The Design and Implementation of the ALTQ Traffic Management System

Kenjiro Cho

January 2001

Graduate School of Media and Governance
Keio University

This document was submitted to Keio University
in January 2001, as a doctoral dissertation.

Preface

This work is in part evaluated in terms of contribution to other research and experiments in the form of an open software implementation. I am grateful to my committee members at Keio University who recognize the validity of research based on a software implementation which has not traditionally been perceived as a research theme.

A software implementation can be compared to a justification process in science or a proof in mathematics. For scientific results to be justified or for mathematical theorems to be proved, they must be reproducible by others. By the same token, a software design is justified and proved through an implementation that allows others to reproduce its results. The power of the computer, along with information sharing over the Internet, resides in its ability to enable peers to easily reproduce and verify results, and to accelerate the cycle of innovation.

Reproducible results are also essential to the robustness of the results. Programs are used and tested in a wider variety of contexts than one could generate, and bugs get uncovered that otherwise would not be found. When the source code is available, bugs can often be removed, not just discovered, by someone outside the development process.

Sharing software components leads to new ideas. A good program often inspires others to tackle new ideas that would not have been conceived without it. Code sharing minimizes duplication of effort. With the source code available, others can easily combine or modify the existing components to try out new ideas.

This work is on a research platform for quality-of-service (QoS) in networking. QoS is one of the most elusive, confounding, and confusing topics in data networking today, mainly because QoS means so many things to so many people. Unfortunately, there are too many conflicting concepts that often involve complex, impractical, incompatible, and non-complementary mechanisms for delivering the desired results. QoS is clearly an area in which research is needed, tools must be developed, and industry needs much improvement, less rhetoric, and more consensus.

In the early stage of my QoS research, I realized a simple fact that QoS is hard to reproduce. There were research implementations customized for specific purposes but they were not generalized for use with other components. These implementations run on a specific OS version and support a limited number of network drivers. The behavior of these systems are often sensitive to test environments.

Clearly, the community lacked a common platform to share results. A common platform was needed to verify the results of a mechanism proposed by others, and to test them in a much wider range of environments. A common platform was needed to enable peers to explore new ideas by combining the existing components and adding new functions. A common platform was also needed to learn and understand non-trivial behavior of QoS systems through experiences.

Hence, I started implementing ALTQ on commodity PC hardware and free versions of UNIX. The engineering challenge was to conceal differences in network cards, device drivers, processors, and operating system versions in order to enable peers to reproduce the same results, or at least similar results in their own environments.

Various experiments have been done with ALTQ in different environments ranging from small test networks in laboratories to large testbed networks, and even in course projects at universities. In the meantime, the quality of the software has been considerably improved through bug reports and feedback from people who have used ALTQ. A number of research projects use ALTQ to explore new ideas in related areas such as signaling, policy management and label-switching.

I believe that, for technological advancement, software implementations to reproduce research results are no less important than new discoveries. We should learn from previous ideas, not only through papers but also through working computers and the Internet. I hope this work will encourage people working on software implementation projects.

Acknowledgments

I would first like to thank my management at Sony Computer Science Laboratories Inc., in particular Dr. Mario Tokoro, for allowing me to work on the ALTQ project and for their guidance throughout the years.

I would like to thank my advisor, Professor Jun Murai, for encouraging me to challenge this dissertation and for his invaluable advice. I would also like to thank the rest of my committee members, Professor Hideyuki Tokuda, Professor Tatsuya Hagino, Professor Hiroaki Chiyokura, and Professor Yoichi Shinoda, for taking time to learn about my work and providing me with valuable feedback on how it could be improved.

I am indebted to many people who have contributed to the ALTQ project by reporting bugs, by providing suggestions, and by other forms of feedback. I would like to express my appreciation to individuals who have helped, either directly or indirectly, to stimulate my thought processes over the course of the past few years while I was working on the project.

I would like to thank my colleagues at Sony Computer Science Labs, among them, Hiroshi Esaki, Fumio Teraoka, Atsushi Shionozaki, Yoshifumi Nishida, and Sohgo Takeuchi, for supporting the project. Thanks to the members of the KAME Project and the WIDE Project for their support and for using ALTQ at a number of occasions. Especially, Junichiro Hagino helped me port ALTQ onto multiple BSD based systems and maintain portability among those systems. Yoshito Tobe of Keio University has guided me through the degree process.

I would also like to thank Professor Hui Zhang of Carnegie Mellon University. He invited me to Pittsburgh in the summer of 1999, and provided me a chance to work with his group, among them, Ion Stoica and T. S. Eugene. We worked together to reimplement their H-FSC scheduler onto the ALTQ framework.

Finally, I heartily thank my family for their support and patience during this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traffic management to provide quality-of-service is of great importance to today's packet networks. Traffic management consists of a diverse set of mechanisms and policies, but the heart of the technology is a packet scheduling mechanism, also known as queueing. Sophisticated queueing can provide performance bounds of bandwidth, delay, jitter, and loss, and thus, can meet the requirements of real-time services. Queueing is also vital to best-effort services to avoid congestion and to provide fairness and protection, which leads to more stable and predictable network behavior. There has been a considerable amount of research related to traffic management and queueing over the last several years.

Many QoS mechanisms including various queueing disciplines have been proposed and studied to date by the research community, mostly by analysis and simulation. Such mechanisms are not, however, widely used because there is no easy way to implement them into the existing network equipment. In BSD UNIX, the only queueing discipline implemented is a simple Drop-Tail FIFO queue. There is no general method to implement an alternative queueing discipline, which is the main obstacle to incorporating alternative queueing disciplines.

On the other hand, the rapidly increasing power of PCs, emerging high-speed network cards, and their dropping costs make it an attractive choice to implement intelligent QoS mechanisms on PC-based routers. Another driving force behind PC-based routers is flexibility in software development as the requirements for a router are growing.

In view of this situation, we have designed and built ALTQ, a traffic management system for a PC-based routers. ALTQ allows implementors to implement various QoS mechanisms on PC-based UNIX systems. A set of queueing disciplines as well as management tools are implemented to demonstrate the traffic management abilities of PC-UNIX based routers.

ALTQ is designed to support a variety of QoS mechanisms with different components: scheduling strategies, packet drop strategies, buffer allocation strategies, multiple priority levels, and non-work conserving queues. Different QoS mechanisms can share many parts: flow classification, packet handling, and device driver support. Therefore, researchers will be able to implement a new QoS mechanism without knowing the details of the kernel implementations.

The ALTQ system is designed to support both research and operation. Once such a framework is widely deployed, research output by simulation can be easily implemented and tested with real machines and networks. Then, if proved useful, the mechanism can be brought into routers in practical service. Availability of a set of QoS mechanisms will raise public awareness of traffic management issues, which in turn raises research incentives to attack hard problems.

Another important issue to consider is deployment of the framework. To this end, just a framework is not enough. In order to make people want to use it, we have to have concrete QoS mechanisms which have specific applications. Therefore, in the early stage of the ALTQ development, we have implemented Class-Based Queueing (CBQ) [FJ95] and Random Early Detection (RED) [FJ93, BCC$^+$98] targeting two potential user groups.

One group is researchers and developers of QoS based systems that assume a traffic control support in the underlying platform. In particular, the RSVP release from ISI [ISI] does not have a traffic control module so that there are great demands for a queueing implementation capable of traffic control.

The other group is researchers and operators who want to investigate effects of traffic management. Active queue management has been extensively discussed to avoid congestion in the Internet but, again, no implementation was available at hand.

Another important factor for widespread acceptance is simplicity and stability of the implementation, which caused us to emphasize practicality instead of elegance while designing ALTQ.

Our primary objective in creating ALTQ is to provide a flexible platform that can support various types of QoS functions. The goals of the ALTQ traffic management system are:

- To find proper abstractions of kernel components to support a wide variety of QoS functions. It also addresses the problems in the current operating system and hardware designs to support QoS functions.

- To provide a flexible and well-engineered platform for QoS related research:
  - ALTQ allows researchers to easily implement new queueing disciplines without knowing the details of kernel programming.
  - ALTQ provides missing components to developers of QoS based systems that assume a traffic control support in the underlying platform.
- To provide a set of tools to gain operational experiences. It is essential to the Internet research to obtain feedback from field experiences.

ALTQ has evolved into a QoS platform implemented in several versions of BSD UNIX, and the number of supported queueing disciplines has added up to more than ten. ALTQ has been used for various research and experiments around the world.

This thesis is organized as follows. We first review QoS related technologies in Chapter 2, and then, discuss the architecture of QoS systems in Chapter 3. Our QoS system architecture consists of a framework, forwarding mechanisms and management mechanisms. The framework is presented in Chapter 4, the forwarding mechanisms are presented in Chapter 5, and the management mechanisms are presented in Chapter 6. Our test results are presented in Chapter 7. We discuss applications of the ALTQ system in Chapter 8, related work in Chapter 9, and conclude in Chapter 10.

Chapter 2

Background

2.1 Output Queueing

Routers in an internet have two major roles: routing and forwarding. Routing is a process to maintain a routing table by exchanging routing information with neighbor routers. A router consults the routing table to look up the next hop for the destination of an arriving packet. Forwarding is a process to actually deliver an arriving packet to the next hop.

On a forwarding path, there are three components: an input component resides in the input interface, and receives a frame and checks the integrity of an arriving packet. A forwarding component connects multiple interfaces, and determines the destination interface to which a packet should be passed. An output component resides in the output interface, and frames and transmits packets to the next hop. Figure 2.1 illustrates the forwarding path components of a router.

Each forwarding component operates on one packet at a time and needs to work asynchronously because packets arriving at different interfaces can be destined for the same output interface at one time. Thus, a packet queue is placed at both input and output interfaces in order to hand over packets to the next component.

Input queues and output queues on a router normally have asymmetric loads. When the speed of a link is slower than the forwarding speed of a router, packets are stored in the output queue most of the time. On the other hand, when the speed of the link is faster, packets need to wait in the input queue.

The former case, the link is a bottleneck, is predominant in the Internet and we can use various software techniques to manage packets in the output

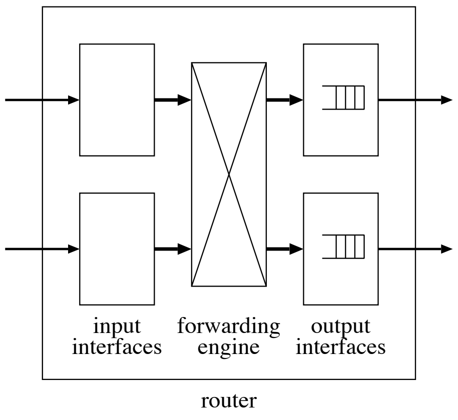


Figure 2.1: forwarding path components of a router

queue. Thus, traffic management and QoS related techniques are usually applied only to output queueing.

