SANDIA REPORT

SAND2001-3138
Unlimited Release
Printed October 2001

Final Report for the Quality of Service for Networks Laboratory Directed Research and Development Project

Rose Tsang, John M. Eldridge, Thomas D. Tarman, Joseph P. Brenkosh, John D. Dillinger, John T. Michalski

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

Final Report for the Quality of Service for Networks Laboratory Directed Research and Development Project

Rose Tsang, Security and Networking Department, Dept 8910

John M Eldridge, Advanced Networking Integration, Dept 9336

Thomas D. Tarman, Advanced Networking Integration, Dept 9336

Joseph P. Brenkosh, Advanced Networking Integration, Dept 9336

John D. Dillinger, Networked System Survivability & Assurance, Dept 6516

John T Michalski, Networked System Survivability & Assurance, Dept 6516

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0785

Abstract

The recent unprecedented growth of global network (Internet) usage has created an ever-increasing amount of congestion. Telecommunication companies (Telco) and Internet Service Providers (ISP's), which provide access and distribution through the network, are increasingly more aware of the need to manage this growth. Congestion, if left unmanaged, will result in a degradation of the over-all network. These access and distribution networks currently lack formal mechanisms to select Quality of Service (QoS) attributes for data transport. Network services with a requirement for expediency or consistent amounts of bandwidth cannot function properly in a communication environment without the implementation of a QoS structure. This report describes and implements such a structure that results in the ability to identify, prioritize, and police critical application flows.

1. Introduction

The area of quality of service for networks has been in the incubation period for the past few years. It is apparent that protocols and standards policies are needed to define the ways in which network resources are allocated on the local and global networks. The IETF (Internet Engineering Task Force) has been the vanguard in its effort to draft RFC's in this area of QoS for networks. Our conducted research explores some techniques that can be utilized to develop a comprehensive QoS network that consists of three areas: the application stream, the marking and network level control of this stream, and finally the identification and reservation of appropriate network resources for the stream. All three areas of interest were studied and developed with the intent to provide an integrated working model.

2. Acronyms and Abbreviations List

CBQ: Class Based Queuing.
DiffServ: Differential Services.
DSCP: Differentiated Services Code Point.
IETF: Internet Engineering Task Force.
PHB: Per-Hop Behavior. Forwarding behavior performed by interior routers
RSVP: Resource Reservation Protocol (see section 4.1.x for details).
TOS: Type of Service. This is a byte in the IP header set aside for specifying the QoS a packet should receive.
QoS: Quality of Service.
WFQ: Weighted Fair Queuing.
COPS Common Open Policy server
IGMP Internet Group Management Protocol
RLM Receiver Layered Multicast
RED Random Early Discard
FIFO First In First Out
PBS Portable Batch System
CPU Central Processing Unit
LDAP Light weight Directory Assistance Protocol
PEP Policy Enforcement Point
PDP Policy Decision Point
TCP Transport Control Protocol
Vic Video Conferencing
AF Assured Forwarding
EF Expedient Forwarding
SCADA Serial Control and Data Acquisition
PCM Pulse Coded Modulation
FTP File Transfer Protocol

3. Motivation for Research

The motivation for research into this area of quality of service for networks was to show that the proper approach to management of network resources could be accomplished without resorting to the need for constant bandwidth upgrades. The constant need to continually provide larger and larger amounts of bandwidth for all bandwidth hungry applications was neither cost effective or practical. Through this research we attempt to show that management of network resources (bandwidth) can be implemented within a framework of control that will allow the distribution of a finite amount of network bandwidth through a priority approach to its distribution. This allows network managers (people) to take back control of network assets instead of the applications that reside on the network.

4. Description of Research

The conducted research can be divided into three areas of exploration: the application stream, the marking and network level control of this stream, and finally the identification and reservation of appropriate network resources for the stream. All three areas of interest were studied and developed with the intent to provide an integrated working model.

The application stream is developed from a NTSC video source, which is encoded by a PVH codec implemented in software. It is the manipulation of this PVH code that creates a resulted hierararcal encoded video stream. This video stream provides the data flow for the marking and control portion of the model. The marking and control function are resident on the network nodes and consist of an operating system stack that can be manipulated to provide the bandwidth allocation for all users of the network. This along with a centralized policy management node provides the basis for the development of an edge-to-edge quality of service protocol.

4.1 Technology Overview

4.1.1 RSVP

A resource reservation protocol is responsible for signaling the QoS requirements of a flow between the flow's end systems and all routers along the path between the end systems. The specification for the Internet resource reservation protocol is RSVP (Resource Reservation Protocol) [Braden]. RSVP is a soft-state protocol, requiring that the reservation of each flow be periodically refreshed. The reservation state is cached at each router along the path and must be periodically refreshed. If refresh messages are not received, the reservations time out and are automatically dropped, releasing the bandwidth. As can be inferred, maintaining a connection for each flow requires a lot of maintenance and state information for networks with a large number of flows.

4.1.2 Differentiated Services

DiffServ (Differentiated Services) are intended to provide scaleable QoS in the Internet without the need for maintaining per flow state information or doing per flow signaling. In a nutshell, DiffServ works in the following manner: Edge routers mark the TOS byte of each packet at the network edges or administrative boundaries. At the interior of the network, core routers provide per-packet QoS by examining each packet's TOS byte and assigning it a pre-specified per-hop behavior or forwarding behavior. See Section 4.2.2 for more details.

4.1.3 Common Open Policy Service (COPS)

COPS is a simple query and response protocol used in a client/server model that is used to exchange policy information between a policy server (Policy Decision Point (PDP)) and its clients (Policy Enforcement Points (PEPs)). There are two IETF RFCs which pertain to COPS: RFC 2748 which describes the COPS protocol, and RFC 2749 which presents COPS usage for RSVP. There are also many IETF Internet-Drafts further describing COPS and also discussing COPS usage for Policy Provisioning (COPS-PR), Differentiated Services (DiffServ), Optical Networks (OSDI) and others. Vendors such as Cisco Systems are now marketing products which support the COPS protocol.

4.2 System Overview

4.2.1 Hierarchical Encoding

In group communication environments where a transmitting source needs to disseminate information to many receiving nodes over a large displaced network, standard uni-cast forms of transmission become inefficient. Multicasting can be used to replace the standard uni-cast mode of delivery. Multicasting based on the IGMP (Internet Group Management Protocol) provides an efficient means to distribute the information. Instead of multiple individual uni-cast sessions originating from the source and traversing the network, and in many cases sharing the same paths, a single stream can replace all the multiple streams and save precious network bandwidth. However because of the many different link capacities that may exist in any given network, a single multicast stream cannot always provide the best service to all receiving nodes. If the transmitting source is providing a high capacity stream to all receiving nodes, the participants that are attached

to low bandwidth links will experience high packet loss, which will cause a degradation of quality. On the other hand if the transmitting source adapts the transmitting stream to support the receiving node's low bandwidth, the end nodes that have high bandwidth link capacities will suffer. Standard multicasting cannot overcome these problems. Layer multicast has been designed to overcome the differences of link capacities found in most networks. In a layer multicast approach the source transmits layered data streams through multiple addressed channels. By subscribing only to the addressed channels that the receiving node can accommodate, the receiver can dynamically adjust to its own capacity. A form of this approach is called Receiver-driven Layered Multicast (RLM).

RLM works in conjunction with the Internet Protocol (IP) architecture and relies on the delivery mechanism of IP multicast. It does not utilize real-time traffic guarantees and presumes only best effort, multipoint packet delivery. A key feature of IP multicast is the level of indirection provided by the host group. Host groups provide a group-oriented communication structure where senders do not need to know specifically about receivers, and receivers do not need to know the end-node specifics of the transmitters. When a transmitting node wants to send out information to multiple receiving nodes it simply transmits its information to a “group address”. Receivers tell the network using the Internet Group Management Protocol that they wish to receive information sent to a particular group address and thus the network relays the information to the appropriate requesting segment. The process, in which these receivers join and leave multicast groups, is much more efficient then using individual connection request to the originating information source.

To create the streams of layered information used in the RLM approach, which is eventually delivered throughout the network, (using IP multicast) a layered codec is utilized. The input signal from the video source is compressed into a number of discrete layers that are arranged in a hierarchy that allows for progressive video refinement. For example, if multiple layers are encoded at the source and only one of these layers is received, then the receiving decoder will produce the lowest quality of signal. If the decoder receives a second or third layer it will combine information from the previous layers to produce improved video quality. In general, the quality of information will improve with the number of layers that are received and processed. Figure 1, on the following page, shows the layered codec approach.

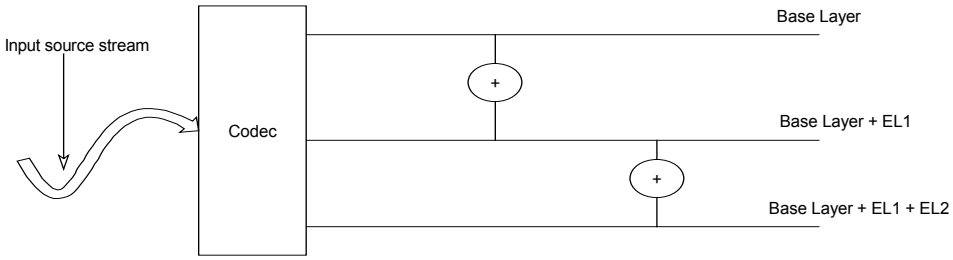


Figure 1

To be able to associate a group address to a specific stream the source distributes a hierarchical signal by striping the different layers across multiple multicast groups, and receivers adjust their reception rate by simply joining and leaving multicast groups. Now that the originating source has divided the over-all deliverable bandwidth into separate but correlated streams, how does the network know if the requesting receiver has the bandwidth resources available on its end segment or even has authorization to use the needed bandwidth?

In receiver-driven layered multicast the receiver is responsible for determining the optimal level of reception by using slow start congestion avoidance algorithm. It subscribes to layers until congestion occurs and backs off to an operating point below the congestion. The receiver determines the network is congested by observing gaps in the sequence space of the inbound packet stream. It is a more difficult task to determine when network bandwidth is available. If only passive monitoring of the inbound packet stream is used then it is not possible to determine if the receiver is using just the right amount of bandwidth or if it is greatly under utilizing network bandwidth. To compensate for this, RLM uses an active-join mechanism that continually subscribes to the next enhancement layer and then monitors for congestion. If it encounters congestion, it exponentially backs off on its request to join the group that contains the next enhancement layer. Figure 2 shows the back-off strategy that is used at the receiver of a host receiving up to three layers.

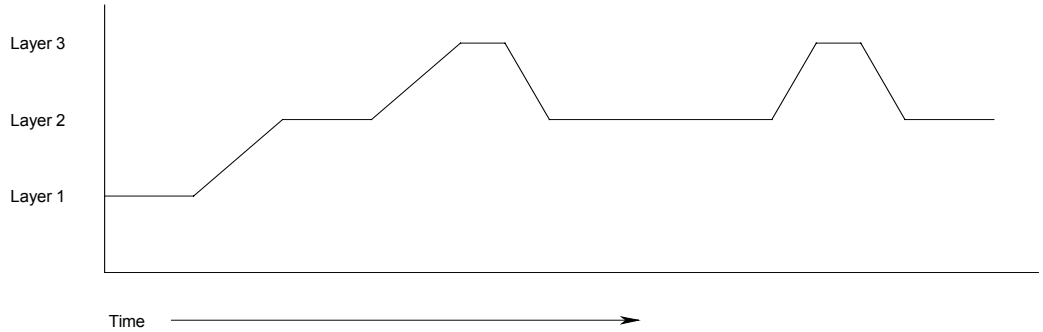


Figure 2

Initially the receiver subscribes to the first layer then monitors for congestion. After no congestion is detected it subscribes to the second layer then again monitors for congestion. With no congestion detected with the addition of the second layer it attempts to join the third layer. After subscribing to the third layer the receiver encounters congestion. It then responds by dropping its subscription to the third layer while continually monitoring the receiver's incoming packet stream for congestion. After a predetermined time has elapsed the receiver again tries to subscribe to the third layer. Again encountering congestion, the receiver once again reduces its subscription.

The implementation of the hierarchical streams can apply to more than just video applications. Imagine a textual-based message that needs to reach all destinations regardless of the end segment bandwidth that is available. The baseline message can be sent to all end segments with an enhanced version (more detailed information) disseminated to nodes more capable and more important. The disadvantage of this receiver-driven model is the network does not participate in the guarantee of even the most basic subscription rate. There needs to be a mechanism that puts control of bandwidth allocation back into the network. It is possible to provide a hierarchical information stream that can be partitioned with quality of service attributes by the network. For these to occur, the network must have the ability to identify, qualify, and regulate flows.

