DiffServ architecture, a considerable amount of work would be needed when the architectures used by both the domains are different.

The entire testing exercise, discussed in Section 5 may also be used to measure the performance of the chosen architecture to enable QoS. It would be interesting to compare the performance of the various architectures. However, it should be understood that the traffic model used in the testing phase would have to be built upon to incorporate other traffic distributions and sources as well, to give dependable results.

References

[1] e. A. Terzis, J. Ogawa. *A prototype implementation of the two-tier architecture for differentiated services.* IEEE Real-Time Technology and Applications Symposium (RTAS99), June 1999.

[2] S. Blake. *Some issues and applications of packet marking for differentiated services.* Internet Draft, November 1997.

[3] S. Bradner. *Internet protocol quality of service problem statement.* Internet Draft, September 1997.
[4] N. S. F. Bennani. *Dynamic management for end to end ip qos: from best-effort to personalized services.* TERENA Networking Conference, May 2000.
[5] e. J. Heinanen, F. Baker. *Assured forwarding phb group.* RFC 2597, June 1999.
[6] e. K. Nichols, S. Blake. *Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers.* RFC 2474, December 1998.
[7] L. Z. K. Nichols, V. Jacobson. *A two-bit differentiated services architecture for the internet.* RFC 2638, July 1999.
[8] J. C. M. R. Rajan, S. Kamat. *Policy action classes for differentiated services and integrated services.* Internet Draft, April 1999.
[9] e. S. Blake, D. Black. *An architecture for differentiated services.* RFC2475, December 1998.
[10] e. S. Kalyanaraman, S. Arora. *A one-bit feedback enhanced differentiated services architecture.* Internet Draft, March 1998.
[11] S. F. S. McCanne. *ns-network simulator.* http://www-mash.cs.berkeley.edu/ns/, January 2000.
[12] R. W. T. Howes, M. Smith. *The java ldap application program interface.* Internet Draft, July 2000.
[13] K. P. V. Jacobson, K. Nichols. *An expedited forwarding PHB.* RFC 2598, June 1999.
[14] S. B. Y. Bernet, A. Smith. *Conceptual model for diffserv routers.* Internet Draft, June 1999.