The latter case occurs when the router is a bottleneck. For example, a lookup operation for the next hop can be expensive when a routing table becomes large. It also happens with a router with high-speed ports or with a large number of ports; arriving packets can exceed the internal switching capacity of a router. Whereas input queueing is important to build high-speed routers, special hardware is required to solve performance problems, and the situation is quite different from output queueing. Thus, we do not discuss input queueing in this paper, and our focus is on output queueing in order to manage traffic.

2.2 Queueing Theory

Queueing theory is analytical study of behavior of queues, and has been an effective tool for studying throughput, response, and other performance measures for computer systems and communication systems. An analytic model based on queueing theory provides a reasonably good projection for the behavior of real-world complex systems, though a target system need to be simplified for mathematical analysis assuming stationarity, independence, ergodicity or other mathematical properties.

History of queueing theory is surprisingly old but its practical applications were brought with the advent of digital computers [Kle75]. Queueing theory was one of the few tools available to analyze the performance of computer systems. Computers were in turn required in solving equations or to simulate the behavior of queueing systems.

Queueing theory played an important role for invention of packet switching network that evolved into ARPANET [Kle76], and eventually into the Internet. Packet switching was introduced for bursty data traffic. To support bursty user traffic, the required capacity for a shared channel is considerably smaller than unshared channels that are used for voice communication. A queue is required to resolve conflicts that arises when more than one demands are placed on the channel.

2.2.1 Analytical Models

Queueing theory requires the mean capacity C of the system must exceed the average arrival rate R. That is, $R < C$. The utilization factor ρ of a queueing system is R/C so that the range of stability is $0 \leq \rho < 1$. Similarly, the average arrival rate λ should be less than the average service rate μ, that is, $\lambda < \mu$. The average number of packets $\overline{N}$ in a queueing system can be denoted by λ and the average time T that packets spend in the system:

$$\overline{N} = \lambda T \tag{2.1}$$

Equation 2.1 is known as Little's formula.

Queueing theory is a powerful tool to estimate system performance. For example, the required buffer size of a router can be calculated for given traffic input. However, to solve problems in queueing theory, we need to simplify the probability distribution of the arrival rate and the service time. In most cases, the arrival rate are assumed to be Poisson distribution. The service time is often assumed to be exponential or constant.

The M/M/1 Queue

The M/M/1 queue has exponential interarrival time distribution, exponential service time distribution, and a single server. The average number of packets $\overline{N}$ in the system for the M/M/1 queue is:

$$\overline{N} = \frac{\rho}{1-\rho} \tag{2.2}$$

The average time T that packets spend in the system is:

$$T = \frac{1/\mu}{1-\rho} \tag{2.3}$$

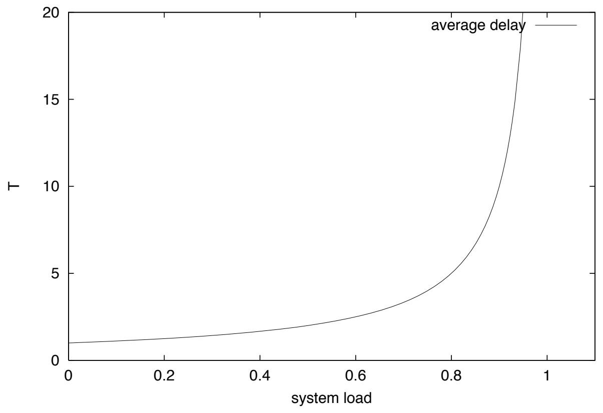


Figure 2.2: average queueing delay as a function of load

The effect is shown in Figure 2.2. As ρ approaches 1, the waiting time in the system grows in an unbounded fashion. Not only M/M/1 but also almost every queueing system shows this behavior.

Impact of Self-Similar Traffic

Performance analysis by queueing theory depends on Poisson packet arrivals. However, in recent years, a number of studies have demonstrated that the traffic pattern under certain conditions is self-similar rather than Poisson [LTWW93]. The implication of self-similar traffic is that traditional queueing theory is not adequate for such environment.

When traffic is Poisson, it has a property that, even if each flow is bursty, an aggregated flow becomes smoother as the scale of aggregation increases by the law of large numbers. It justifies the design of large-scale switches. Self-similar traffic, however, does not have this property. The implication is that traffic aggregation does not reduce the bursty nature of data traffic. In addition, a switch requires drastically larger buffer, especially when the utilization is high. Hence, it is important to understand assumptions and limitations when queueing theory is applied.

2.3 Queueing Disciplines

A queueing discipline has two orthogonal components: a packet scheduler and a buffer manager. A packet scheduler selects a packet to transmit among packets waiting in the queue. A buffer manager selects a packet to discard when the buffer becomes insufficient. It is also possible for a buffer manager to discard packets to signal a congestion notification before it runs out of buffer space.

A FIFO queue is the simplest form of a queueing discipline. The scheduler simply transmits packets in the order they arrive. The buffer manager simply drops an arriving packet when the queue is full.

Most queueing disciplines have a set of internal queues as its components. Arriving packets are classified into one of the internal queues according to a predefined set of rules. Usually, each internal queue is a unit of scheduling, and thus, called a scheduling class or simply a class. A class can be per-flow or aggregated-flows. A per-flow class is for a single micro-flow such as a single TCP session. An aggregated-flow class contains multiple micro-flows sharing some attributes. For example, an aggregate-flow can consist of packets with the same destination address, or packets with the same TCP port number.

A scheduler usually selects a class, rather than a packet, to be served next because there is a restriction on packet re-ordering. A queueing discipline should not change a packet order within a single micro-flow because packet re-ordering has a negative impact to the performance of the existing transport mechanisms (e.g., TCP). Although a transport mechanism is designed to handle packet re-ordering, the performance is considerably degraded since an end node cannot tell whether it is caused by re-ordering or by packet loss when it receives an out-of-order packet. Therefore, packets are usually served by a FIFO policy within a class.

2.3.1 Packet Scheduling

Packet scheduling provides preferential or fair treatment to scheduling classes. An example of preferential scheduling is strict priority queueing [Cob54] in which a class with higher priority is always served first as long as the class has backlogged packets. An example of fair scheduling is fair queueing [Nag87] in which each class is served in a round-robin fashion.

Bandwidth allocation is one of the most important goals of packet scheduling. Fair or preferential bandwidth allocation can be achieved by using an appropriate scheduling method. The same mechanism also isolates a misbehaving flow, and thus, protects other traffic.

When packets are of the same size, simple round-robin scheduling provides fair bandwidth allocation. Weighted-round robin scheduling allows to allocate bandwidth proportional to the weight associated with a class.

To deal with variable packet size, these scheduling policies can be easily extended to take the packet size into account.

It is also possible to allocate bandwidth hierarchically by hierarchical scheduling policies [KKK90, FJ95, SZ97]. For instance, hierarchical round-robin [KKK90] has a tree of sets of slots, where each set at each level provides round-robin service. A set of slots share bandwidth assigned to its parent slot. Hierarchical sharing allows to distribute bandwidth according to an organizational structure. For example, a scheduler tree can be configured so as to allocate bandwidth share to each division, and then, to each employee. Each division can receive assigned share regardless of the number of employees, still each employee of the same division can have fair share.

Another important goal of scheduling is to control delay and jitter that are critical to emerging real-time applications. It is possible to bound the delay and jitter of a flow by reserving the necessary network resources. Admission control is required to decide whether requested resources can be allocated. It is also needed to regulate the rate of the reserved flow by means of shaping. The incoming rate should be less than the reserved rate to avoid delay caused by the flow's own traffic.

However, benefits of packet scheduling do not come for free. An implementation cost is a crucial factor to a packet scheduler. A scheduler resides on the forwarding path, and thus, the overhead of the scheduler is directly added to latency. A scheduling algorithm which is efficient with a small number of classes could have a performance problem with a large number of classes. It is also important to consider the scheduling overhead relative to the total forwarding overhead. Because the link speed of a network varies from a 14.4k modem line to 2.4G WDM, requirements for a scheduling cost are completely different for a different speed range. An scheduling algorithm may need to be implemented in hardware or by parallel processing for high speed routers.

2.3.2 Buffer Management

Packet switching network is based on statistical multiplexing, and thus, packet loss is inevitable when arriving traffic exceeds the capacity of the output link. Routers are forced to discard packets under heavy congestion.

A traditional FIFO queue drops arriving packets when the queue is completely full. This packet drop policy is called Drop-Tail. Drop-Tail, although it is simple, has several drawbacks.

A misbehaving source that keeps the buffer always full can deprive bandwidth of other flows sharing the same queue. A queueing discipline which has independent queues for scheduling classes can isolate such a negative effect from other classes.

Some applications are more resilient to packet loss than other applications. For instance, a voice conversation over the Internet can tolerate infrequent packet loss. It is also useful to have different drop precedence within a single application. For example, if a video stream is encoded by a motion prediction method, the video quality is less susceptible to loss of B-frames than I-frames. Note that, even some frames are lost, successfully arriving frames are still expected to come in order. That is, it has drop precedence within a single scheduling class. Also, drop precedence can provide a way to control bandwidth allocation for best-effort traffic by assigning low drop precedence to packets within a user profile and high drop precedence to packets out of the user profile [CF98].

The Drop-tail policy is easy to implement since the dropper can simply discard an incoming packet without touching the contents of the queue. However, Drop-tail has an undesirable side effect in which a receiver cannot notice packet loss when the last packets from a sending window are dropped. On the other hand, the receiver finds a hole if successive packets arrive, and can take an action to ask the sender for retransmission.

This observation leads to the Drop-Head policy [LNO96] in which a packet is dropped from the head of the queue in order to improve loss recovery time for TCP and other transport protocols. The cost of Drop-head is marginal

The Drop-Random policy selects packets randomly within a queue in the hope that flows sharing the queue have the drop probability proportional to their queue occupancy.

A proactive buffer management such as RED (Random Early Detection) [FJ93, BCC^{+}98] discards packets before the buffer becomes completely full. RED drops packets stochastically according to the average queue length in order to signal a congestion sign to flows that share the queue. The method has an effect to avoid traffic synchronization in which many TCPs lose packets at one time [FJ91], and it also makes TCPs keep the queue length short.