4.2.2 Flow Identification and Queuing

As mentioned before, DiffServ provides a means for providing QoS in the Internet without the need for per-flow state and signaling in every router. By aggregating a large number of QoS-enabled flows into a small number of aggregates that are given a small number of per-hop or differentiated treatments within the network, DiffServ eliminates the need to recognize and store information about each individual flow in core routers.

Each DiffServ flow is identified (e.g., via IP source and destination address), policed and marked at the first edge router according to a contracted service profile. Downstream from the nearest edge router, a DiffServ flow is aggregated with similar DiffServ traffic from other flows to form an aggregate. All subsequent forwarding and policing are performed on aggregates.

Marked packets receive per-hop behaviors (PHBs) by interior routers. The IETF DiffServ Working Group has defined the following PHBs: (1) Expedited Forwarding – a simple high priority PHB, and (2) Assured Forwarding – a group of PHBs with different delay and drop priorities. Assured Forwarding PHB received higher priority than best-effort service but lower priority than Expedited Forwarding PHB.

The IETF was very careful about not defining the actual implementation of the PHBs. Thus a vendor may choose any queuing discipline as long as it provides the PHB as defined in the standard. The following are the most common queuing algorithms:

(1) Priority Queuing, (2) Weighted Fair Queuing (WFQ), and (3) Class-Based Queuing (CBQ).

The Priority Queuing algorithm creates multiple queues per port and assigns each queue a relative priority value. Devices that use Priority Queuing decide which queue to place traffic in by checking pre-configured mappings of DSCP markings to priorities. Priority Queuing then transmits traffic in high-priority queues before transmitting packets in lower-priority queues. Priority Queuing is obviously better at ensuring different levels of service for different types of traffic than FIFO queuing is. However, Priority Queuing gives the queue being serviced all available bandwidth. The potential result is that when used alone, Priority Queuing can starve lower-priority traffic of bandwidth by always servicing a high-priority queue despite traffic piling up in a low-priority queue. Routers, which want to avoid this starvation problem, must use other methods, such as WFQ.

Like Priority Queuing, WFQ creates multiple queues for different traffic classes and assigns each queue a relative priority value. However with WFQ, weight values are assigned to each queue in proportion to its level of priority. Queues are weighted to ensure that higher priority queues get a larger percentage of available bandwidth than lower priority queues get. The exact amount of bandwidth each queue actually gets depends on the number of queues sharing that bandwidth at a given moment. WFQ ensures that traffic classes always get some bandwidth; the specific rate is variable -- not guaranteed.

In contrast to Priority Queuing and WFQ, CBQ enables you to allocate a guaranteed bandwidth rate to each traffic class. Like Priority Queuing and WFQ, CBQ creates multiple queues for different traffic classes. Unlike Priority Queuing and WFQ, CBQ enables you to assign traffic classes guaranteed data rates. For example, if you allocate 56 kbit/s to a high-priority traffic class, CBQ ensures that that traffic class always gets 56 kbit/s. With CBQ you can also define parameters that enable devices to distribute additional bandwidth to traffic classes as bandwidth is needed. You can also ensure that traffic classes can burst above those guaranteed rates when necessary by (1) setting bits in the TOS octet at network edges and administrative boundaries, (2) using those bits to determine how packets are treated by the routers inside the network, and (3) conditioning the marked packets at network boundaries in accordance with the requirements of each service.

The DiffServ architecture is composed of a number of small functional units implemented in the network nodes. This includes the definition of a set of Per-Hop Behaviors (PHBs), packet classification and traffic conditioning functions like metering, marking, shaping and policing. The resource allocation for each service type adds a new dimension to the problem, for which the Bandwidth Brokers are being considered. The DiffServ model is scalable and suggests that the more expensive tasks like multi-flow classification, policing, shaping and marking be done at the border routers of the ISP networks. This is because the border routers deal with the customer links that are slow as a result of doing the costly functions like MFC and traffic conditioning. The core routers, on the other hand simply do the forwarding based on the DiffServ code point (DSCP),

which is the first six bits in the TOS byte in the IP header. Since the core routers need not maintain any per-flow state, this model is more scalable. The granularity of service provisioning is a class in *diff-serv*, as opposed to being a flow in *int-serv*. Multiple flows can then be mapped on to a single per-hop behavior, which allows for a more efficient use of the network control resources.

4.2.2.1 Linux traffic control

To integrate the DiffServ model into a laboratory environment to support this QoS effort there was a need to identify and acquire all appropriate hardware and software. See section entitled *System Test Methods* for a listing of supporting hardware. The operating system chosen to support our efforts was Linux OS version 2.2.14. This operating system has traffic control features built into its kernel. Most of all, the functionality needed to implement traffic control exists within the kernel. Some additional files are needed to patch the kernel to allow the ability to control features of the kernel from user space. The Linux kernel resides in /usr/src/linux; from there the tc program is stored in /iproute2/tc. The tc file is a utility in user space and is used to manipulate traffic control features in the kernel. Its source is in the file iproute2 that was retrieved from ftp://ftp.inr.ac.ru/ip-routing/. Most of the kernel code can be found in /net/sched under the linux path. The interface between the kernel traffic control functions and user space programs is declared in include/linux/pkt_cls.h and include/net/pkt_sched.h. The rtnetlink socket interface used for communications between the linux kernel and the user space programs is implemented in net/core/rtnetlink/.c and include/linux/rtnetlink.h. The rtnetlink is based on netlink which can be found in net/netlink/ and include/linux/netlink.h. The kernel source can be found from www.kernel.org. The diffserv patch can be found at http://lrcwww.epfl.ch/linux-diffserv. The iprout2 program can be found at http://defiant.coinet.com/iproute2. Figure A shows the relationship between the Linux kernel and the associated user space where the tc files, that are use to manipulate the traffic control attributes in the kernel, are located. The interface that provides the communication between the two components is the netlink socket.

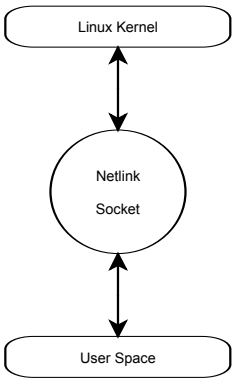


Figure A

The manipulation of traffic flow within our lab setup can be grouped generically into one of two locations; the ingress of the network, and the interior or "core" of the network. The actual implementation of traffic control in Linux is implemented at the output port of the network device. The processing of network data starts at the input port where the transmitted frame is examined to determine if it is addressed to the Linux device. If the address of the frame matches the associated address of the Linux device it is retrieved and its IP address is examined. If the IP address is associated with the Linux device itself, it is processed up the stack. If the address is foreign, Linux examines the forwarding table to determine to which port this packet needs to be forwarded, and then sends it to the appropriate output port. Figure B shows the packet processing that is implemented in the Linux devices.

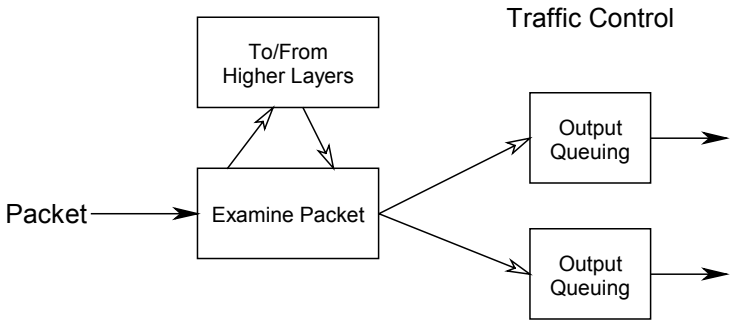


Figure B

It is on this output port where the traffic control features decide on how the packet will be processed. The traffic control features for Linux can be broken down into the following parts:

- Queuing disciplines
- Classes
- Filters
- Policing

Queuing disciplines

Each output port has an associated network device, and each network device has a queuing discipline associated with it. The queuing discipline controls how each packet is handled or treated. A very simple approach to queuing packets would use a First In first Out (FIFO) buffer scheme. The FIFO queuing scheme stores packets as they are received and then sends them out onto the line at the operating speed of the network device. Linux supports multiple queuing disciplines including RED, SFQ, TEQL, TBF, GRED, WFQ,

and simple FIFO. Many different types of packet manipulation can occur depending on which queuing discipline is chosen. Each queuing discipline provides the following set of functions to control its operation

- Enqueue: Associates a packet with the queuing discipline. If the queuing discipline has classes, the enqueue function first selects a class and then invokes the enqueue function of the corresponding queuing discipline for further enqueuing.
- Dequeue: Returns the next packet eligible for sending. If the queuing discipline has no packets to send dequeue returns null.
- Requeue: Puts a packet back into the queue after it has been dequeued by dequeue. This differs from enqueue because the packet should be queued at exactly the place from which it was removed by dequeue, and it should not be included in the statistic of cumulative traffic that has passed the queue. That was already done in the enqueue function.
- Drop: Drops one packet from the queue.
- Init: Initializes and configures the queuing discipline.
  Reset: Returns the queuing discipline to its initial state. All queues are cleared timers are stopped. Also the reset functions of all queuing disciplines associated with classes of this queuing discipline are invoked.
  Destroy: Removes a queuing discipline. It removes all classes and filters, cancels all pending events and returns all resources held by the queuing discipline.
- Dump: Returns diagnostic data used for maintenance.

Classes

The term Classes is used to describe how packets will be treated. Classes do not manipulate the packets themselves; it's the queuing discipline or disciplines associated with the class that determines the handling. Queuing disciplines and classes are bound together. The occurrence of a class or classes within queues is fundamental property of the queuing discipline. Classes can be identified in two ways - by the class ID, which is assigned by the user, and by the internal ID which is assigned by the queuing discipline. Class IDs are structured like queuing discipline IDs, with the major number corresponding to their instance of the queuing discipline, and the minor number identifying the class within that instance. Queuing disciplines with classes provide the following set of functions to manipulate classes.

- Graft: Attaches a new queuing discipline to a class and returns the previously used queuing discipline.
- Get: Looks up a class by its class ID and returns the internal ID. If the class maintains a usage count, *get* should increment it.
- Put: Is invoked whenever a class that was previously referenced with *get* is de-referenced. If the class maintains a usage count, *put* should decrement it.
- Change: Changes the properties of a class and it is also used to create new classes; where applicable some queuing disciplines have a constant number of classes which are created when the queuing discipline is initialized.

- Delete: Handles requests to delete a class. It checks if the class is not in use, and de-activates and removes it in this case.
- Walk: Iterates over all classes of a queuing discipline and invokes a callback function for each of them. This is used to obtain diagnostic data for all classes of a queuing discipline.
- Tcf_chain: Returns a pointer to the anchor of the list of filters associated with a class. This is used to manipulate the filter list.
- Bind_tcf: Binds an instance of a filter to the class. Identical to function *get* except when the queuing discipline needs to be able to explicitly refuse class deletion.
- Unbind_tcf: Removes an instance of a filter from the class. Usually identical to the function *put.*
- Dump_class: Returns diagnostic data.

Classes are selected in the enqueue function of the queuing discipline. Usually be invoking the function tc_classify in include/net/pkt_cls.h.

Filter

The term Filter describes the mechanism that is deployed to assign incoming packets to one of the classes. Filters can be ordered by priority in ascending order and act upon the incoming data stream by directing the appropriate matched packets to their intended class destinations. Filters are kept in filter lists, which can be maintained by the queuing discipline, or per class, depending on the design of the queuing discipline. Because of their classifying function, they are also referred to as classifiers. In our approach a Route classifier (on the ingress of the network) is used to filter out and match source IP addresses on each incoming packet. Filters vary in the scope of packets their instances can classify. When using the cls_fw and cls_route filters, one instance per queuing discipline can classify packets for all classes. The other types of filters, cls_rsvp and cls_u32, need one or more instances of the filter or its internal elements per class. These filters are called specific filters. Unlike classes, filters have no filter ID; instead they are identified by the queuing discipline or class for which they are registered. Because specific filters have at least one instance or element per class they can store the internal ID of that class and provide it as a result of classification. This allows for quick retrieval of class information by the queuing discipline. Filters are controlled by the following functions:

- Classify: Performs the classification and returns one of two values. It can return the selected class ID and optionally also the internal class ID in the struct tcf_result.
- Init: Initializes the filter.
- Destroy: Is invoked to remove a filter. Also the queuing disciplines sch_cbq and sch_atm use this function to remove stale filters when deleting classes.
- Get: Looks up a filter element by its handle and returns the internal filter ID.
- Put: Is invoked when a filter element previously referenced with the function *get* is no longer used.

- Change: Configures a new filter or changes the properties of an existing filter. Configuration parameters are passed with the same mechanism as used for queuing disciplines and classes.
- Delete: Deletes an element of a filter. To delete the entire filter the function *destroy* is used.
- Walk: Iterates over all elements of a filter and invokes a callback function for each of them. This is used to obtain diagnostic data.
- Dump: Returns diagnostic data for a filter or one of its elements.

Policing

After the packets have been filtered they are metered or policed. The term policing refers to the control mechanism within a defined queue that allows the queue to regulate its traffic flow in a described manner. How the traffic flow (packets) are regulated is determined on how the class was defined. For example, if a class is defined to only use bandwidth that is available at the output of a network device, it is possible that some packets will be discarded to remain within this class description (this is a form of policing or traffic shaping). There are four types of policing mechanisms: policing decisions by filters, refusal to enqueue a packet, dropping a packet from an inner queuing discipline, and dropping a packet when enqueuing an new one. The first type of action is decisions made by the filters. The classifying function of a filter can return three types of values to indicate a policy decision:

- Tc_police_ok No special treatment requested
- Tc_police_reclassify Packet was selected by filter but it exceeds certain bounds and should be reclassified
- Tc_police_shot Packet was selected by filter and found to violate the bounds such that it should be discarded.

Currently the filters cls_rsvp, cls_rsvp6, and cls_u32 support policing. The policing information is returned via tc_classify located in include/net/pkt_cls.h to the enqueue function of the queuing discipline. It is then the responsibility of the queuing discipline to take appropriate action.

The packets are then marked by inserting a label in the type of service (TOS) field in the IP header. This mark will subsequently provide the means of identifying the packet as it transverses the interior of the network until it reaches its destination. Figure 3 shows the associated steps that occur on the ingress of the network to initiate the quality of service for each incoming packet.

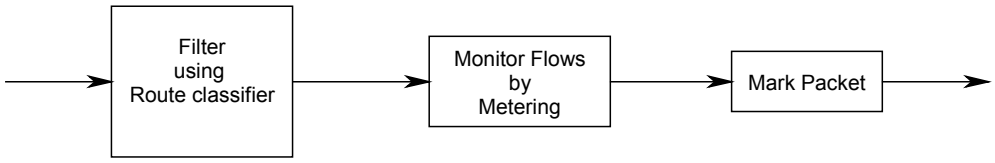


Figure 3

After the packets are classified and marked on the edge of the network they are free to continue until they reach their final destination. While they are traversing the interior of the network, the handling or treatment will be based on the mark that was transcribed at the ingress of the network. The classification or filtering of the packets and the assignment into classes within a queuing discipline is now solely dependent on the mark. Each mark will have an associated class assigned within a queuing discipline that will determine the type of service each packet will be given. A very simple queuing discipline may consist of a single queue such as a FIFO where all packets are stored in the order in which they are queued. More granular queuing disciplines use filters to distinguish among different classes of packets, and process each class differently based on priority. Queuing disciplines and classes are associated with each other. Filters, on the other hand, can be combined with any queuing discipline as long as the queuing discipline has a defined class to map the packets too. Normally each class owns a queue but it is also possible that several classes can share the same queue. There can be associated priorities within a class queuing discipline that allows another level of granularity control. This traffic control process, which is found on the interior network devices, is shown in figure 4.

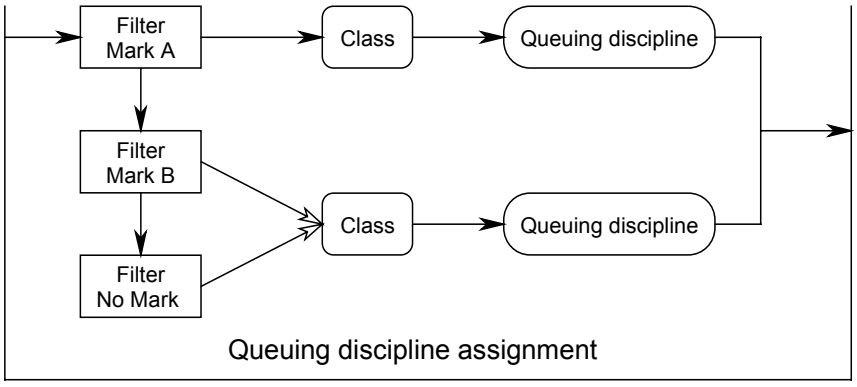


Figure 4

4.2.3 Policy Management

Integrating Distributed Resource Management with Network QoS

For some applications, "best effort" datagram delivery is not enough for proper functioning of the application. Datagram loss or mistiming can result in serious degradation of application performance. Therefore, a mechanism is required where the network can provide Quality of Service (QoS) guarantees to these select flows. However, when QoS is assigned to some flows, it is at the expense of other flows. Therefore, QoS must be carefully assigned according to the network usage policy. This policy describes how network resources are allocated to users, hosts, and applications depending on such factors as current network load, time of day, importance or priority of the service, and the existence of an advance reservation for the flow.

An advance reservation provides an assurance to a requestor that at the time the flow is to be established, the network can provide the required QoS for that flow. This is not unlike reservations that are commonly made when scheduling computing jobs. In this case, however, network bandwidth is being reserved instead of compute nodes.

Due to the similarity between bandwidth reservation and distributed computing reservation, a distributed resource management (DRM) package was selected as a basis for investigating advance and priority reservations for network QoS. The package that was selected for this project is the Portable Batch System (PBS), which is available from MRJ Inc**.** http://pbs.mrj.com. This package was selected because it is well supported, in use at many sites, and available in source (so that we can modify it to support management of network bandwidth).

PBS is designed primarily for reserving nodes in multiprocessor systems or distributed computing systems. It consists of three major entities – the server, the scheduler, and the "moms". All jobs that are submitted to the system are submitted to the server, which creates the batch job. When a job is submitted to the server, the server submits the batch job to the scheduler, which decides when to run it. This decision is based on local policy and the availability of resources at the server's site. The scheduler determines the availability of nodes by maintaining contact with the PBS mom processes that run on each node. When a job is selected to run, mom is also responsible for actually running it on her node.

The relationship between these components is roughly as follows: one server per distributed system, one scheduler per site, and one mom per CPU or node. When PBS is integrated with the standard network QoS mechanisms of RSVP and COPS, it will work as follows:

The user will use the "xpbs" front-end to instruct the PBS server that he wishes to reserve bandwidth for an application. The PBS server or scheduler will check with the list of existing advance reservations and possibly consult a lightweight directory assistance protocol (LDAP) database (containing network topology and QoS state information) to

determine whether the reservation request can be honored. The PBS server/scheduler will also determine if the user is authorized to 1) make the reservation, and 2) be granted the requested QoS. In this fashion, PBS acts as a QoS policy manager. When the time comes to allocate the bandwidth that was reserved earlier, one or two mechanisms may be used to allocate it:

Use RSVP (signaling only). In this mode, the Policy Enforcement Engines (PEPs, to use COPS terminology) will consult the Policy Decision Point (in this case, the QoS policy manager, or the PBS server/scheduler). This method will probably not be implemented.

The PBS scheduler contacts the Linux routers (and possibly switches) to use DiffServ to prioritize the flow`s packets. The protocol for this may be the standard PBS server-mom protocol, or it could be a custom protocol. This method is preferred.

Once the bandwidth is allocated, confirmation is sent to the user who reserved the bandwidth, and to the application that will use the bandwidth. Upon receipt of confirmation, the application traffic flow will begin.

4.3 Common Open Policy Service (COPS)

COPS is a simple query and response protocol used in a client/server model that is used to exchange policy information between a policy server (Policy Decision Point (PDP)) and its clients (Policy Enforcement Points (PEPs)). There are two IETF RFCs which pertain to COPS: RFC 2748 which describes the COPS protocol, and RFC 2749 which presents COPS usage for RSVP. There are also many IETF Internet-Drafts further describing COPS and also discussing COPS usage for Policy Provisioning (COPS-PR), Differentiated Services (DiffServ), Optical Networks (OSDI) and others. Vendors such as Cisco Systems are now marketing products which support the COPS protocol.

The basic COPS model consists of at least one client (PEP) and one server (PDP). In most cases, there would more likely be several PEPs being serviced from a single PDP, but there could also be more than one PDP. This would permit another PDP to act as a backup to the primary PDP. There may optionally be a local PDP (LPDP) which would reside on the same host as the PEP. This would be used to execute local policy decisions in the event the remote PDP(s) were unreachable.

4.3.1 COPS Models

There are two basic models of COPS, outsourcing and provisioning. The outsourcing model is essentially a client/server approach. It addresses events at the PEP which require instantaneous policy decision. The PEP, being aware that it must perform a policy decision, delegates the responsibility to the PDP for policy information. The PDP

responds to incoming policy requests by the PEP. All requests are in the form of COPS data objects. This model works well with RSVP.

The provisioning model is a server-driven approach where the PDP pre-provisions the PEPs beforehand. It makes no assumptions of a direct 1:1 correlation between PEP events and PDP decisions. In the provisioning model, after the PEPs identify themselves to the PDP, the PDP downloads the policy information to the PEPs. This information may be in bulk, such as an entire router configuration or in portions (e.g. updating a DiffServ marking filter). If changes occur in the PEP in ways not covered in policies already known to the PEP (such as an interface removal), the PEP sends this unsolicited new information to the PDP. Once this information is received, the PDP sends to the PEP any new policies needed by the PEP.

In both models, the PEP is responsible for initiating a persistent TCP connection to a PDP. The PEP uses this TCP connection to send requests to and receive decisions from the PDP. The PEP is able to report to the PDP that it has successfully performed the PDP`s request. This is useful for monitoring and accounting purposes. The PEP is responsible for notifying the PDP when a request state has changed on the PEP.

The COPS protocol involves sending messages between the PEP and PDP. One PDP implementation per server must listen on a well known TCP port number, i.e. 3288. Each message consists of a COPS header followed by typed objects. The header consists of the following fields. Each octet is 8 bits in length. The following illustration describes the header:

Octet 0	**Octet 1**	**Octet 2**	**Octet 3**
Version and Flags	Op Code	Client-type	
Message Length			

Figure 1. The COPS header.

Version: 4 bits in length. The current version is 1.
Flags: 4 bits in length. This flag is set when the message is solicited by another COPS message.
Op Code: 1 octet in length. Typical operations are Requests, Report State, Client Open, Client Close, etc.
Client-type: 2 octets in length. Identifies the policy client.
Message Length: 4 octets in length. This field describes the number of octets (including the header) that compose the object. Messages must be aligned on 4 octet intervals.

All COPS objects follow the same format. This includes a four octet header followed by one or more 32 bit words. Objects are in the following format:

Octet 0	Octet 1	Octet 2	Octet 3
Length (octets)		C-Num	C-Type
Object Contents			

Figure 2. A COPS object.

Length: 2 octets in length. Contains the number of octets including the header, that compose the object.
C-Num: 1 octet in length. This field identifies the subtype or version of the information contained in the object. Some typical values for the C-Num field are:

> 1 = Handle Object. This object contains a unique value that identifies an installed state. It must be unique from other client handles from the same PEP. It is initially set by the PEP and is used to identify a particular request for a Client-type.
> 2 = Context. This denotes the type of events that triggered the query. This is required for request messages. Some request types are: Incoming Message/Admission control and Resource-Allocation.
> 3 = In Interface. This identifies the incoming interface on which a particular request applies and also the address where the received message originated.

The provisioning model uses a set of policy rules. A named data structure known as a Policy Information Base (PIB) is used to identify the type and purpose of unsolicited policy information that is sent to the PEP for provisioning policy. The PIB name space is common to both the PEP and the PDP, and names within this space are unique within the scope of a given PDP/PEP/Client-type communication channel. The PIB is similar to an SNMP MIB and is based on the ASN.1 data definition language. In order to simplify coding and allow reuse of SNMP encoding/decoding code, the wire representation of the policy information that is encoded in the COPS protocol objects follows the Basic Encoding Rules.

4.3.2 COPS Security

Security in COPS is negotiated once per session during the initial PEP (client) open/accept. The first Client-Open request must contain a PEPID and Integrity object which will contain the initial sequence number the PEP requires the PDP to increment during communications after the initial Client-Open/Client-Accept. The Integrity object will also contain the Key ID identifying the algorithm and security key used to compute the digest.

If the PDP accepts the security key and algorithm from the PEP, the PDP must send a Client-Accept message with a Client-Type=0 with an Integrity Object. This object will contain the initial sequence number the PDP requires the PEP to increment during all further communications with the PDP, and the Key ID identifying the algorithm and key used to compute the digest. If the PEP from the perspective of the PDP sends an invalid Integrity object, the PDP must send the PEP a Client-Close message with the appropriate

error code. Conversely, if the PDP from the perspective of the PEP sends an invalid Integrity object, the PEP must send the PDP a Client-Close message with the appropriate error code. All COPS implementations must support the HMAC-MD5-96 cryptographic algorithm for computing the digest of the Integrity object which is appended to the message. COPS can also reuse existing protocols such as IPSEC for security.