2.4 Performance Guarantee

Nagle proposed fair queueing in which N independent queues are assigned to N flows and served in a round-robin fashion [Nag87]. Demers *et al.* extends fair queueing from packet-by-packet round-robin to bit-by-bit round-robin in order to take packet size into consideration [DKS89]. To emulate bit-by-bit round-robin scheduling in packet network, virtual finishing time is calculated for an arriving packet and the scheduler always serves the packet with the minimum virtual finishing time. Finishing time is computed as follows. Let $R(t)$ denote the number of rounds for time t, and P_i^α denote the transmission time of packet i for queue α. $R(t)$ is monotonically increasing virtual time and P_i^α is packet size normalized to the output link rate. We can calculate transmission start time S_i^α and finish time F_i^α of packet i for queue α when an packet arrives.

$$\begin{aligned} F_i^\alpha &= S_i^\alpha + P_i^\alpha \\ S_i^\alpha &= max[F_{i-1}^\alpha, R(t_i^\alpha)] \end{aligned} \tag{2.4}$$

Parekh proved that, if the sending rate of a flow is regulated by a token bucket, the worst-case delay is bounded in arbitrary topology networks of WFQ [Par92]. When each flow i is assigned weight ϕ_i for output link rate C, flow i is guaranteed a rate of

$$g_i = \frac{\phi_i}{\sum_j \phi_j} C \tag{2.5}$$

Assume the sending rate of flow i is constrained by a token bucket with rate r_i and depth B_i. In a fluid model such as bit-by-bit round-robin, the maximum delay D_i experienced by flow i is bounded by

$$D_i \leq \frac{B_i}{r_i} \tag{2.6}$$

In a WFQ model in which packetization is taken into account,

$$D_i \leq \frac{B_i}{r_i} + \frac{(h_i - 1)l_i}{r_i} + \sum_{m=1}^{h_i} \frac{l_{max}}{C_m} \tag{2.7}$$

Here, h_i is the total number of hops, l_i is the maximum packet size of the flow, l_{max} is the maximum packet size permitted in the network. The first term accounts for delay due to the token bucket size. The second term reflects packetization delay within the flow. The last term accounts for head-of-line blocking at each hop.

Equation (2.7) has important implications; strong guarantees on delay bounds can be achieved, regardless of the behavior of other traffic. The maximum buffer required at each node is proportional to the maximum

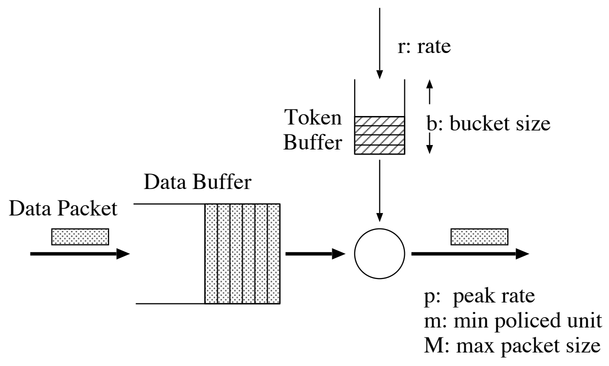


Figure 2.3: IntServ token bucket parameters

delay. In particular, it approaches $r_i D_i$. Equation (2.7) is basis for delay guarantee in the Integrated Services model.

Other queueing disciplines that bound queueing delay are essentially based on the same principle. They differ in degree of guarantee, implementation costs, required buffer size and other factors [ZK91].

2.5 Quality of Service in the Internet

2.5.1 IntServ/RSVP

Traditionally, IP-based internets have provided a simple best-effort service to all users. As the Internet becomes widespread, there are demands to support a variety of traffic with a variety of QoS requirements. To provide QoS transport over the Internet, IETF developed a suite of standards, called Integrated Services (IntServ) model [BCS94].

The IntServ model extends the traditional Internet architecture to support real-time services. The model integrates both non-real time traffic and real-time traffic in the Internet, and assumes real-time services require explicit resource reservation. A fundamental change to the Internet model is that routers need to keep flow-specific state for real-time services.

The IntServ model assumes that the sending rate of a real-time service is regulated by a token bucket, and it guarantees a delay bound using Parekh's theoretical result. The token bucket parameters are illustrated in Figure 2.3. A guaranteed service in the IntServ model provides an upper bound on a worst case delay. In the guaranteed service model, the maximum end-to-end queueing delay is calculated from the token bucket parameters, r and

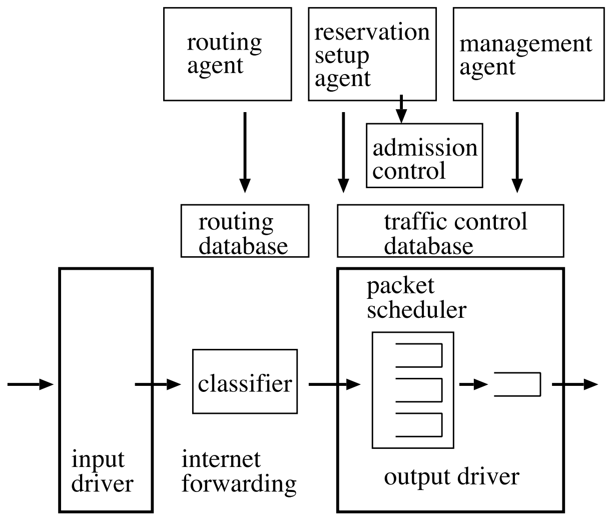


Figure 2.4: IntServ reference implementation model

b, in TSpec and the requested rate, R, in RSpec [SPG97]. The equation is derived from Parekh's equation (2.7).

$$\begin{aligned} D &\leq \frac{b}{R} + \frac{C_{tot}}{R} + D_{tot} \\ C_{tot} &= \sum_h C_h \\ D_{tot} &= \sum_h D_h \end{aligned} \tag{2.8}$$

C_h and D_h are rate-proportional and constant error terms at node h along the path.

A controlled-load service is also defined to provide low delay but without an upper bound. It is intended to support a broad class of applications which have been developed for use in today's Internet, but are highly sensitive to overloaded conditions. The guaranteed service defines a theoretical reference for the IntServ model and the controlled-load service provides a more practical compromise for implementations.

A reference implementation model of IntServ described in [BCS94] includes four components: packet scheduler, classifier, admission control and a reservation setup protocol as shown in Figure 2.4. Admission control implements the decision algorithm to grant a QoS request made by the setup protocol. If the reservation is granted, the packet scheduler is configured to provide the requested QoS to the reserved flow. The classifier is also configured to select packets belonging to the reserved flow, and map them into the corresponding scheduling class.

To request a guaranteed or controlled-load service, an application must specify the desired QoS. A list of QoS parameters is called FlowSpec. FlowSpec consists of TSpec and RSpec. Tspec provides a traffic spec represented by the token bucket parameters. Rspec provides a service spec represented by rate R and slack S. The FlowSpec is carried by the reservation setup protocol, and passed to admission control to test for acceptability, and ultimately used to parameterize the packet scheduling mechanism. The application also must specify filters to select packets for the reservation. A list of filters is called FilterSpec, and used to set up the classifier.

RSVP

Resource ReSerVation Protocol (RSVP) [ZDE+93] is a setup protocol designed for IntServ. The RSVP protocol is used by a host to request specific QoS from the network for particular application data streams or flows. RSVP is also used by routers to deliver QoS requests to all nodes along the path of the flows and to establish and maintain state to provide the requested service. RSVP requests will generally result in resources being reserved in each node along the data path.

The features of RSVP are:

Unicast and Multicast RSVP makes reservations for both unicast and multicast.

Simplex A reservation is unidirectional.

Receiver-initiated reservation The receiver of a flow requests a reservation.

Soft state RSVP maintains soft state in routers and host, providing graceful support for dynamic membership changes and automatic adaptation to routing changes.

Independent of routing RSVP is not a routing protocol but depends upon present and future routing protocols.

Independent of traffic control and policy control RSVP transports and maintains traffic control and policy control parameters that are opaque to RSVP.

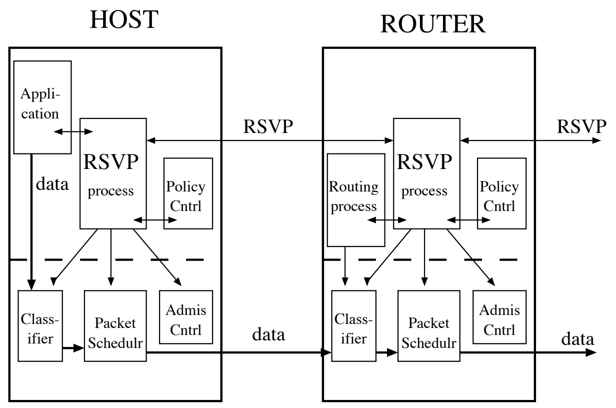


Figure 2.5: RSVP implementation model

Providing different reservation styles RSVP provides several reservation models or styles to fit a variety of applications.

Transparent operation through non-RSVP routers RSVP protocol is designed to be transparent to routers that do not support RSVP, which allows incremental deployment of RSVP-ready routers.

Support for IPv4 and IPv6 RSVP supports both IPv4 and IPv6.

We do not go into the details of RSVP but briefly review how it interacts with the traffic control module. RSVP itself transfers and manipulates QoS and policy control parameters as opaque data, passing them to the appropriate traffic control and policy control modules for interpretation. The structure and contents of the QoS parameters are documented in a specification [Wro97].

Quality of service is implemented for a particular data flow by mechanisms collectively called *traffic control.* Traffic control implements QoS service models defined by IntServ, and follows the IntServ reference implementation model in Figure 2.4. These mechanisms include (1) a packet classifier, (2) admission control, and (3) a packet scheduler as shown in Figure 2.5. The packet classifier determines the QoS class for each packet. For each outgoing interface, the packet scheduler achieves the promised QoS.

During reservation setup, an RSVP QoS request is passed to two local decision modules, admission control and policy control. Admission control determines whether the node has sufficient available resources to supply the requested QoS. Policy control determines whether the user has administrative permission to make the reservation. If both checks succeed, parameters are set in the packet classifier and in the packet scheduler to obtain the desired QoS. If either check fails, the RSVP program returns an error notification to the application process that originated the request.

Problems of RSVP

IntServ and RSVP have contributed to the Internet community to understand various problems in providing QoS. However, there are several problems about the wide-scale deployment of RSVP.

The first problem is scalability. RSVP requires routers to maintain per-flow state for each reservation. It would not be a problem for a small or medium-scale network but routers in a backbone network need to maintain a large number of reservation states. RSVP could introduce a negative impact on performance of high-speed backbone routers.

Another problem is that the development of RSVP is research oriented and several practical issues are left behind. The issues include business models for ISP, policy control to authorize reservations, accounting, operational complexity, implementation costs, and signaling support by applications.

2.5.2 DiffServ

IntServ aims to provide end-to-end guaranteed services on a per-flow basis. DiffServ, however, is intended to provide coarser level of service differentiation to a small number of traffic classes.

DiffServ makes a clear distinction between edge routers and core routers based on a concept similar to the Internet end-to-end model, that is, the inside of the network should be simple, and complex functions should be placed at end nodes.

Edge routers are located at network boundaries, and responsible for traffic conditioning. Traffic conditioning operates on incoming packets, meters users' traffic, and marks a Differentiated Services Code Point (DSCP) into the header of a packet according to the user contract. Traffic conditioning may also perform dropping or shaping.

Core routers are located inside the network, and responsible for providing different treatment for different traffic classes. Core routers implement a limited set of scheduling classes called Per-Hop Behavior (PHB). The DSCP set by an edge router is used to select one of PHBs.