4.3.3 COPS Operation

A COPS session would begin with one or more PEPs (clients) connecting to a PDP (server). Once connected, the PEPs can request provisioning/policy information from the PDP if the outsourcing model is being used. If the provisioning model is being used, the PDP will provision the PEPs without being requested by the PEPs to do so. The PDP may change provisioning information if conditions require. The PEP may notify the PDP on the status of an installed state using a Report message to signify when billing can begin or to produce periodic updates for monitoring. The PEP must send periodic Keep-Alive messages to the PDP. After receiving the message, the PDP must echo it back to the PEP. If either of these do not occur, the connection is considered lost. When the PEP detects a lost connection, it should send a Client-Close message for each opened client type specifying a communications failure error code.

4.3.4 COPS Prototype

Due to the lack of available public domain COPS implementations, a prototype COPS implementation was developed to support this LDRD. The implementation is written in Perl and is a modified version of the Client/Server example in the *Learning Perl* book written by Randal L. Schwartz. Although not a "true" COPS implementation, it provides some extra features, which are useful in the QOS LDRD, implementation and debugging phases. These features include:

- User specified ports, not just port 3288.
- Output file creation on the PDP containing the output of the policy commands executed on each PEP.
- File locking to prevent "runaway" conditions while testing.
- Ability to set or check non COPS parameters on each PEP from the PDP.

In the prototype, the PEP is the server and the PDP is the client. This is just the opposite of how it was described earlier. The functions of the PEP and PDP have not changed, however. The PEP executes "COPS like" policy commands it receives from the PDP. The PDP can support multiple PEPs. The prototype operates using the provisioning model and would need to be modified to operate using the outsourcing model.

5. System Test Methods

The intent of this section is to define the configuration needed to demonstrate all the components that are required to implement a policy based QoS queuing scheme. The lab setup, which was used to test all the components of this QoS effort, is shown in Figure 1, on the following page.

The integration effort to create a network quality of service model consists of three parts. The first part is comprised of the chosen application that will be used to compare and contrast the assigned network QoS attributes. The second part is made up of the filter and queuing scripts located on the network nodes (Linux devices) that will perform the classification and priority queuing of the network traffic. And the third part comprises the policy manager, which will assign the traffic profiles for applications that will transverse the network.

5.1 Application and Traffic Generation

Any application flow including the hierarchical video conferencing utility called "vic", which was previously discussed in the *Technology overview section*, can be used to demonstrate our QoS implementation. The origination of the application flow is labeled "Data Src" as shown in Figure 1. When the application "vic" is being used each hierarchical stream could be identified with a different resulting priority. This would allow, for example, all base frames and the associated enhancement layers to be transmitted to all end nodes during times of no network congestion. This would result in the highest clarity of reception. But during times of congestion, only the base frame, with the highest level of priority would be allowed to reach its destination in a minimally delayed manner. All other enhancement frames, depending on traffic conditions, would be delayed or discarded. To create network congestion, a large amount of traffic needs to be generated and sent across our lab demonstration network. In the lab setup, a traffic generating application called TTCP was used. TTCP allows for an operator to select the rate of flow of traffic and the target host. It then blasts out this high-speed stream from its source to its directed destination. This creates an over-subscription of bandwidth at eth0 located on Router 1 node shown in Figure 1. This condition results in queuing being initiated on eth0 by the Linux Operating System which comprises Router 1. Linux Bandwidth management uses "user defined" queuing scripts that are assigned to the device driver on each output port of the Routers. These queuing scripts are described in the following section entitled *5.2 Queuing Scripts*.

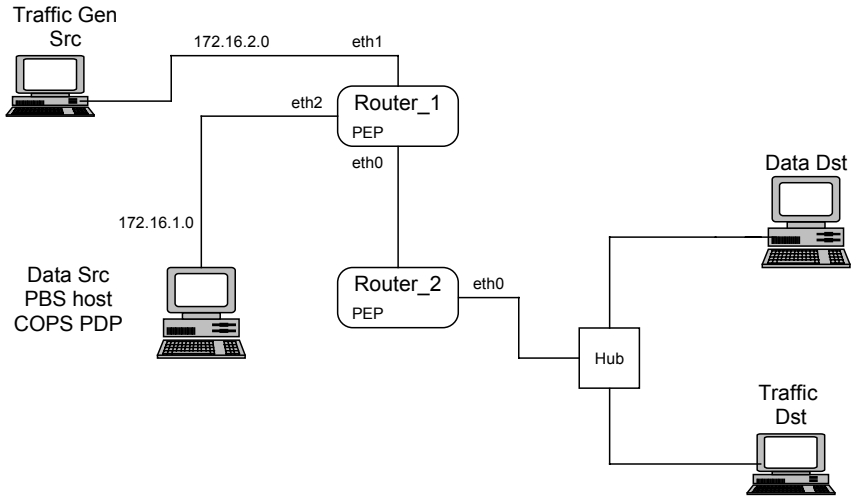


Figure 1

5.2 Queuing Scripts

The implementation uses Class Base Queuing (CBQ) to set the priorities for each class flow. There are two primary scripts, one located at the ingress of the network and the other located in the core. The ingress script uses the classifier cls_u32 which handles all micro flow classification tasks. It tags micro flows into classes. The marking of the packets is done when the packet is dequeued from sch_dsmark. This function uses skb>tc_index as an index to a table in which the outbound differentiated services code point (DSCP) is stored and puts this value into the packets type of service field (TOS). It is possible for an application on a source workstation to mark packets when they are generated, using the IP_TOS socket option available on Unix machines including Linux. This option is not used in this configuration because it weakens the control of the Policy manger to dictate centralized priority assignments. This implementation of DiffServ requires the addition of three new traffic control elements to the kernel: the queuing discipline sch_dsmark to extract and to set the DSCP, the classifier cls_tcindex which uses this information, and the queuing discipline sch_gred which supports multiple drop priorities

The ingress script located on Router #1 uses queuing discipline CBQ to implement class priories. The enqueue function of this queue scans the filters until one of them indicates a match to a class identifier. The filters in this case are setup as route classifiers that look at the IP source address of each packet. When a match occurs it queues the packet for the corresponding class. Packets that do not match any of the filters are assigned into a default class. The dsmark queue discipline is then responsible for marking the type of service (TOS) field in the IP header. A u32 classifier is then used to assign a priority to

this class, which is mapped into a flow. Each individual class will have the same ID for its associated flow. The script below illustrates this functionality and is located on Router #1 (the ingress network device) as seen if Figure 1.

Edge Device "Ingress buffer"

*Setup the "dsmark" queue-discipline and assign it the Ethernet interfaced 0

1) tc qdisc add dev eth0 handle 1:0 root dsmark indices 64

*Declare two classes of dsmark EF, and AF21. Mask first two bits

2) tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
3) tc class change dev eth0 classid 1:2 dsmark mask 0x3 value 0x18

*Setup u32 classifier and priority for EF flow

4) tc filter add dev eth0 parent 1:0 protocol ip prio 4 handle 1: u32 divisor 1

*Setup u32 classifier and lower priority for AF21 flow

5) *tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 2: u32 divisor* 1

*Assign the classifier and priority flow rate to the EF class (do not include metering)

6) tc filter add dev eth0 parent 1:0 prio 4 u32
match ip src 172.16.1.0/24
flowid 1:1

*Assign the classifier and priority flow rate to the AF class

7) tc filter add dev eth0 parent 1:0 prio 5 u32
match ip src 172.16.2.0/24
flowid 1:2

The above Ingress filter queue attaches a dsmark queue to the interface eth0 on the root node as seen in line 1. The second and third lines instruct the dsmark queue to declare two classes of flows an EF (Expedient Forwarding) high priority queue, and an AF (Assured Forwarding) lower priority queue. This is accomplished by masking out bits 6 and 7 then or-ing that with the value of 0xb8 and 0x18 respectively. The Fourth and Fifth lines set up classifiers for each flow and set the relative priorities of these flows. Lines 6 and 7 assign the classifier and priority flows to each class by matching source IP

addresses of incoming IP packets and assigning these packets into their respective flows. These flows then are mapped back into the previously defined EF and AF classes. The classifier on line 6 will map all packets with network source IP address of 172.16.1.0 into flow 1:1 and assigns it into the class 1:1. The classifier on line 7 will map all packets with network source IP address of 172.16.2.0 into flow 1:2 and assigns it into class 1:2. The result of this script is two different packet flows are identified, marked, and sent to queues where each is assigned a different priority for output buffering

The core script located on Linux #2 has the responsibility of examining every incoming packet to determine if it has a DSCP point (label) written into its` IP header. Then DSCP is then mapped into classes associated with the queuing discipline CBQ. The queuing discipline is then responsible for the prioritization of each packet flow. The script below illustrates this procedure.

Core Device "Egress buffer"

*Attach a root node to Ethernet port 0 and copy the TOS information into the tc_index

1) tc qdisc add dev eth0 handle 1:0 root dsmark indices 64 set_tc_index

*Mask out ECN bits and extract DSCP (differentiated services code point?) information

2) tc filter add dev eth0 parent 1:0 protocol ip prio 1 tcindex mask 0xfc shift 2

*Setup a Class base queue with a handle of 2:0 that will be a child of the root node 1:0. Set available parent bandwidth at 10 Mbits.

3) *tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq bandwidth 10Mbit allot 1514 cell* 8 *avpkt 1000 mpu 64*

*Set first child class to 1.5 Mbit with no sharing. This will be enough for VIC.

4) *tc class add dev eth0 parent 2:0 classid 2:1 cbq bandwidth 10 Mbit rate 1500Kbit avpkt 1000 prio 1 bounded isolated allot 1514 weight 1 maxburst 10 defmap 1*

*The following line assigns a buffer management scheme to the pre-defined class. What is the impact of not using buffer management?

5) tc qdisc add dev eth0 parent 2:1 pfifo limit 5

*Assign the EF flow to the 2:1 child class. The attributes for this class were assigned on line 4. The command pass_on allows TOS bytes not marked as EF to fall through.

6) ***tc filter add dev eth0 parent 2:0 protocol ip prio 1 handle 0x2e tcindex classid*** **2:1** ***pass_on***

*Set second child class to 5Mbit sharing which can be borrowed if not used (use it or lose it) TTCP gets this allocation

7) ***tc class add dev eth0 parent 2:0 classid 2:2 cbq bandwidth 10Mbit rate 5Mbit*** **avpkt** ***1000 prio 7 allot 1514 weight 1 maxburst 21 borrow***

*Use the same buffer management scheme as previous 2:1 class

8) tc qdisc add dev eth0 parent 2:2 pfifo limit 5

*Treat all other packets not marked with the TOS "EF" marking the same and assign them to class 2:2. The command handle 0 mask 0 is the "catch-all" default.

9) tc filter add dev eth0 parent 2:0 protocol ip prio 2 handle 0 tcindex mask 0 classid **2:2 pass_on**

Line 1 attaches to the root node on interface eth0 a dsmarker that copies the TOS byte into function set_tc_index. Line 2 adds a filter to the root node that masks out the ECN bits and extracts the DSCP field by shifting to the right two bits. This is needed because only 6 bits of the 8 bit TOS field in the IP header is used to create DSCP labels. A queuing discipline CBQ is attached to node 2:0 which is the child of the root node 1:0 assigned in line 3. Two child classes are then defined out of the 2:0 node. Child class 2:1 uses a Class base queue which is bound to a rate of 5.0 Mbps. A packer counting FIFO queuing discipline (pfifo) is attached to the CBQ as the buffer management scheme. Line 6 adds a tcindex classifier which will redirect all packets with a set_tc_index of 0x2e (the DSCP for EF service) to the classID 2:1. Packets that are not matched with this label are allowed to fall through so they can be matched with another filter. The pass_on command on line 6 allows this to happen. Line 7 defines another CBQ class 2:2 that is intended to be the "best effort" classes that the other markings by default will be treated. The rate is limited to 1.5 Mbps but has the option of borrowing unused bandwidth from the higher priority queue if available. This is accomplished by the operator borrow located at the end of line 7. Since the AF class does not allow sharing of its unused bandwidth as noted in the operator bounded isolated on line 4, this will limit the EF class to 5Mpbs less bandwidth to use even if the AF class is not using it.