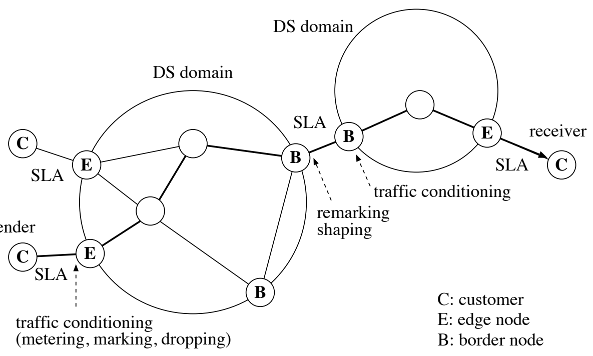


Figure 2.6: DS domain

In the DiffServ architecture, only edge routers maintain per-flow state to perform traffic conditioning. Core routers always handle traffic as an aggregate. Hence, the architecture is considered scalable. Traffic conditioning are configured so as to meet the user contract, and no signaling is required to provide service differentiation.

The concept of DiffServ lays emphasis on provisioning; coarse grained control should work reasonably well as long as network resources are well provisioned.

DS Domain

A DS domain is a contiguous set of DS nodes which operate with a common service policy and PHB definitions. A DS domain normally consists of one or more networks under the same administration; for example, an organization's intranet or an ISP. The administration of the domain is responsible for ensuring that adequate resources are provisioned and/or reserved to support the differentiated services offered by the domain.

DS CodePoint

Each packet entering a DS domain is assigned a value called a Differentiated Services Code Point (DSCP). The assigned value is written into the DS field of the packet header. For IPv4, the DS field is in the TOS field in the IPv4 header as shown in Figure 2.7; for IPv6, it is in the Traffic Class field in the IPv6 header. The DS field is 6 bits long and the remaining two bits are currently unused.

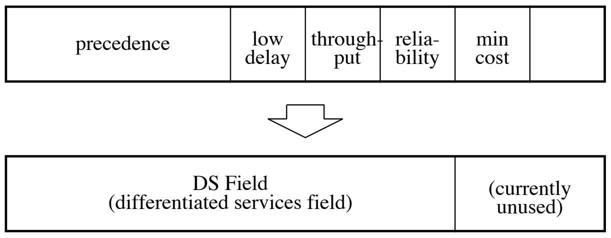


Figure 2.7: diffserv field in the IPv4 TOS octet

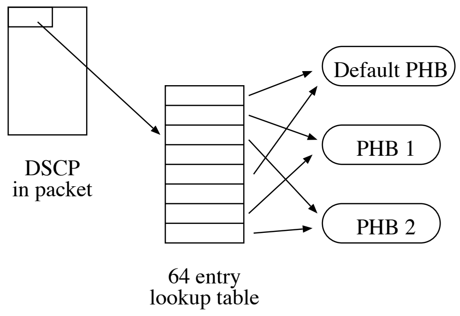


Figure 2.8: Per Hop Behavior

IETF defines a small set of standard DSCPs for interoperability among different DS domains. However, a DS domain is free to use non-standard DSCPs within the domain as long as packets are remarked when they leave the DS domain.

Per-Hop Behavior

A per-hop behavior (PHB) is a description of the externally observable forwarding behavior of a DS node applied to a particular DS behavior aggregate. DS nodes decide how the forwarding is performed on a per-hop basis according to the DSCP values in packets. The concept of a PHB is to define the minimum requirements for forwarding a particular behavior aggregate, still router vendors are allowed to use proprietary algorithms to realize it.

A DSCP selects one of PHBs supported in a DS node. At most 64 different PHBs can be supported by a given DS node since the DS field is 6 bits. Different DSCPs can be mapped to the same PHB as long as the resulting forwarding behavior is in accordance with the service agreement.

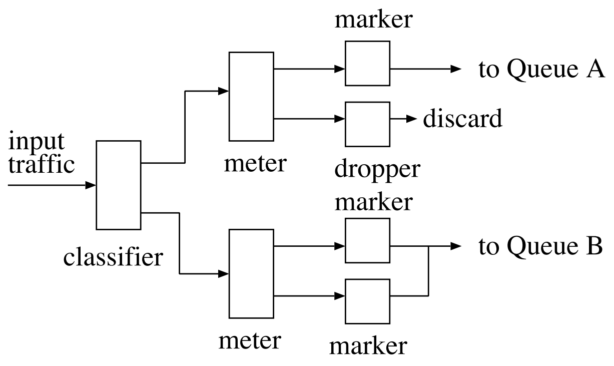


Figure 2.9: an example traffic conditioning block

Mapping from a DSCP value to a PHB can be implemented as a 64-entry lookup table as shown in Figure 2.8.

Traffic Conditioning

Traffic conditioning performs metering, shaping, policing and/or re-marking to ensure that the traffic entering the DS domain conforms to the rules specified in the user contract, in accordance with the domain's service provisioning policy. The extent of traffic conditioning required is dependent on the specifics of the service offering, and may range from simple codepoint re-marking to complex policing and shaping operations.

A traffic conditioning block is a self-contained functional block used to implement some desired network policy. A traffic conditioning block consists of a number of elements such as meters, markers and droppers. Figure 2.9 shows an example of a traffic conditioner block.

Reference Implementation Model

Figure 2.10 shows a conceptual implementation model of a diffserv node. The diffserv functions can be divided into two blocks; traffic conditioning at the ingress interface and PHBs at the egress interface.

Incoming traffic is conditioned at the ingress interface. Packets are first classified and fed into the corresponding traffic conditioning block. A traffic conditioning block meters users' traffic, and takes appropriate traffic conditioning actions.

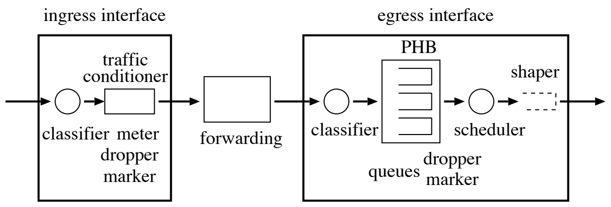


Figure 2.10: DiffServ reference implementation model

At the egress interface, a set of components are organized to form a queueing discipline that realizes PHBs. Packets are classified by DSCP values and put into one of the scheduling classes.

2.6 Summary

Queueing is a basis of packet switching, and plays an important role in packet delivery systems in the Internet. The behavior of queues has been theoretically studied to project statistical system performance. A queueing discipline has two orthogonal components: a packet scheduler and a buffer manager. A large number of queueing disciplines has been proposed to provide quality-of-service in the Internet. There are two models to provide QoS in the Internet: IntServ and DiffServ. Queueing is an essential component in the design of IntServ and DiffServ.

Chapter 3

QoS System Architecture

In this section, we will discuss system architecture to support QoS. A QoS system, by definition, provides different service levels to different users. A service level is achieved by mechanisms and provisioning, and a mechanism consists of different functional components.

There are many different types of QoS mechanisms and it is believed that no single mechanism will likely meet the needs of all applications. Therefore, the architecture of a QoS system should be flexible in order to allow users to build their mechanisms by combining functional components.

3.1 QoS System Design

A QoS system should provide a general-purpose framework within which different QoS mechanisms are implemented. QoS mechanisms can be divided into two types: forwarding mechanisms and management mechanisms. Forwarding mechanisms reside on the packet forwarding path of a router, and work on virtually every packet. Queueing disciplines and traffic conditioners are forwarding mechanisms. On the other hand, management mechanisms control forwarding mechanisms from outside of the forwarding path. Admission control and policy control are management mechanisms.

Forwarding mechanisms and management mechanisms are quite different in their design. Performance is given priority in forwarding mechanisms so that forwarding mechanisms should be simple and efficient, and must be implemented in the kernel. Management mechanisms, on the other hand, take care of complex procedures to control simple forwarding mechanisms. Most of management mechanisms are not too sensitive to performance so that they are suitable to be implemented in the user space. Thus, the architecture of a QoS system needs to be built in such a way that forwarding mechanisms can be optimized for performance and management mechanisms can be extended for more functionality.

The design of a QoS system can be logically divided into three major categories. These three categories are

1. QoS system framework
2. QoS forwarding mechanisms
3. QoS management mechanisms

A QoS system framework implements interfaces between the existing operating system and QoS components. The QoS system framework itself does not provide any QoS facility but it allows the operating system to handle different mechanisms in a uniform manner. The most important part of the framework design is the abstraction of QoS mechanisms. The abstraction should be flexible to accommodate different QoS mechanisms and should be efficiently implemented within the current operating systems.

QoS forwarding mechanisms perform actual QoS functions. QoS forwarding mechanisms are divided into two functional blocks: one is a traffic conditioning block in the input path and the other is an output queueing block on the output path. These blocks are further divided into components such as classifiers, markers and queueing disciplines. Our architecture does not try to provide a model of each individual component but provides abstractions of larger functional blocks, namely, the traffic conditioning block and the output queueing block as black boxes. The details of components are contained within these black boxes, and they are not exposed to the other part of the kernel.

The core component of QoS forwarding mechanisms is a queueing discipline that is the point of actual service differentiation. Our assumption of the system design is that forwarding performance of the system is much faster than the link speed so that packet scheduling is effective only at output queueing. Thus, at the input interface, a series of QoS processing can be sequentially applied to a packet in the traffic conditioning block. On the other hand, enqueue processing and dequeue processing are asynchronous in the output queueing block. As a result, the traffic conditioning block can be modeled as a sequential procedure but the output queueing block should be modeled as an event-driven state-machine.

QoS management mechanisms are a set of tools and libraries used for traffic management. For example, a setup tool is needed to configure queueing disciplines by combining QoS components and setting appropriate parameters to the components. A monitoring tool is indispensable for operation. A sophisticated QoS manager requires admission control and policy control so that it needs a library to interface with admission control and policy control. Simple management tools are useful in the early stage of QoS deployment but, as the technology matures, more elaborate systems will be used in this area.

In fact, the focus of the ALTQ development has been moving from the system framework to the forwarding mechanisms, and then, to the management systems. The initial focus was the system framework in order to prove our QoS abstraction in BSD UNIX. As researchers started using ALTQ for various research experiments, the focus moved to the forwarding mechanisms, that is, the performance of the QoS functions, new functions (e.g., diffserv support) and the flexibility in the design and setting. Then, as the ALTQ users grow in size and diversity, the importance of the management tools is increasing.

3.2 ALTQ Design Overview

3.2.1 Goals

Our primary objective in creating ALTQ is to provide a flexible platform that can support various types of QoS functions in networking. QoS functions are not only for real-time applications such as audio and video but required to manage any network resource in a predictable manner.

The goals of the ALTQ system are:

1. To find proper abstractions of kernel components to support a wide variety of QoS functions. It also addresses the problems in the current operating system and hardware designs to support QoS functions.