5.3 Policy Manager

Another aspect of a complete quality of service system is the ability to define the applications on a network that will operate with the least amount of forwarding delay. Or described another way, a policy manager has a pre-defined profile of which applications on its network are granted the highest level of transport and those that receive a lower level of transport. The level of transport is defined as a function of usable bandwidth and forwarding delay. In our lab setup we used the Portable Batch System (PBS) as the framework in building the Policy Server. The PBS structure allowed for the ability to assigned pre-define traffic queues based on a "real-time" clock.

The communication between the policy manager and the network router nodes was through an interface as defined in the Common Open Policy Server (COPS) *section 4.3*. The COPS interface defines two components, the Policy Enforcement Point (PEP) located on each router node, and the Policy Decision Point (PDP) located on a host. The host is labeled "*PBS Host, COPS PDP*" in figure 1. The PBS code was hacked to provide a communications interface between itself and the locally resident PDP code. Communications between the PBS and the PDP code was through the /tmp directory. The PDP code would continually look for the presence of any files with a ".cops" extension handed down from the PBS. When the PDP found the job request handed down from the PBS it then would initiate a connection request through the use of a Perl programmable socket to the PEP or PEP's located on the network router nodes. A description of the Perl socket code can be found in the appendix of this document.

After the connection request is completed and a communication path is created the PDP informs the PEP which queuing script needs to be launched to create the desired priority queue for the requested application flow. The PEP then initiates the script or scripts for each appropriate output port and sends command output back to the PDP. The command output is then stored back into the "/tmp" directory so the PDP knows which queues are currently active. When a new script is initiated by the PBS host the PDP must first request that the currently running profile on the PEP routers be removed. The Linux traffic control stack does not allow for multiple profiles to be assigned to an individual output port concurrently. So the PDP informs the PEP to run a clean up script which is used to remove any queuing profiles assigned to any of the Ethernet ports on the router node or nodes. Its responsibility is to remove any data traffic profiles that were downloaded from the Policy Decision Point (PDP) "policy manager". In the demonstration when the network batch job has been completed the PDP informs the PEP to run the script clean_script as shown below.

```
#!/bin/sh –x

TC=/usr/src/qos/iproute2/tc/tc

#erase etho diffserv parameters
$TC qdisc del dev eth0 handle 1 root

#erase eth1 diffserv parameters
$TC qdisc del dev eth1 handle 1 root

#erase eth2 diffserv parameters
$Tc qdisc del dev eth2 handle 1 root
```

This clean up script simply deletes all associated scripts that are assigned to each device interface. This will allow the PDP to down load different policy profiles for new or existing applications.

6. System Modeling

6.1 Introduction

One of the unknowns that still exists with a priority queuing scheme for quality of service is its ability to scale to larger network sizes. To determine if our lab approach to QOS can maintain its contractual obligation in prioritizing flows, two-network case scenarios were created. The first takes a look at priority queuing at the node level. That is, it displays two Router nodes implementing priority queues and then shows the resulting bandwidth utilization of pre and post queuing. The second scenario tracks an application flow across an expansive network to compare and contrast this flows attributes with, and without, using priority queuing. The modeling software package that was utilized to provide the simulations is Opnet Modeler Version 24, Release 7.0.B and is copyright protected by OPNET Technologies, Inc.

6.1.1 Baseline Node Simulation

The network configuration for the baseline node simulation is shown in Figure 1. Three case scenarios were created and simulated based on this configuration. These included a control run, a full run with no QoS mechanisms, and a full run using QoS. The scenarios will be discussed in more detail later. As can be seen in the baseline figure, there are four LAN elements Source_Scada, Sink_Scada, Source_Back, and Sink_Back. The LAN elements are interconnected by routers, which are in turn interconnected by a T1 (1.544Mbps) trunk. The LANs use 10 Mbps links.

A video streaming process provides the traffic load for all of the scenarios. The video stream is uni-directional from the source to the sink. The only data flowing from the sink to the source is link control information. The model routes traffic from Source_Scada to Sink_Scada, and it routes traffic from Source_Back to Sink_Back. Both video sources use the same traffic generation pattern. The traffic pattern mimics a continuous video stream at a rate of 1,200,000 bps. The stream model is that of 15,000 byte frames at ten frames per second. I chose this traffic pattern to match the available bandwidth of the T1 trunk. The bandwidth of the T1 can accommodate a single video stream but not that of two streams. This allows for testing of the QoS process.

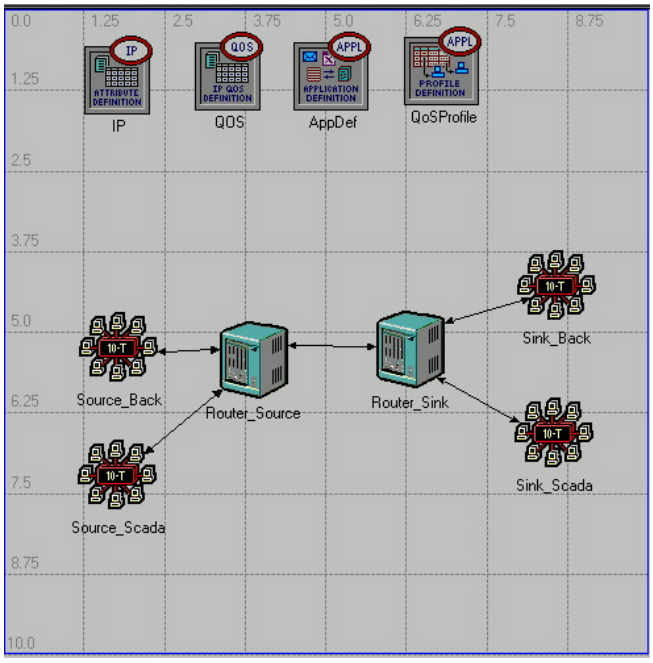


Figure 1 -- Baseline Configuration

The three test scenarios look at the performance of the network with respect to QoS. The first scenario, Control, is a control run. It has a single video source in the Source_Scada LAN and no sources in the Source_Back LAN. This scenario provides an indication of the base operating performance of the network with the video load. The second scenario,

No_QoS, builds on the Control scenario by adding a video source to the Source_Back network. No QoS provisions are made in this scenario. The third scenario, QoS, then builds on the No_QoS scenario by adding priority queuing with priority given to the Source_Scada LAN. The No_QoS and QoS scenarios generate total traffic at a rate of 2,400,000 bps at the sources which exceeds the capacity of the T1 trunk. Figure 2 shows the T1 utilization over time. In the Control scenario, the T1 trunk carries traffic from a single video source and it has additional unused capacity. In the other scenarios, the T1 is at full capacity.

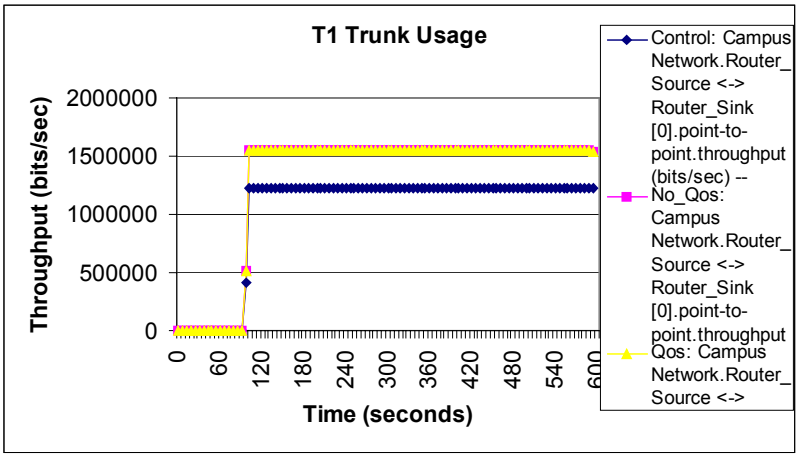


Figure 2 - T1 Trunk Traffic

Figure 3 shows the point-to-point packet queuing delay on the T1 link between the routers for the No_QoS scenario. As you can see in the figure, the queuing delay increases with time since there is not sufficient T1 bandwidth to accommodate two simultaneous video streams.

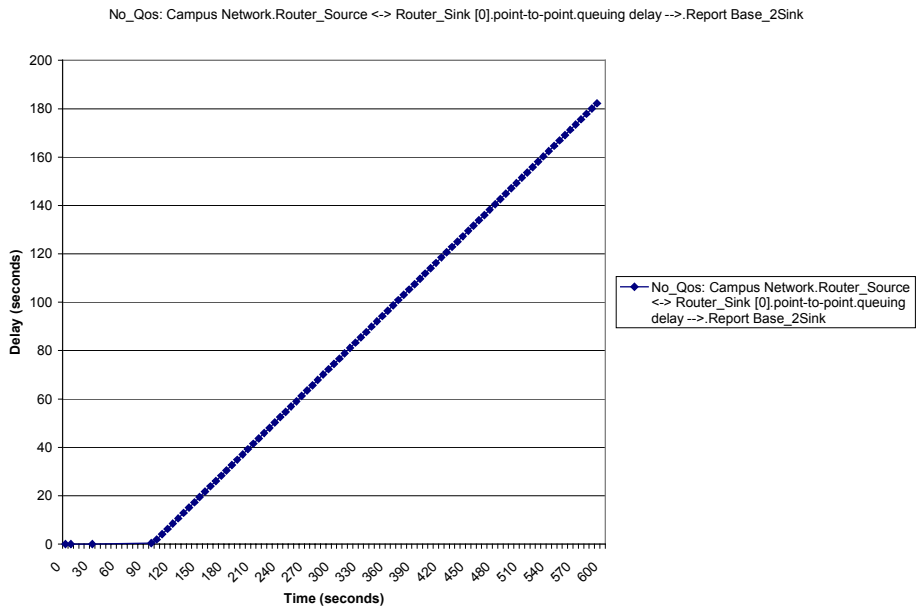


Figure 3 -- Queuing Delay to T1 Link Without QoS

Figure 4 shows the video stream traffic to the destination LANs under scenarios No_QoS and QoS. (Video stream traffic from each source is identical for these scenarios.) The data measures are on the Ethernet links between Router_Sink and the respective sink LAN. Without priority queuing, the available T1 trunk bandwidth is evenly split between the video sources on the Source_Scada and Source_Back LANs. With priority queuing, the video source on the Source_Scada LAN receives the full application bandwidth, and the video source on the Source_Back LAN receives the remainder of the T1 bandwidth

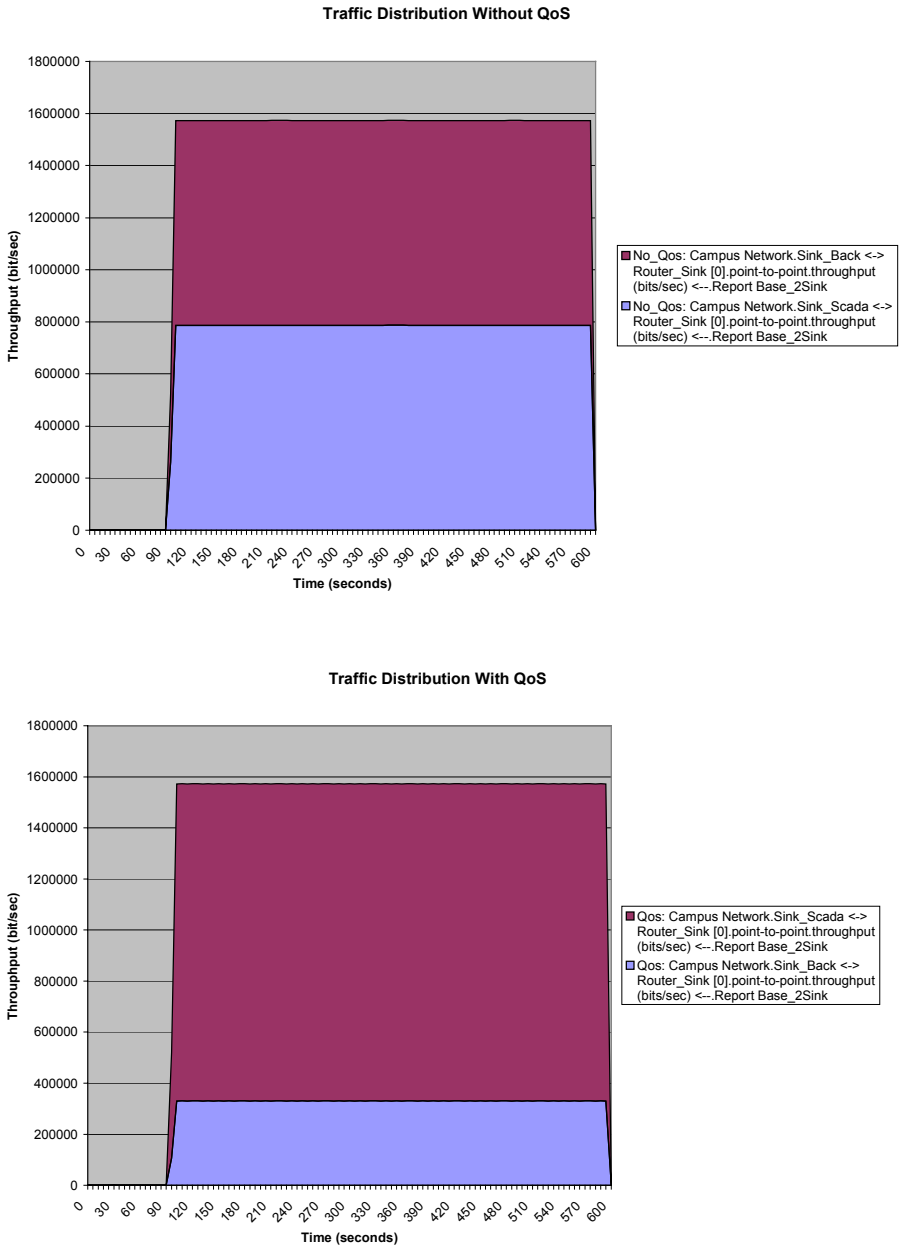
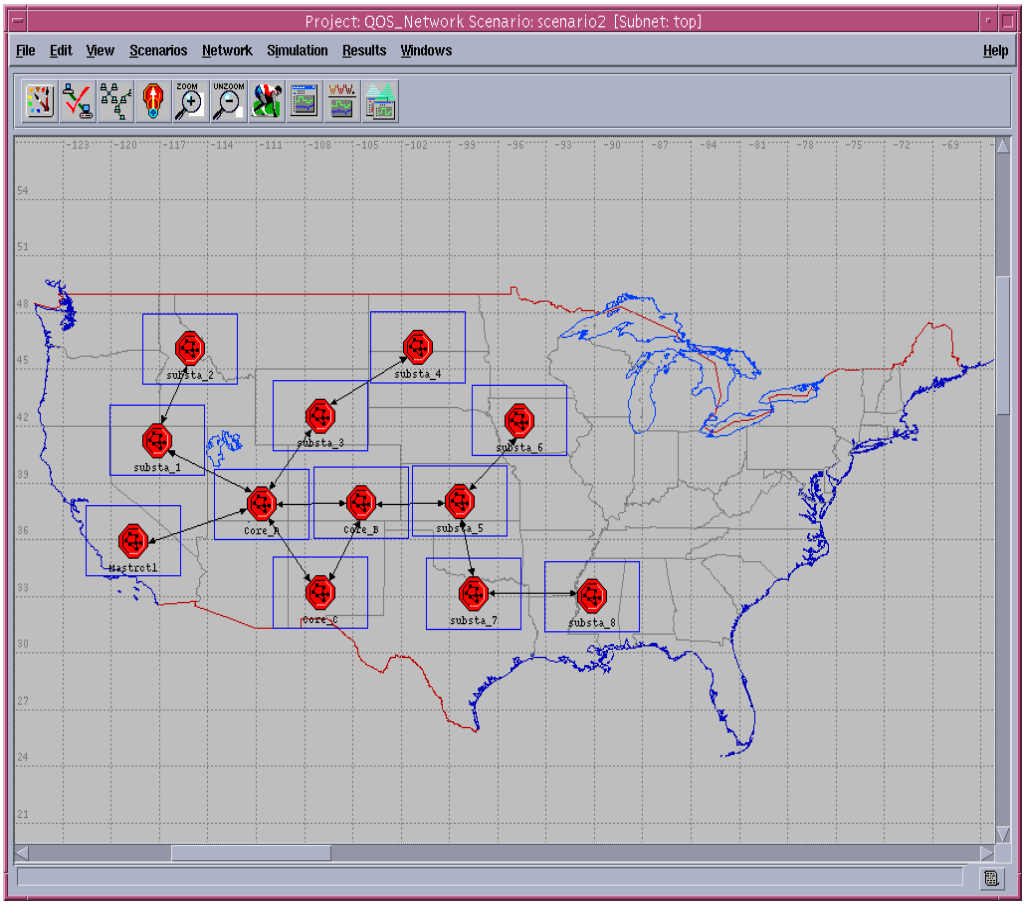


Figure 4 No_QoS and QoS Scenario Comparisons

6.1.2 System Flow Simulation

The network depiction shown below represents a baseline of network connectivity. The cluster of nodes represents a national sized network that will be used to simulate the traffic pattern of a single master station node labeled as “Mastrctl” and many subordinate nodes labeled as “substa_x”. Many flows are created between the dispersed substa nodes and the Mastrctl node. The substa nodes represent client traffic with a mix of heavy web page activity along with some FTP file transfer activities. All client nodes access the servers located in the Mastrctl node. Also for additional link loading, a video conferencing stream has been created between substa_1 and substa_8. The location of these substas, along with the types of services implemented, was designed to create congestion at the access points of the core nodes. All substa nodes are interconnected by T1s, with the following exceptions; the Mastrctl to Core A node is a T3 circuit, the substa_1 to Core A link is a T3 circuit, and the substa_7 to substa_5 link is a T3 circuit. T1 stream represents a data rate of 1.544 Mbps. A T3 stream represents a data rate of 45 Mbps. All outlying T1s are then aggregated at the"core" nodes into T3s. The data flows



that are created from the network shown in figure 1 will be used to create congestion for a pre-defined flow. The pre-defined flow represents a packet voice application. It is a 64 Kbps PCM encoded voice stream with a need for "real time" delivery. The two end stations that are participating in this voice application are located on workstations within the Mastrcntl node and the substa_8 node. These locations were purposely chosen to create a long distance, with multiple hops between these two participating end stations. The distance and the intermediate nodes that this flow must traverse increase the receival and response delay of each transmitted voice packet.

As can be seen in Figure 1, the two major areas of congestion occur at the nodes where many flows converge. This is identified at nodes core A, and substa_5. These two nodes are comprised of Cisco 7200 series routers. The core A router has three active access interfaces that are comprised of one T1 (1.544Mbps) link, along with two T3 (45Mbps) links that contend for 45 Mbps available bandwidth. The substa_5 router has two access interfaces; one is a T3 link the other is a T1. These two links contend for an output bandwidth of 45Mbps.

The first simulation run consists of all nodes actively participating in data retrieval and transmission. The "real-time" application is launched and two parameters, *packet end to end delay* and *packet delay variation,* are monitored for performance. These two parameters were chosen because in a "real-time" data flow these measurements can predict the quality of the data flow.

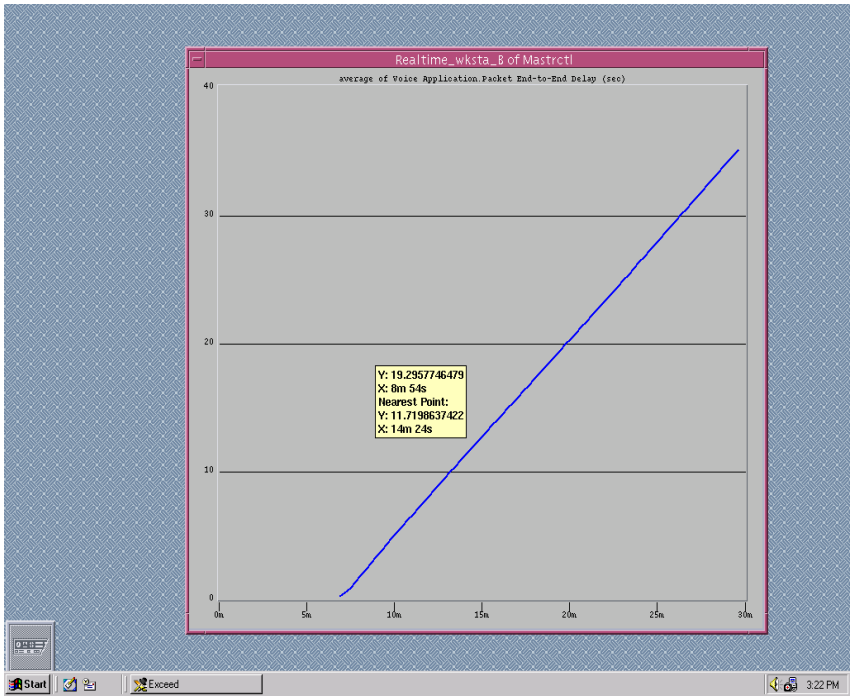


Figure 2

The minimized amount of time between the launching and the subsequent receival of each packet prevents lapses in the voice speech. The *packet delay variation* refers to the difference in time between each arriving packet. This also needs to be kept to a

minimum to allow build out buffers in the receiving node from being starved of data to process. When this happens, synchronization of the voice circuit can be lost. Figure 2 shows the results of the *packet end to end delay* **without** QoS prioritization for this voice flow.

As seen on the graph in Figure 2, the time line for the *packet end to end delay* continuously increases for the entire duration of the simulation. This is a clear indication that the over all end to end delay never reaches a point of normalization and within the queuing buffers of the over-subscribed link bandwidth of the core routers, packets are being delayed to the point of discard. A similar result occurs when the *packet delay variation* graph is plotted as shown in Figure 3 below.

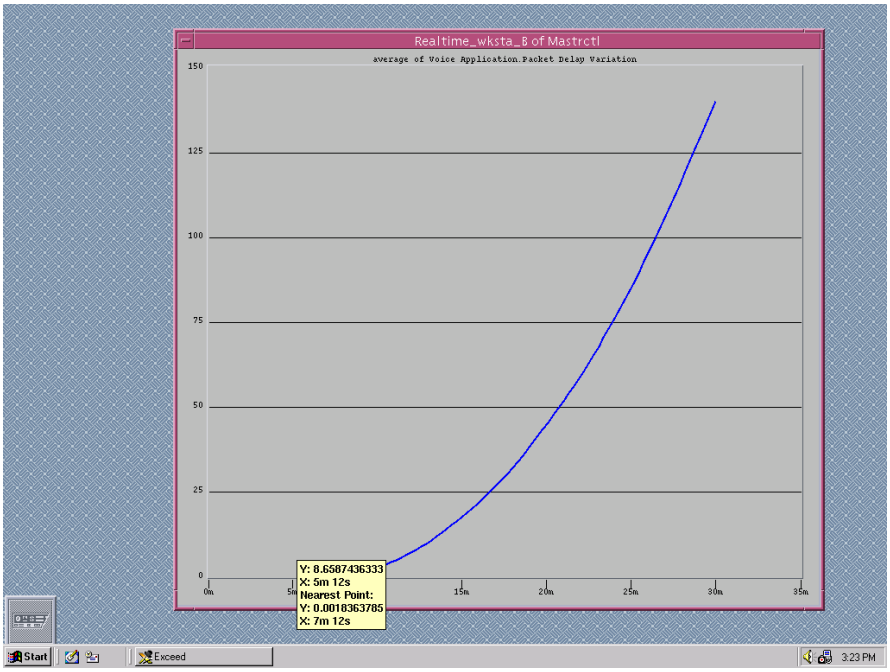


Figure 3

The monitored voice application flow can not compete with the other applications that are all contending for the over-subscribed link capacity. The continual increase in the amount of time is an indication that the simulation cannot normalize the overall loss of packets.

To provide QoS to the voice application flow, a priority queuing scheme had to be constructed. This was accomplished by assigning each application flow within the model a relative ranking or "priority". This priority would then be used within the congested routers to determine which flows would be given preferential treatment. A profile was created on each application that was active during the simulation. The Web browsing and the FTP applications were given the lowest priority. This meant that when congestion occurred, the packets belonging to these data flows would be buffered the longest amount of time and eventually discarded. The competing video conferencing application, which was implemented on the workstations located in node substa_1 and

substa_8 was given a medium priority. This would allow these packets to be forwarded more often than the data application, but not as frequently as the highest priority, which was given to the monitored voice application. The QoS priority schemes were implemented on the Core A and the substa_5 routers. Looking at the model in Figure 1, it can be seen that the point of contention for all the active flows is present at both of these nodes. Once the priority schemes are implemented at these two critical points the network flows are now regulated, based on the priority queuing profile, to and from the Mastctl node. The simulation is then run again monitoring the previously mentioned voice application attributes with vastly different results as seen in Figures 4 and 5.