2. To provide a flexible and well-engineered platform for QoS related research:

    - ALTQ allows queueing researchers to easily implement new queueing disciplines without knowing the details of kernel programming.
    - ALTQ provides missing components to developers of QoS based systems that assume a traffic control support in the underlying platform. RSVP and Bandwidth-Broker are examples of such systems.

3. To provide a set of tools to gain operational experiences. It is essential to the Internet research to obtain feedback from field experiences. ALTQ allows administrators to try out new ideas to manage traffic.

These goals are closely related still each has different focus. The basic design of the framework should be simple and generic but implementing it requires dirty work in order to hide various implementation issues. Flexibility and rich interfaces are important for research but robustness and simplicity are required for operation.

The ALTQ system consists of the three major components. The framework at the bottom takes care of interfaces to the operating systems in order to use QoS mechanisms. The QoS forwarding mechanisms actually provide QoS. The management mechanisms in the user space take care of interfaces to human or other management systems (e.g., RSVP). One can easily customize part of the ALTQ components and benefit from facilities provided by the other part of ALTQ. To this end, our focus is not only on designing a framework but also on engineering the system as a platform for further research and experiments.

3.2.2 ALTQ Features

ALTQ has the following features:

- A flexible and well-engineered framework to support various types of QoS mechanisms.
- A number of practical queueing disciplines are built into ALTQ.
- Minimal changes to the source code of the existing system.
- No performance impact when an alternative discipline is not enabled.
- Queueing disciplines are implemented as kernel loadable modules.
- ALTQ works as a traffic control module for RSVP.
- ALTQ provides functions required to build a DiffServ network.

3.2.3 System Model

Figure 3.1 shows the QoS system model of ALTQ. QoS Forwarding mechanisms are implemented within the kernel, and divided into two functional blocks: the traffic conditioning block and the output queueing block. QoS

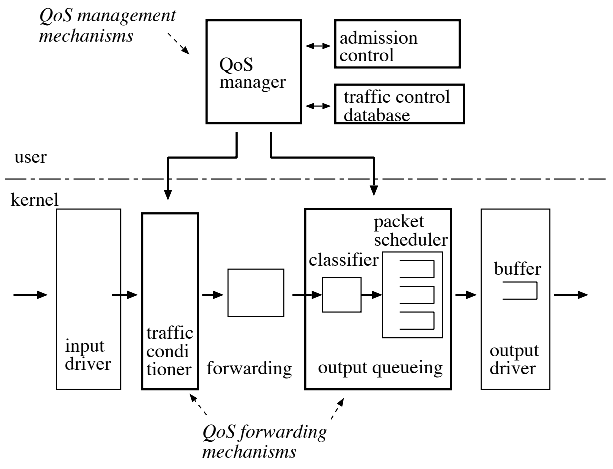


Figure 3.1: ALTQ traffic control model

management mechanisms are implemented in the user space, and controls the forwarding mechanisms. QoS management mechanisms are shown as a QoS manager in the figure but they could be implemented as a set of independent tools. The system framework is not shown in the figure.

The traffic conditioning block implements DiffServ traffic conditioning, and operates on incoming packets. The traffic conditioning block classifies incoming packets, meters users' traffic, and marks or drops packets according to the user contract.

The output queueing block implements a queueing discipline and related functions. A classifier is part of the output queueing block in ALTQ.

These forwarding mechanisms are controlled by the QoS manager. The QoS manger usually reads a configuration file at startup and configures components in the kernel accordingly. It is also possible to dynamically change the state of the components through an API of the QoS manager. The QoS manager is also responsible for admission control and for maintaining the state required for traffic management.

This model is consistent with the IntServ reference implementation model in Figure 2.4 as well as the DiffServ reference implementation model in Figure 2.10.

3.2.4 System Implementation Model

The system model presents functional division of QoS mechanisms. The system implementation model, on the other hand, presents how the system model is integrated into the existing operating system. It is not straightforward to implement the QoS system model into the existing systems since the current operating systems are not designed to support traffic management components. As a result, one need to modify several layers in the network stack that are already quite complex. Modifying the network stack require in-depth knowledge about the operating system, and releasing the modified code and maintaining it require further engineering skills.

ALTQ took a 3-step approach to solve the problem. The first step is to build a flexible framework to accommodate different types of QoS mechanisms. The framework is embedded into the existing operating systems. The second step is to build actual QoS forwarding mechanisms on this framework. The mechanisms include queueing disciplines and DiffServ traffic conditioners. The third step is to build management mechanisms in the user space in order to make use of forwarding mechanisms in the kernel. Management mechanisms are implemented as a QoS manager and monitoring tools.

This design allows to confine the operating system details into the framework, and make forwarding mechanisms easy to implement and more portable. Management functions such as admission control and policy control are separated from the kernel components and implemented in the user space. Consequently, crucial to QoS mechanisms is flexibility of the framework. The design of the framework needs to be general enough to support different mechanisms and flexible enough to accommodate future mechanisms, and also needs to hide all problems in the current operating systems.

Figure 3.2 shows the system implementation model of ALTQ. The traffic conditioning block is attached to *ip_input*. The output queueing block is attached to *struct ifnet* and coexists with the traditional FIFO queue that is already in the system. A QoS manager controls the traffic conditioning block and the output queueing block through the device interfaces. A QoS manager could be a stand-alone manager, or a more sophisticated coordinator such as a RSVP manager and a Bandwidth-Broker.

3.2.5 Implementation Strategy

Our design policy is to make minimal changes to the existing kernel, but it turns out that we have to modify several different parts in the existing kernel. As we will review in Section 4.1, the current operating system does

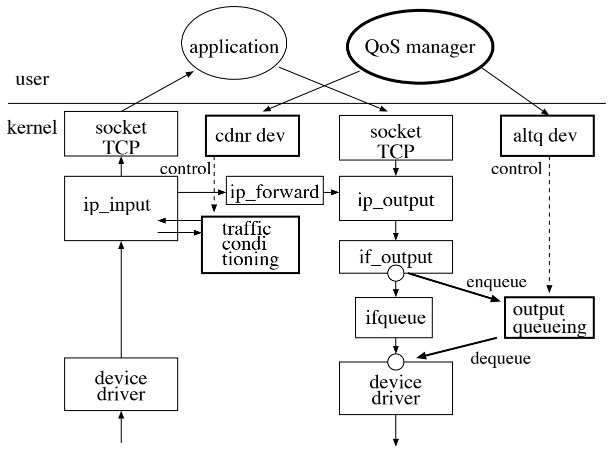


Figure 3.2: ALTQ system implementation model

not have enough abstraction in its queueing operations to support different types of queueing disciplines, and there are problems in device drivers that need to be fixed.

Although we need to modify drivers, it is not practical to modify all the existing drivers at a time. Therefore, we took an approach that allows both modified drivers and unmodified drivers to coexist so that we can modify only drivers we need, and incrementally add supported drivers.

We expect the number of queueing disciplines will increase over time. Therefore, queueing disciplines are handled through the discipline switch table without hard-coding a specific queueing discipline. Adding a new discipline requires only adding an entry to the table.

3.3 Summary

ALTQ has three goals; designing a framework, providing a research platform, and providing tools for experiments. These goals are closely related still each has different focus.

The system model of ALTQ consists of three major categories: the framework, forwarding mechanisms, and management mechanisms. The model is compatible with the reference implementation models of IntServ and DiffServ. Based on this system model, ALTQ is implemented onto the existing operating systems.

One can easily customize part of the ALTQ components and benefit from facilities provided by the other part of ALTQ. To this end, our focus is not only on designing a framework but also on engineering the system as a platform for further research and experiments.

Chapter 4

System Framework

The ALTQ framework provides mechanisms to use alternative queueing disciplines and traffic conditioners in BSD UNIX. The framework itself does not provide any traffic management facility but it provides the abstraction of queueing disciplines to the other part of the operating system, and also provides the abstraction of the operating system to queueing disciplines.

4.1 Problems in Current BSD UNIX

There are problems in the current BSD UNIX to support QoS functions, in particular, to support different types of queueing disciplines.

BSD UNIX, like the majority of systems currently in use, assumes FIFO queueing with the Drop-Tail policy, and does not have an abstraction to support other types of queueing disciplines.

In BSD UNIX, network interfaces are abstracted by a structure called *struct ifnet*. A standard output queue, *struct ifqueue if_snd*, is implemented in *struct ifnet*. *if_snd* is manipulated by *IF_ENQUEUE()* and *IF_DEQUEUE()* macros. These macros are used by two functions registered in *struct ifnet*, *if_output* and *if_start*; *if_output* defined for each link type performs the enqueue operation, and *if_start* defined as part of a network device driver performs the dequeue operation [MBKQ96].

One might think that just replacing *IF_ENQUEUE()* and *IF_DEQUEUE()* will suffice to implement a new queueing discipline but, unfortunately, it is not the case. The problem is that the queueing operations used inside the kernel are not only enqueueing and dequeueing. In addition, surprisingly many parts of the kernel code assume FIFO queueing and the *ifqueue* structure. There are other problems in device drivers.

4.1.1 Abstraction of Queueing Operations

First, we review the queueing abstraction in the current BSD UNIX, and point out problems. A new set of queueing operations need to be defined in order to interface different types of queueing disciplines into the framework.

Drop-Tail Assumption

The existing code implicitly assumes the Drop-Tail policy. The following code in Figure 4.1 shows a typical enqueue sequence found in *if_output*.

```
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);
    splx(s);
    m_freem(m);
    return(ENOBUFS);
}
IF_ENQUEUE(&ifp->if_snd, m);
if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);
splx(s);
```

Figure 4.1: enqueue operation in *if_output*

The code performs three operations related to queueing.

1. check if the queue is full by *IF_QFULL()*, and if so, drop the arriving packet.
2. enqueue the packet by *IF_ENQUEUE()*.
3. call the device driver to send out the packet unless the driver is already busy.

The code assumes the Drop-Tail policy, that is, the arriving packet is dropped. But decision of dropping and selection of a victim packet should be done by a queueing discipline. Moreover, in a random drop policy, the drop operation often comes after enqueueing an arriving packet. That is, the order of drop decision and victim selection also depends on a queueing discipline. Furthermore, in a non-work conserving queue, enqueueing a packet does not mean the packet is sent out immediately, but rather, the driver should be invoked later at some scheduled timing.

To support non-Drop Tail policies, it is difficult to separate the drop operation from the enqueue operation without knowledge of the structure of the queueing discipline. For example, when a discipline has multiple classes, a drop decision cannot be made until a packet is classified. Thus, the drop operation needs to be part of the enqueue operation.