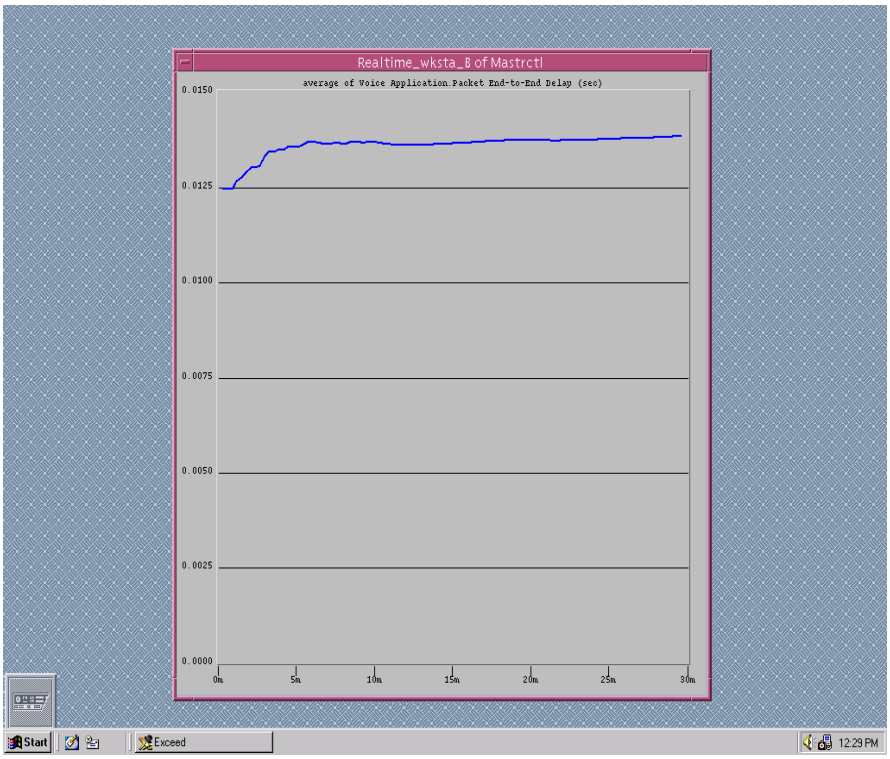


Figure 4

Notice in Figure 4 that the average packet end to end delay normalizes between the values of .0125 seconds (12.5 milliseconds) and .0150 seconds (15.0 milliseconds) and very reasonable delay for a large expanse, multi-hop network. Notice the Figure 4 graph compared with the Figure 2 graph. Because of the lack of a priority queuing scheme, the average packet end to end delay was unable to normalize even past 30 seconds.

The graph in Figure 5 shows the average packet delay variation normalizing at an average of 20 microseconds per packet; a very minimal delay between packet arrivals. Compare this graph with the one in Figure 3. Notice that without any priority queuing scheme the packet delay variation is unable to normalize.

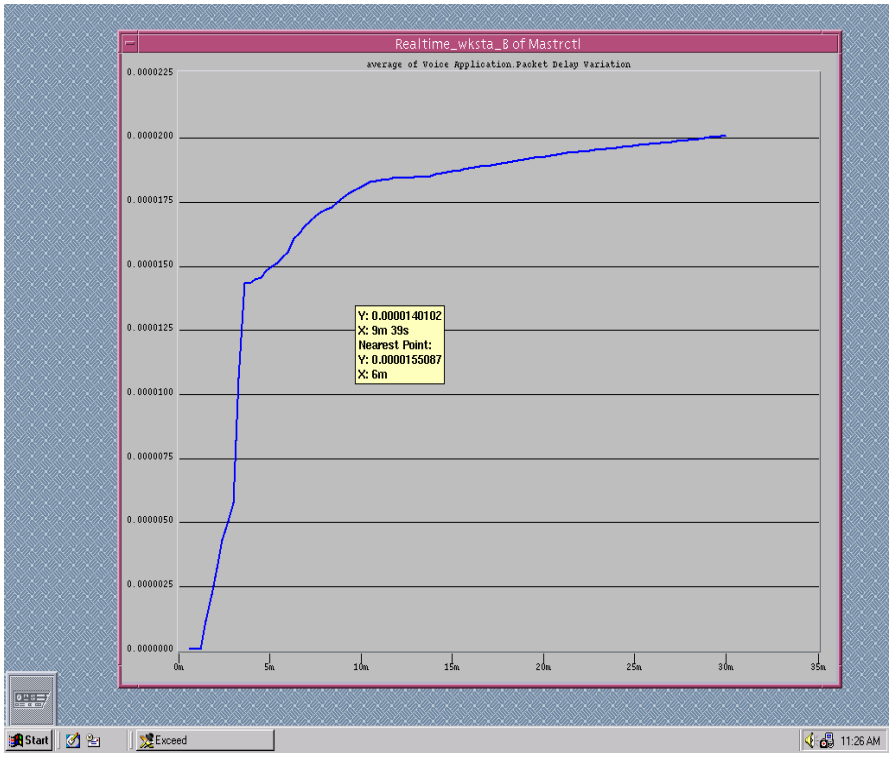


Figure 5

As was shown in the previously discussed simulation results, a priority queuing scheme can be constructed that would allow for "real-time" response across a geographically dispersed network. This previous simulation, for purposes of simplicity, only monitored the results of one priority flow. It would be reasonable to state that the overall *packet end to end delay* and the *packet delay variation* would increase with respect to time based on the number of priority flows defined.

7. System Results

The results of the lab tests that were conducted as shown in section *5, System Test Methods* were favorable. Using the video conferencing tool as a priority steam the QoS mechanism that was deployed allowed as much bandwidth as needed for transport across the lab network. Another result of a test using a large FTP file transfer while the traffic generator was transmitting a large volume of traffic was also very favorable. Without QoS queuing enabled the file transfer required several minutes to complete because of congestion created by the traffic generator. With the queuing enabled the same large file transfer only took a mater of seconds to complete its transfer. This showed that the queuing principles could actually be implemented effectively.

The *System Modeling* section showed that these queues could be implemented over a large-scale network without an over-abundance of processing delay per each hop node. It showed that a chosen "real-time" application flow could be successfully sustained across an expansive network with very favorable inter-packet and end to end delay results.

8. Conclusions

It was quite apparent during this research effort that there is a definite need to deploy procedures and protocols that enable the ability of network managers to control the distribution of bandwidth. Bandwidth is the commodity of the communications world. It needs its distribution controlled to allow all services on a network to co-exist. Without control of this commodity the ability to provide needed guarantees for priority services becomes impossible.

The techniques described in this research combined the three necessary components of flow identification and marking, queuing and flow control, and policy management to create a complete QoS system. Although the methods demonstrated were somewhat rudimentary they demonstrated the needed configurations to allow network control of bandwidth resources.

As the attention to network "quality of service" intensifies the choices and refinements of approaches will eventually lead to a over-all standard approach to a protocol implementation. This will surely include the components that were demonstrated in this research effort.

9. References

1) [Braden] Braden, R., et.al., “Resource Reservation Protocol (RSVP) – Version 1 Functional Specification”, 1996.

2) [McCanne1] McCanne, S., Jacobson, V., “*VIC*: A Flexible Framework for Packet Video”. In *Proceedings of ACM Multimedia 1995*, ACM.

3) [Almesberger1] Almesberger, W., Salim, J., Kuznetsov, A., "Differentiated Services on Linux". In ???, 1999.

4) [McCanne2] McCanne, S., Jacobson, V., Vetterli, M., “Receiver-Driven Layered Multicast”. In *Proceedings of SIGCOMM 1996*, ACM.

5) Ekstein, Ronnie, T`Joens, Yves, and Sales, Bernard "AAA Protocols: A Comparison between RADIUS, DIAMETER and COPS", *IETF Internet Draft, January 2000.*

6) Boyle, Jim, Cohen, Ron, et al. "The COPS (Common Open Policy Service) Protocol" *IETF RFC 2748, January 2000.*

7) [Almesberger2] Almesberger, W., “Linux Network Traffic Control – Implementation Overview”. In ???, 1999.

.

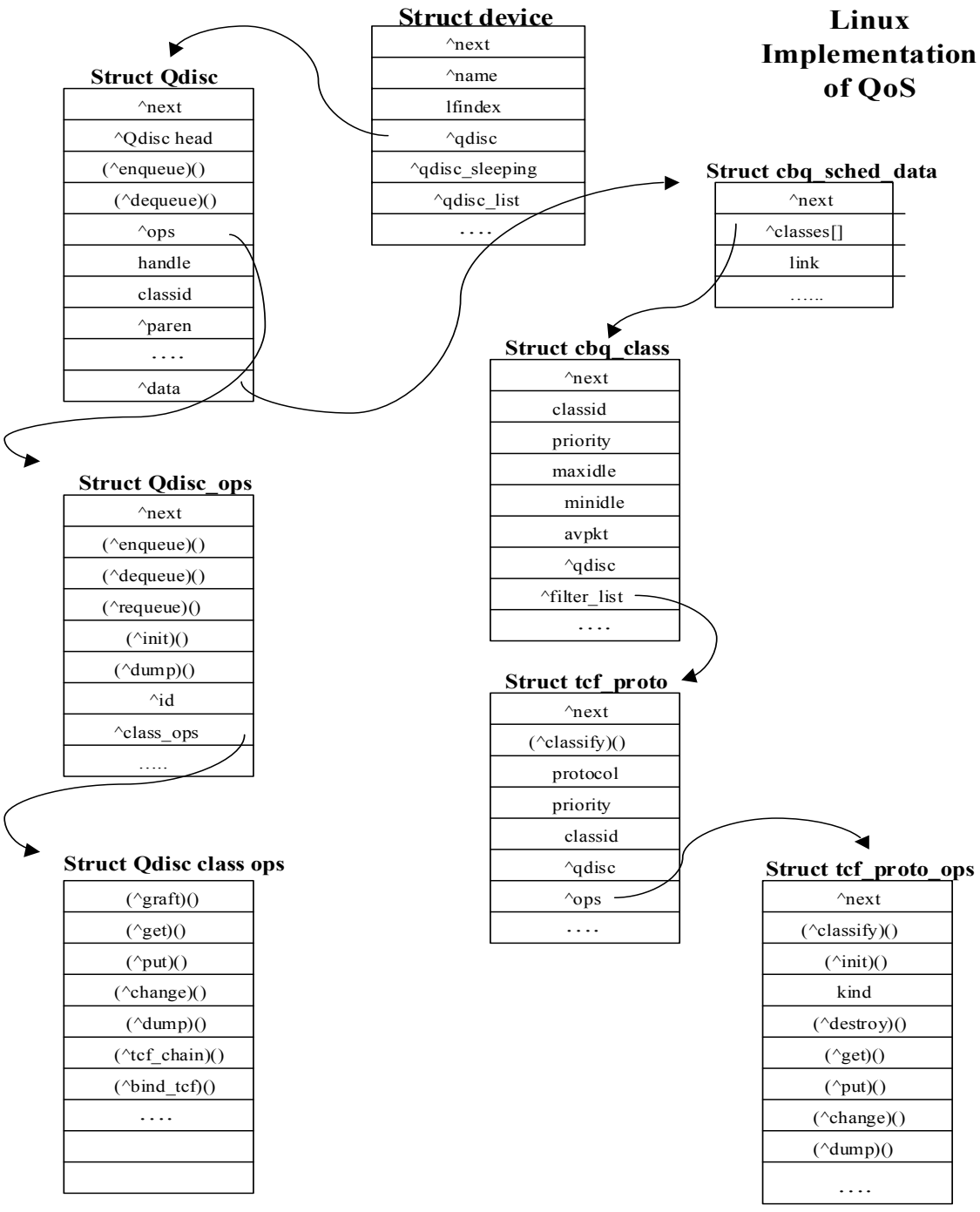
8) Reichmeyer, Francis, Herzog Shai, Chan, Kwok Ho, et al. "COPS Usage for Policy Provisioning" *IETF Internet Draft, August 2000.*

9) Schwartz, Randal L., "Learning Perl", *O'Reilly & Associates, Inc., 1993, ISBN 1-56592-042-2.*

10) Herzog, S., Boyle, J., Cohen, R., et al. "COPS usage for RSVP" *IETF RFC 2749, January 2000.*