Poll Operation

There are also problems in *if_start* routines. Some drivers poll at the head of the queue to see if the driver has enough resources (e.g., buffer space and/or DMA descriptors) for the next packet. A typical poll operation found in drivers is shown in Figure 4.2.

```
while (ifp->if_snd.ifq_head != NULL) {
    /*
     * get resources to send a packet
     */

    IF_DEQUEUE(&ifp->if_snd, m);

    /*
     * DMA setup and kick the device
     */
}
```

Figure 4.2: poll operation in *if_start*

Those drivers directly access *if_snd* using different methods since no standard procedure is defined for a poll operation. A queueing discipline could have multiple queues, or could be about to dequeue a packet other than the one at the head of the queue.

Therefore, a poll operation that returns the next packet without removing it should be part of the generic queueing operations. Although it is possible in theory to rewrite all drivers not to use poll operations, it is wise to support a poll operation, considering the labor required to modify the existing drivers. A discipline must guarantee that the polled packet will be returned by the next dequeue operation.

On the other hand, *IF_PREPEND()* is defined in BSD UNIX to add a packet at the head of the queue, but the prepend operation is intended for a FIFO queue and should not be included in the generic queueing operations. Fortunately, the prepend operation is rarely used—with the exception of a few Ethernet drivers. These drivers use *IF_PREPEND()*, when there are not enough DMA descriptors available, to put back a dequeued packet as in Figure 4.3. Such a driver should be modified to use a poll-and-dequeue method instead of a dequeue-and-prepend method.

```
while (ifp->if_snd.ifq_head != NULL) {

    IF_DEQUEUE(&ifp->if_snd, m);

    /*
     * get resources to send a packet
     */
    if (not_enough_resource) {
        IF_PREPEND(&ifp->if_snd, m);
        break;
    }

    /*
     * DMA setup and kick the device
     */
}
```

Figure 4.3: prepend operation in *if_start*

In fact, it is also possible to use only the prepend operation instead of the poll operation. That is, the driver performs a speculative dequeue, and then, prepends the dequeued packet if something goes wrong. However, for some queueing disciplines, it is not so simple to cancel a dequeue operation once the internal state is updated.

Purge Operation

Another problem in *if_start* routines is a purge operation to empty the queue. A dequeue loop as in Figure 4.4 is often used in drivers to empty the queue. However, a non-work conserving queue cannot be emptied by this method since a packet is not dequeued until its departure time. Therefore, the purge operation should be defined and drivers should be modified to use the defined purge operation to empty the queue.

4.1.2 Driver Issues

There are two other issues in the device driver level to efficiently support packet scheduling.

```
/*
 * Purge an interface queue.
 */
while (ifp->if_snd.ifq_head != NULL) {
    IF_DEQUEUE(&ifp->if_snd, m);
    m_freem(m);
}
```

Figure 4.4: purge operation in drivers

Negative effects of long DMA chains

There is a trade-off in setting the transmission buffer size in a network card. If the buffer size is too small, there is a risk of buffer under-run that makes the link under-utilized even when packets are backlogged. Another concern is the overhead of interrupt processing. A larger buffer helps to reduce the number of interrupts for those network cards which interrupt only when all transmission is complete. On the other hand, if the buffer is too large, it has negative effects to packet scheduling.

Many modern network cards support chained DMA, typically, up to 128 or 256 entries. Most network drivers are written to buffer packets as many as possible in order not to under-utilize the link and to reduce the number of interrupts. However, it creates a long waiting queue after packets are scheduled by the packet scheduler, and large buffers in network cards adversely affect packet scheduling. The device buffer has an effect of inserting another FIFO queue beneath a queueing discipline.

An obvious problem is delay caused by a large buffer. Even if the packet scheduler tries to minimize the delay for a certain packet, the packet needs to wait in the device buffer for hundreds of packets to be drained. Thus, delay cannot be controlled if there is a large buffer in the network card.

Another less obvious but more serious problem is bursty dequeues. When the device buffer is large, packets are moved from the queue to the device buffer in a very bursty manner. If the queue gets emptied when a large chunk of packets are dequeued at a time, the packet scheduler loses control. A packet scheduler is effective only when there are backlogged packets in the queue.

These problems are invisible under FIFO, and thus, most drivers are not written to limit the number of packets in the transmission buffer. However, the problem becomes apparent when preferential scheduling is used. The transmission buffer size should be set to the minimum amount that is required to fill up the link. Although it is not easy to automatically detect the appropriate buffer size, the number of packets allowed in the device buffer should be limited to a small number. Many drivers, however, set an excessive buffer size. Hence, it is necessary to have a way to limit the number of packets (or bytes) that are buffered in the card.

Note that the number of packets in the device buffer should be reflected to the delay calculation. In literature, the existence of device buffer is usually not assumed but buffer is necessary in practice to avoid under-utilization of the link and excessive interrupts.

Use of Interrupts

Some cards generate interrupts every time a packet is transmitted, some generate interrupts only when the buffer becomes empty, and some allows a driver to program when to interrupt. Many network cards with DMA capability distinguish "DMA complete" and "transmission complete". How often the hardware generates interrupts is device-dependent.

It is generally believed that a smart network card should reduce interrupts to alleviate CPU burden. However, a queueing discipline can have finer grained control with frequent interrupts; it is a trade-off between CPU control and CPU load. For example, there is an interesting report that, under some conditions, the precision of CBQ's control is considerably improved with an old NE2000 card that interrupts a lot and has small buffers.

A packet scheduler needs to make use of transmission complete interrupts to transmit backlogged packet in the queue. This is essential to making a queueing discipline work-conserving.

Most drivers in BSD UNIX call *if_start* to transmit the next packet from a transmission complete interrupt. Then, a dequeue operation is executed from *if_start*. However, there is no clear rule regarding how often *if_start* should be called, and it is currently driver-dependent.

There is a potential problem with regard to limiting packets in the transmission buffer. When we limit the number of packets that are buffered in the card to N, the hardware will generate transmission complete interrupts at least for every N packets, and the driver calls *if_start* accordingly. However, drivers are often tuned with a large buffer and not optimized to use a small buffer.

Another concern is that, in the current model, it is not clear when the driver should call *if_start*. *if_done* hook was once added to *struct ifnet* in 4.3BSD-reno in order to provide a way to notify a transmission complete event to an upper layer but it was never supported by drivers [MBKQ96].

It might be a good idea to reconsider the use of this hook.

4.1.3 Passing Classifier Information

Another issue is related to a classifier that divides arriving packets into different classes. A classifier needs to look into the headers of a packet; depending on the configuration, IP header and TCP/UDP header are usually used. The role of a classifier is to perform filter matching to an arriving packet and, if a matching entry (class) is found, returns the matching entry. Thus, a classifier is usually not specific to a queueing discipline, and can be logically separated from a queueing discipline.

A classifier looks into IP (and optionally TCP/UDP) headers. Therefore, the network layer is the best place to implement a classifier. The problem, however, is that the class attribute given by the classifier is used by a queueing discipline, and thus, should be passed to the queueing discipline. There is no simple way in BSD UNIX to pass the class attribute down to the queueing discipline.

There are at least four ways to do it.

1. Place a classifier within *ip_output*, and add a field to *struct mbuf* in order to set the class attribute in a packet. A class attribute belongs to the packet and it is natural to tag the class attribute to the packet itself. The mechanism to tag attributes to a packet should be generic since there are other places in the kernel that can benefit from this mechanism. This method, although it would be a good way for a long run, changes *struct mbuf* that is used throughout the kernel. The impact of this change is fairly large.

2. Place a classifier within *ip_output*, and add an argument to *if_output* to pass the class attribute. This method changes the interface of *if_output*. Again, the impact of change is large. Also, it is not generic to add an argument just for classifiers.

3. Place a classifier within *if_output* and pass the class attribute to a queueing discipline. This method confines the change for classifiers within *if_output*. This method is not clean from the architectural point of view since it requires to look into headers of upper layers from the link layer (layer violation).

4. Place a classifier within a queueing discipline. *if_output* needs to pass a pointer to the IP header in a packet since *if_output* prepends a link-level header before calling the enqueue operation. This method also confines

the change for classifiers within *if_output* and has a layer violation problem. The problem of this method is that there is a risk that the pointer to the IP header becomes stale. There is no guarantee in BSD UNIX that a pointer into *mbuf* data is valid since *mbuf* is allowed to be re-packed.

Each of the four approaches has advantages and disadvantages. We chose the third approach because the impact to other part of the kernel is smaller than the first two, and safer than the last. However, we also hope that the first approach will be incorporated into the base operating system and available in the future.

4.1.4 Summary

The current BSD UNIX has a number of problems to support different types of queueing disciplines.

- A set of queueing operations need to be defined. The operations needed are ENQUEUE, DEQUEUE, POLL, and PURGE. Packet dropping should be part of the ENQUEUE operation.
- *if_output* needs to be modified to allow different drop policies.
- Direct references to *if_snd* in drivers should be replaced by the POLL operation.
- *IF_PREPEND()* should be removed from drivers, and the drivers should be modified to use the poll-and-dequeue policy.
- The PURGE operation should be used to empty a queue.
- Drivers should limit the number of packets in the transmission buffer unless the queueing discipline is FIFO.
- Drivers should be tuned to work efficiently with a small transmission buffer size.
- A mechanism is required to pass class attributes to a queueing discipline.

Correcting these problems requires a proper design and in-depth knowledge of kernel internals, which underlines the importance of a framework in support of queueing disciplines.

4.2 ALTQ framework

The ALTQ framework provides mechanisms to support QoS functions in the exiting BSD UNIX kernel. The framework provides models of queueing disciplines and traffic conditioning blocks, and defines the operations in order to make use of QoS components from the rest of the kernel.

The ALTQ framework does not try to provide a model of each individual component. The framework uses abstractions of larger functional blocks, namely, output queueing for outgoing packets and traffic conditioning for incoming packets. The details of components are contained within these blocks, and they are not exposed to the other part of the kernel.

The ALTQ framework can be further decomposed into three parts:

1. output queueing support
2. traffic conditioning support
3. module management support

The output queueing support provides mechanisms to support various queueing disciplines for output interfaces. The traffic conditioning support provides mechanisms to support various types of traffic conditioning for input interfaces. The module management support provides mechanisms to control QoS mechanisms from the outside. The output queueing support is the main part of the framework. As explained in the previous section, it requires a number of changes in the already complicated networking code. A simple system model is needed but its design should be thought out carefully.

The ALTQ framework described here is the second generation of the framework design used in version 3.0 of the ALTQ release. In the first generation design, the focus was for research use. The details are described in [Cho98]. However, as ALTQ has evolved into a traffic management system for wider use, the requirements of the framework are gradually changed. The biggest problem is that changes are required to every network device driver but it is impossible to update hundreds of the existing drivers. The maintenance of the drivers becomes too time-consuming. The framework needed to be redesigned to solve the maintenance issues. The changes from the original framework design are:

- output queueing is contained within an output queue structure.
- a complete set of macros are defined to manipulate an output queue structure. the new macros hide ALTQ specific implementation issues. ALTQ specific changes are removed as much as possible from drivers.

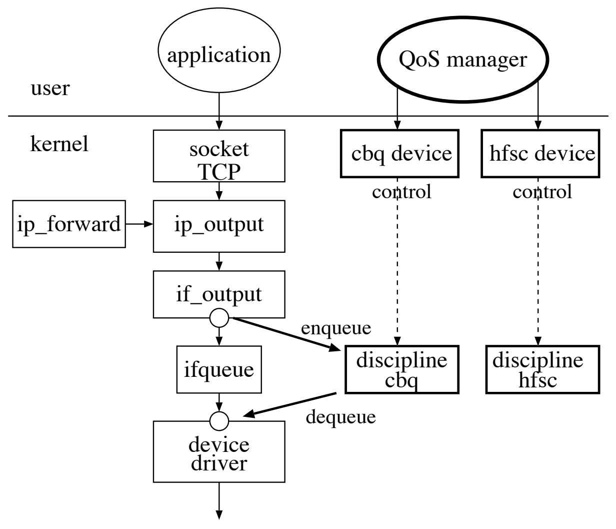


Figure 4.5: output queueing support

- required changes in drivers are reduced. how to change a large number of drivers in an incremental manner is addressed.
- separation of the model and implementation. for example, a dequeue operation and a poll operation are separated in the model, but implemented in the same function.
- a token bucket regulator is used to control the behavior of a driver in a device independent way. a token bucket regulator also makes it easier to tune the behavior of each driver.
- separation of a classifier and a queueing discipline.

4.3 Output Queueing Support

The output queueing support provides mechanisms to support alternative queueing disciplines for the output queue of a network interface. The output queueing support can be viewed as a switch to a queueing discipline, and it coexists with the existing output queue structure as shown in Figure 4.5. When ALTQ is enabled, alternative enqueue and dequeue functions are used instead of the original enqueue and dequeue operations. As a result,

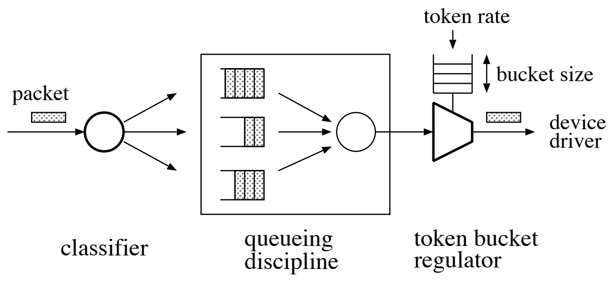


Figure 4.6: output queueing model consists of classifier, queueing discipline, and token bucket regulator

packets are stored in an alternative queueing discipline, and controlled by this queueing discipline.

4.3.1 Output Queue Model

The output queue model in ALTQ consists of three independent components.

1. **Classifier** classifies a packet to a scheduling class based on predefined rules.
2. **Queueing discipline** implements packet scheduling and buffer management algorithms.
3. **Token-bucket regulator** limits the amount of packets that a driver can dequeue at a time.

Figure 4.6 illustrates the relation of these components. A classifier assigns each arriving packet to a scheduling class, subsequently enables a queueing discipline to differentiatedly handle the packet. A classifier is made independent from a queueing discipline because the classification algorithm of a classifier and the scheduling algorithm of a queueing discipline are independent. They share only mapping of classifier filters to scheduling classes. This separation allows us to easily optimize or extend our classifier algorithm in the future.

A token bucket regulator controls the behavior of network device drivers in a device-independent manner. In the earlier implementation of ALTQ, each driver was modified not to buffer too many packets at a time. However, there were too many drivers to be modified. In addition, it is impossible to

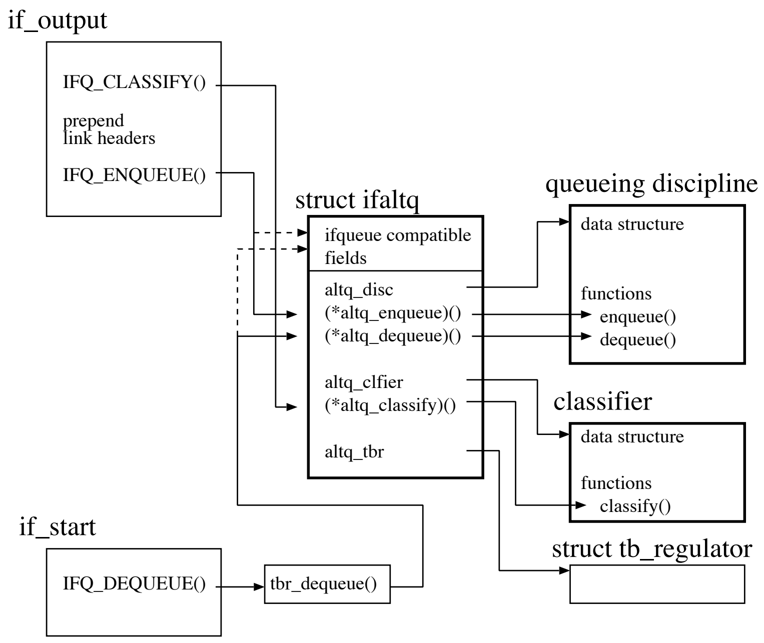


Figure 4.7: output queue implementation in ALTQ

pre-define the burst size allowed for a driver since the required burst size depends also on the CPU power and traffic. The token bucket regulator allows to tune the behavior of a driver by adjusting the bucket size parameter.

Another advantage of the token bucket regulator mechanism is discipline-independence. It is difficult to debug the behavior of a queueing discipline if there is interference between a driver and a discipline. A token bucket regulator decouples a driver and a discipline so that the driver can be independently tuned first using a simple discipline, and then, applied to a more sophisticated disciplines.

Figure 4.7 shows the implementation overview of the output queue model. *ifaltq* structure at the center is an abstracted output queue structure. *struct ifaltq* works as an indirect reference to entities of the output queue components, that is, classifier, queueing discipline, and token bucket regulator.

All operations to the output queue work on *struct ifaltq*, and then, they are redirected to the entity of an output queue components. In Figure 4.7, there are three operations which manipulate the output queue. On the enqueue side, (1) *IFQ_CLASSIFY()* calls a classifier function. The result is stored as a packet attribute and passed to the enqueue operation. (2)

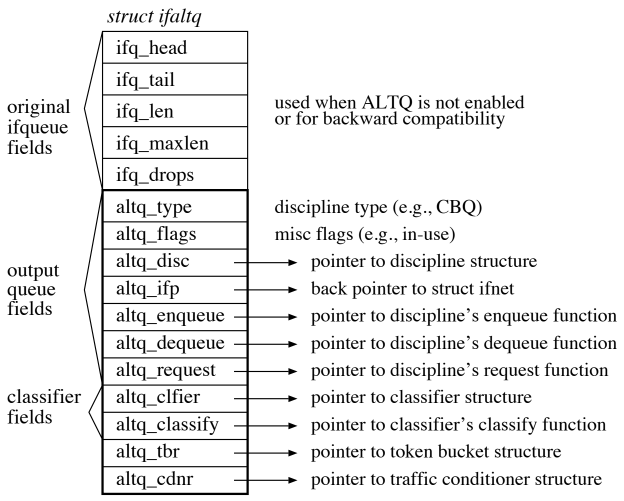

struct ifaltq
original ifqueue fields
ifq_head
ifq_tail
ifq_len
ifq_maxlen
ifq_drops
used when ALTQ is not enabled or for backward compatibility
output queue fields
altq_type
altq_flags
altq_disc
altq_ifp
altq_enqueue
altq_dequeue
altq_request
classifier fields
altq_clfier
altq_classify
altq_tbr
altq_cdnr
discipline type (e.g., CBQ)
misc flags (e.g., in-use)
pointer to discipline structure
back pointer to struct ifnet
pointer to discipline's enqueue function
pointer to discipline's dequeue function
pointer to discipline's request function
pointer to classifier structure
pointer to classifier's classify function
pointer to token bucket structure
pointer to traffic conditioner structure

Figure 4.8: output queue structure

IFQ_ENQUEUE() calls the enqueue function of the queueing discipline. (3) On the dequeue side, *IFQ_DEQUEUE()* checks the token bucket regulator first. If there is enough tokens, *IFQ_DEQUEUE()* calls the dequeue function of the queueing discipline. Otherwise, it returns *NULL*.

4.3.2 Output Queue Structure

The output queue structure, *struct ifaltq*, is illustrated in Figure 4.8. The fields in *struct ifaltq* can be divided into 5 groups:

1. fields compatible with struct ifqueue: these fields are compatible with the original output queue structure, *struct ifqueue*, and used when ALTQ is not enabled.

2. alternate queueing discipline related fields: *altq_type* identifies a discipline type such as CBQ and H-FSC. *altq_enqueue* and *altq_dequeue* are the enqueue and dequeue functions of a discipline. *altq_dequeue* works as either dequeue or poll by the second argument. *altq_request* is used for a purge operation but could be used for other purposes. *altq_disc* points to a discipline structure, and it is passed to the discipline functions. *altq_ifp* is a back pointer to *struct ifnet* in order to allow a non-work conserving discipline to call *if_start*.

3. classifier fields: *altq_classify* is the classification function of a classifier. *altq_clfier* points to a classifier structure, and it is passed to *altq_classify*.

4. a pointer to the token bucket regulator structure.

5. a pointer to the top-level traffic conditioner block of the interface. although traffic conditioning does not belong to the output queue, this field is placed here to put the ALTQ related fields together.

4.3.3 Incremental Driver Support

The biggest problem in introducing our output queue model into the existing BSD UNIX is that network device drivers need to be modified but there are too many drivers. It is not practical to update hundreds of drivers at one time. Our approach is to allow both modified drivers and unmodified drivers to coexist. Then, we can modify only drivers we need, and incrementally add supported drivers.

We introduced a new output queue structure, *struct ifaltq*, since the current queue structure, *struct ifqueue*, is used for purposes other than interface queues. The structure type of an output queue in *struct ifnet* is changed from *struct ifqueue* to *struct ifaltq* as shown in Figure 4.9.

```
          ##old-style##                               ##new-style##
                                      |
struct ifnet {                        | struct ifnet {
    ....                              |     ....
                                      |
    struct ifqueue if_snd;            |     struct ifaltq if_snd;
                                      |
    ....                              |     ....
};                                    | };
                                      |
```

Figure 4.9: output queue structure type is changed in *struct ifnet*

struct ifaltq has fields compatible with *ifqueue* as shown in Figure 4.10. It allows the macros for *struct ifqueue* to work with *struct ifaltq*. When the *ifqueue* macros are used, *struct ifaltq* behaves exactly the same as *struct ifqueue*.