10. Appendices

Appendix A: Linux Implementation of QoS



Appendix B: Perl code for Policy Decision Point Socket

```
#!/usr/bin/perl

############################################################
# Used for PDP socket interface construction
# Slightly hacked and heavily documented simple client example from the book
# Programming Perl, 1st Edition.
# Some things are done a little differently than in the server examples to give you a little
# more experience with Perl constructs.
#
# This client accepts input from standard input, sends the input to a server, and sends
# responses from the server to standard output.
#
# Hacking and documentation by Proffesor Golden D. Richard III, Department of
# Computer Science, University of New Orleans, April 1196 –March 1998.
#
#Command line arguments expected are:      serverIPaddress, port
#
============================================================
############################################################
# Perl socket stuff is based on, and very similar to , C socket stuff.  The Unix man pages
# for the various socket system calls such as `listen`, `bind`, etc. may be very useful to
# you.  To get information on a particular call, such as `accept`, use the following
# command:
#
#
#
#         % man –s3N accept
#
# The –s3N tells man to look in 3N section of the Unix manual, a section that`s not
# searched by default.
#
# Use port addresses in the range 5000-6000 for the assignments.  If you get an “address
# already in use” error, try a different port.
#

#########################################################################
# In Perl 5 and above, the socket module contains Perl definitions for the stuff in the C
# include file "“usr/include/sys/socket.h”.  Rather than manually poking through that
# include file to get values for each architecture, a “use socket” assures that you`ll get the
# right values.  This replaces the following lines in the example in the book:
#
# $AF_INET = 2;
# $SOCK_STREAM = 1;
#
```

```
# which turn out to be wrong for Solaris, anyway…
# ($SOCK_STREAM should be 2)

use Socket;

# I don`t like it in the command line arguments (resident in the array varible ARGV)
# and assigns them to the variables listed in ().  In this case, the first is the IP address of
#the machine on which the server is listening.

($them, $port) = @ARGV;

# This checks to see if `port` has been assigned a value and if it hasn`t assigns the default
# port value 2345.  If `them` (the machine on which the server is running) hasn`t been
# assigned a value, `local host` is assumed.  This means that the server is on the same
# machine as the client (us).

$port = 2345 unless $port;
$them = `localhost` unless $them;

$fname = "/tmp/" . $them . ".cops";
$lname = "/tmp/" . $them . " .lock";
$oname = "/tmp/" . $them . " .out";
while (true)     {
        if ( ! –r $fname ) {
                print " No data file for $them…  Retrying  !\n";
                sleep 5;
        } else {
           last;
        }
}
while (true) {
        if ( -r $oname ) {
                print "Output file exists for $them….Waiting for its removal !\n";
                sleep 5;
        } else {
                last;
        }
}
# This sets up a signal handler to kill the child we`ll create later in the event that we die.
#
#

$SIG{ `Int` } = `dokill`;

# This is a subprogram that does the killing.  The variable $child is initially undefined, so
# we won't try to kill a non-existent child.  Later it will be the PID of the child we
```

```
# created.
Sub dokill {
        Kill 9, $child if $child;
}

# Strings in back quotes like `ls` run the Unix command inside the quotes and return the
# output. The hostname command returns the name of the machine we`re running on.
# `chop` removes the newline character in the output from the hostname command.

Chop($hostname = `hostname`);

# looks up important information related to the network protocol you wish to use. `tcp` is
# a connection-oriented protocol. Look in /etc/protocols for examples of others.  Do not
# change this unless you know what you're doing.

($name, $aliases, $proto) = getprotobyname (`tcp`);

# This is a slightly different way of doing the service look up that was done in the server
# code.  Basically, if the specific port is an integer, the port number look up isn`t done,
# otherwise it is.

( $name, $aliases, $port) = getservbyname ( $port, `tcp` )
        unless $port =~ /^\d+$/;

# let the user know what port we're using and where the server is expected to be, just in
# case (s)he accidentally typed an incorrect port or machine name.

Print "Using port $port to connect to server on host $them…\n";

# This looks up numeric IP address information corresponding to the hostname for the
# current machine.

($name, $aliases, $type, $len, $thisaddr) = gethostbyname($hostname) ;

# This looks up numeric IP address information corresponding to the hostname where the
# server is expected to be running.

($name, $aliases, $type, $len, $thataddr) = gethostbyname($them) ;

# Create on end point of the communication link (our end).  See the server code for a
# more complete explanation of what `socket` does.  If we fail to create a socket, `die`
# causes execution to terminate and display the reason (stored in $! ) for the failure.

# Remember that AF_INET and SOCK_STREAM are symbols provided by the "use
# socket;" line.  Don't put $ in front of them!
```

```
If (socket(s,AF_INET, SOCK_STREAM, $proto))   {
        Print “Socket created succeeded. \n”;
}
else {
        die $!;
}

# This is similar to the `listen` in the server, except the server is waiting for a call and
# we’re actively initiating the conversation.  `connect` connects our socket to the server`s
# socket on the other end.

If (connect(S, $that))   {
        Print “Connect succeeded. \n”;
}
else {
        die $!;
}

# Force our socket to flush output immediately after a print

select (S);
$| = 1;

# Make standard output the default again

select(STDOUT);

# The interactions we want are to type lines of input and have them echoed back by the
# server.  But how can we both wait for input and be receptive to the server’s output?
# Answer:       By forking a process to accept input from standard input (the keyboard)
# and send it to the server, and using our current process to receive and display input
# from the server.

If ($child = fork)        {

# We’re the parent.  Read lines of input from the standard input and send them to the
# server until end of file is seen.

# $fname = $them . “ .cops”;

open(STDIN, $fname)  | | die “Can`t open $fname for reading”;
        while (<STDIN>)  {
                print S ;
}
# Sleep for 3 seconds then …
```

```
sleep 3;

# … then kill ourselves and the child

do dokill ();
}
else {

# We`re the child.  Read lines of input from the server over the socket S and output them.
# Stop if end of file is seen.

$ofile = " /tmp/"  .  $them .  " .out";
while (<S>)  {
        open(OF, ">>$ofile")  |  | die "can't open $ofile for writing";
        print "$them: $_";
                close OF;
        }
}
```

Appendix C: Perl code for Policy Enforcement Point Socket

```
#!/usr/bin/perl

##########################################################
# Used for PDP socket interface construction
# Slightly hacked and heavily documented simple client example from the book
# Programming Perl, 1st Edition.
# Some things are done a little differently than in the server examples to give you a little
# more experience with Perl constructs.
#
# This client accepts input from standard input, sends the input to a server, and sends
# responses from the server to standard output.
#
# Hacking and documentation by Proffesor Golden D. Richard III, Department of
# Computer Science, University of New Orleans, April 1196 –March 1998.
#
#Command line arguments expected are:       serverIPaddress, port
#
=========================================================
##########################################################
# Perl socket stuff is based on, and very similar to , C socket stuff.  The Unix man pages
# for the various socket system calls such as `listen`, `bind`, etc. may be very useful to
# you.  To get information on a particular call, such as `accept`, use the following
# command:
#
#
#
#         % man –s3N accept
#
# The –s3N tells man to look in 3N section of the Unix manual, a section that`s not
# searched by default.
#
# Use port addresses in the range 5000-6000 for the assignments.  If you get an “address
# already in use” error, try a different port.
#

##############################################################################
# In Perl 5 and above, the socket module contains Perl definitions for the stuff in the C
# include file ""usr/include/sys/socket.h”.  Rather than manually poking through that
# include file to get values for each architecture, a “use socket” assures that you’ll get the
# right values.  This replaces the following lines in the example in the book:
#
# $AF_INET = 2;
```

```
# $SOCK_STREAM = 1;
#
# which turn out to be wrong for Solaris, anyway…
# ($SOCK_STREAM should be 2)

use Socket;

# I don't like it in the command line arguments (resident in the array variable ARGV)
# and assigns them to the variables listed in ().  In this case, the first is the IP address of
#the machine on which the server is listening.

($port) = @ARGV;

# This checks to see if `port` has been assigned a value and if it hasn't assigns the default
# port value 2345.

$port = 2345 unless $port;

# Looks up important information related to the network protocol that you wish to use.
# `tcp` is a connection oriented protocol.  Look in /etc/protocols for examples of others.
# Don't  change this unless you know what your doing.

( $name, $aliases, $protocol) = getprotobyname(`tcp`);

# The following line illustrates both how terse Perl can be and how cool Perl can be.
# $port is the port the user specified either on the command line or by default.  =~ is the
# Perl regular expression binding operator and !~ is the negation of =~.  regular
# expressions are enclosed between / / characters in Perl.  \d matches a single digit, \d+
# matches a string of 1 or more digits, ^ specifies "beginning of string", and $ specifies
# "end of string".  /^\d+$/ as a regular expression, therefore, matches integers of arbitrary
# length.  So the following line says:  "If the port contains any non-digit characters, look
# up the port number associated with the symbolic name specified by the user.  The
# look up is done against /etc/services.

If ($port !~ /^\D+$/)  {
        ($name,  $aliases,  $port) = getservbyport($port,  `tcp`);
}

# Let the user know what port we're listening on, just in case (s)he accidentally typed an
# incorrect port.

Print "Listening on port $port…\n";

# `socket` creates one end point for a communication link (think of it as creating a
# telephone.  Later, some one else will create another telephone and wire will be attached
# between them).  The S parameter is the handle associated with the created
```

```
# communication endpoint.  AF_INET specifies that we're talking using ports.
# AF_UNIX would specify that we'd be communicating through special files created in
# the file system.  This is very attractive because then you can do away with the port
# number business, but unfortunately AF_UNIX sockets only work on the same machine.
# SOCK_STREAM sockets communicate using streams of characters.  Another
# possibility is unreliable datagram communication using SOCK_DGRAM.  Just stick
# with the parameters used here unless you know what you're doing.  Because there are
# other implications you need to understand for datagram communication.

# In case you're wondering, the AF_INET and SOCK_STREAM symbols are provided
# by the `use socket;` statement at the top of the file.  You do not want to put $'s in front
# of these symbols.

# The `die` causes execution to terminate with an error message ( which is stored in $_) if
the `socket` call fails.

Socket(S,AF_INET,SOCK_STREAM,$protocol)  | | die "socket : $!";

# the `bind` hooks your phone to the port number that was specified.  Think of all the
# ports as a telephone switchboard.  `bind` requires its parameters to be in a C structure
# format.  `pack` smooches things together  into a form that `bind` can stomach.  The
# $sockaddr thin below says "An unsigned short, followed by a short in `network order`
# followed by a null-padded four character string, followed by eight null bytes."  It's
# magic.  Don't worry too much about it.

$sockaddr = `S n a4 x8`;
$this = pack($sockaddr, AF_INET, $port,  "\0\0\0\0");
bind(S, $this) | | die "bind : $! ";

# The following arranges to queue up as many as 10 PDP's until we have a chance to
# service them.  If more than 10 PDP's "get in line", the excess may receive "connection
# refused" errors.

Listen(S,10) | | die "listen: $!";

# Select S temporarily as the default output channel, turn on autoflushing, then select
# standard output again.

Select (S) ;
$| = 1;
select(STDOUT) ;
```

```
# Create connections as PDP's "arrive".  $con maintains connection number of the last
# PDP.

For      ($con = 1;  ; $con++)   {

# let the user know we're waiting for a connection…

printf ("waiting for connection %d…..\n" , $con);

# `accept` blocks until it notices that a connection has been made to out socket S.  When
# this occurs, the incoming connection is actually attached to the socket NS rather than S,
# thus leaving S free for other incoming connections.

# One more time said another way, when we hear the telephone S ringing, we quickly
# transfer the call to a different phone (NS) to accept the call.  This allows S to stay free
# for the next call.  Make sense?

# The value that's returned ( in $addr) gives some information about the address of the
# caller.

($addr = accept (NS, S)) | | die $!;

# Temporarily set default output to the handle NS so….

Select (NS);

# …so we can set autoflushing.  Setting $| to a non-zero value causes output to the
 # currently selected output channel to be immediately flushed.


$| = 1;

# set default output back to the standard output channel

select(STDOUT);

# The `fork` creates a child process to handle this PDP.  Remember that `fork` returns 0
# for the child and a positive number for the parent.

If       (($child = fork()) = = 0)        {

# we're the child… unpack the information returned by `accept` to get some (readable)
# information about the PDP we're serving and print it to standard output

($af, $port, $inetaddr) = unpack($sockaddr, $addr);
@inetaddr = unpack(`C4` , $inetaddr);
```

```
print “Serving connection $con @ Internet address @inetaddr \n”;

# NS is the handle for the socket we’re listening to it’s connected to the current PDP.
# <NS> reads and returns the next line of input from the handle NS.  The following while
# loop form is a special Perl construct that reads input from the handle NS line by line
# until the end of the file is reached.  Each line is placed into the special Perl variable $_
# (the Perl “default value” variable).

While (<NS>) {

# output stuff from PDP here and ….

Print “received from PDP $con:  $_”;
        System( “ $_ 2>&1> perl.out”);
        $command = `$_`;

# …and echo it back to the PDP, too.

Print NS “PEP $con: $_”;
Print NS “$command”;
Print “$command”;
}

# close the socket connection when the PDP goes away.

Close(NS);

# The fork PEP dies here

print “PDP went away.  Forked PEP $con exiting…\n”;
exit;
}

# This is where the parent returns.  All we do is close the socket connection (it’s being
# handled by the child we forked) and then enter another iteration of the big for loop.

Close(NS);
}
```