A new set of macros to manipulate *struct ifaltq* are defined. The new macros use the compatible macros when ALTQ is not used, and use the ALTQ macros when ALTQ is enabled, Figure 4.11 is a simplified version

```
          ##old-style##                         ##new-style##
                                       |
struct ifqueue {                       | struct ifaltq {
   struct mbuf *ifq_head;              |    struct mbuf *ifq_head;
   struct mbuf *ifq_tail;              |    struct mbuf *ifq_tail;
   int         ifq_len;                |    int         ifq_len;
   int         ifq_maxlen;             |    int         ifq_maxlen;
   int         ifq_drops;              |    int         ifq_drops;
};                                     |    /* altq related fields */
                                       |    ......
                                       | };
                                       |
```

Figure 4.10: compatible fields in *struct ifaltq*

of the new dequeue macro, and illustrates how it switches between the two modes.

```
#define IFQ_DEQUEUE(ifq, m)          \
    if (ALTQ_IS_ENABLED((ifq))       \
        ALTQ_DEQUEUE((ifq), (m));    \
    else                             \
        IF_DEQUEUE((ifq), (m));
```

Figure 4.11: a simplified version of the new dequeue macro

Because the behavior of the system does not change in the compatible mode, incorporating ALTQ has a minimal impact. This approach also makes it possible to fall back to the original queueing if something goes wrong. As a result, the system becomes more reliable, easier to use, and easier to debug.

Network device drivers modified to support ALTQ can be identified by setting a flag bit in *struct ifaltq*. This bit is checked when a discipline is attached.

4.3.4 Queueing Operations

To handle different types of queueing disciplines in a uniform manner, a set of common queueing operations are defined, and implemented as macros. The new macros are designed for ALTQ but they are intended to be general enough for other possible implementations. Once the existing code, especially, drivers are converted to use the new macros, it becomes easy to incorporate yet another output queue model.

Table 4.1 lists the new macros defined to manipulate an output queue.

Table 4.1: new output queue macros

Macro	**Description**
`IFQ_ENQUEUE(ifq, m, pktattr, err)`	enqueue a packet to the queue
`IFQ_DEQUEUE(ifq, m)`	dequeue a packet from the queue
`IFQ_POLL(ifq, m)`	poll the next packet to be dequeued
`IFQ_PURGE(ifq)`	discard all packets in the queue
`IFQ_IS_EMPTY(ifq)`	TRUE if the queue is empty
`IFQ_CLASSIFY(ifq, m, af, pktattr)`	classify a packet
`IFQ_SET_MAXLEN(ifq, len)`	set the queue size limit
`IFQ_INC_LEN(ifq)`	increment the packet count
`IFQ_DEC_LEN(ifq)`	decrement the packet count
`IFQ_INC_DROPS(ifq)`	increment the drop count
`IFQ_SET_READY(ifq)`	indicate the driver is ready for the new model

There are four major operations: enqueue, dequeue, poll and purge. Each queueing discipline should provide these four queueing operations. Then, different queueing disciplines can be handled through a uniform interface.

- *IFQ_ENQUEUE()* adds a packet to the queue. *IFQ_ENQUEUE()* performs 2 actions: (1) dropping a packet if necessary (2) enqueueing the arriving packet. *IFQ_ENQUEUE()* differs from *IF_ENQUEUE()* in semantics; *IFQ_ENQUEUE()* combines enqueue and drop since they cannot be easily separated in many queueing disciplines. An error (*ENOBUFS*) is set when an arriving packet is dropped to signal the drop to upper layers. The *mbuf* is freed in either case; by the driver when it succeeds, and by the discipline otherwise. Thus, the caller must not touch the *mbuf* after *IFQ_ENQUEUE()* is called. *IFQ_ENQUEUE()* in the compatible mode can be written with the old macros as shown in Figure 4.12.

- *IFQ_DEQUEUE()* removes the next packet to send from the queue. It returns NULL when there is no eligible packet to dequeue.

- *IFQ_POLL()* returns the next packet to send without removing it from the queue. The poll operation is intended to be used by drivers to check available resources (e.g., DMA descriptors) for the next packet. It is guaranteed that a following dequeue operation returns the same packet, provided that no other queueing operation is called in between.

```
#define IFQ_ENQUEUE(ifq, m, pktattr, err)       \
do {                                            \
    if (IF_QFULL((ifq))) {                      \
        m_freem((m));                           \
        (err) = ENOBUFS;                        \
        IF_DROP(ifq);                           \
    } else {                                    \
        IF_ENQUEUE((ifq), (m));                 \
        (err) = 0;                              \
    }                                           \
} while (0)
```

Figure 4.12: *IFQ_ENQUEUE()* semantics

- *IFQ_PURGE()* empties the queue. All packets stored in the queueing discipline are discarded. The purge operation is the only way to empty a non-work conserving discipline.

Other than the above four major operations, the following operations are defined.

- *IFQ_IS_EMPTY()* returns *TRUE* when the queue is empty. Note that *IFQ_POLL()* can be used for the same purpose, but *IFQ_POLL()* could be costly for a complex scheduling algorithm since the *IFQ_POLL()* needs to run the scheduling algorithm to select the next packet. On the other hand, *IFQ_EMPTY()* checks only if there is any packet stored in the queue.

- *IFQ_CLASSIFY()* calls the classifier. The result is stored as a packet attribute, and passed to *IFQ_ENQUEUE()*.

- *IFQ_SET_READY()* sets a flag bit to indicate this driver is converted to the new style and supports ALTQ. The flag bit is used by ALTQ to distinguish new-style drivers.

The rest of the macros are for completeness to eliminate direct references to the compatible fields.

- *IFQ_SET_MAXLEN()* sets the queue size limit.

- *IFQ_INC_LEN()* increments the packet count in the queue.

- *IFQ_DEC_LEN()* decrements the packet count in the queue.

- *IFQ_INC_DROPS()* increments the drop count.

4.3.5 Token Bucket Regulator

The purpose of a token bucket regulator is to limit the amount of packets that a driver can dequeue. A token bucket has "token rate" and "bucket size". Tokens accumulate in a bucket at the average "token rate", up to the "bucket size". A driver can dequeue a packet as long as there are positive tokens, and after a packet is dequeued, the size of the packet is subtracted from the tokens. Note that this implementation allows the token to be negative as a deficit in order to make a decision without prior knowledge of the packet size. It differs from a typical token bucket that compares the packet size with the remaining tokens beforehand.

The token bucket regulator is implemented as a wrapper function of the dequeue operation. A simplified version of the dequeue function using a token bucket regulator is shown in Figure 4.13.

```
struct mbuf *
tbr_dequeue(ifq)
    struct ifaltq *ifq;
{
    struct tb_regulator *tbr = ifq->altq_tbr;
    struct mbuf *m;

    update_token(tbr);
    if (tbr->tbr_token <= 0)
        return (NULL);
    if (ALTQ_IS_ENABLED(ifq))
        ALTQ_DEQUEUE(ifq, m);
    else
        IF_DEQUEUE(ifq, m);
    if (m)
        tbr->tbr_token -= m->m_pkthdr.len;
    return (m);
}
```

Figure 4.13: dequeue operation with a token bucket regulator

It is important to understand the roles of "token rate" and "bucket size". The bucket size controls the amount of burst that can dequeued at a time, and controls a greedy device trying dequeue packets as much as possible. This is the primary purpose of the token bucket regulator in ALTQ. Thus, the token rate should be set to the actual maximum transmission rate of the interface.

On the other hand, if the rate is set to a smaller value than the actual transmission rate, the token bucket regulator becomes a shaper that limits the long-term output rate. Another important point is that, when the rate is set to the actual transmission rate or higher, transmission complete interrupts can trigger the next dequeue. However, if the token rate is smaller than the actual transmission rate, the rate limit would be still in effect at the time of transmission complete interrupt, and the rate limiting falls back to the kernel timer to trigger the next dequeue. In order to achieve the target rate under timer-driven rate limiting, the bucket size should be increased to fill the timer interval.

An efficient implementation of a token bucket is important because tokens need to be updated every time packet is dequeued. Our implementation uses a high resolution clock on Intel Pentium and DEC/Compaq alpha architecture but uses *microtime()* on other platforms or on multi-processor systems. Intel Pentium processor has a 64-bit time stamp counter driven by the processor clock, and this counter can be read by a single instruction. If the processor clock is 500MHz, the resolution is 2 nanoseconds. The problem is that processors have different clocks so that the time stamp counter value needs to be normalized to be usable on different machines. Normalization requires expensive multiplications and divisions, and the low order bits are subject to rounding errors. Our approach is the other way around. When the token bucket regulator is configured, the parameters are scaled to the time unit of the processor clock to avoid expensive arithmetic operations at transmission time.

4.3.6 Classifier

A classifier maps a packet to a scheduling class by some form of packet filtering. Different types of classifiers will be used for different purposes. To take the DiffServ model as an example, a MF (multi-field) classifier is used at an edge of a DiffServ domain but a BA (behavior aggregate) classifier is used inside the DiffServ domain. This model also suggests that different classifiers can be used for the same queueing discipline depending on the administrative policy.

Therefore, the ALTQ framework does not assume any classifier algorithm. A classifier is called in *if_output* though a hook in *struct ifaltq*. The framework treats the classification result as an opaque object and, subsequently, the result is passed to the queueing discipline as an argument of the enqueue operation. A classifier is called before link-level headers are prepended to avoid handling various link-level headers to locate the IP header.

It could be possible in the future to implement a classifier in a different location. A possible approach is to integrate a classifier into a part of the existing IP packet filter implemented in the IP layer. However, in order to pass the classification result to the queueing discipline at the interface level, there must be a mechanism to tag the result to the packet itself. It requires modifications to *struct mbuf* to hold a packet attribute. Although it would be convenient for users, integration of an IP-level mechanism and an interface-level mechanism is not so simple.

4.3.7 Modifications to *if_output*

The ALTQ output queue model requires two modifications in *if_output* routines. One is classifier support and the other is to convert the enqueue operation.

Classifier

Figure 4.14 illustrates how a classifier is supported in *if_output()*. *struct pktattr* is used to store the classification result, and passed to *IFQ_ENQUEUE()*. However, *struct pktattr* is an opaque object to *if_output()*, so that *if_output()* does not use the contents of *struct pktattr*.

```
int
ether_output(ifp, m0, dst, rt0)
{
    ......
    struct pktattr pktattr;

    ......

    /* classify the packet before prepending link-headers */
    IFQ_CLASSIFY(&ifp->if_snd, m, dst->sa_family, &pktattr);

    /* prepend link-level headers */
    ......

    IFQ_ENQUEUE(&ifp->if_snd, m, &pktattr, error);

    ......
}
```

Figure 4.14: classifier support in *if_output*

Enqueue Operation

The second modification to *if_output* routines is to support the new enqueue operation as shown in Figure 4.15. Because enqueue and drop are combined in the new enqueue semantics, the code becomes simpler.

Figure 4.15 compares the old-style code with the new-style code. *IFQ_ENQUEUE()* enqueues an arriving packet. *IFQ_ENQUEUE()* sets an error if the packet is dropped. The *mbuf* is freed in either case; by the driver when it succeeds, and by the discipline otherwise. The caller must not touch the *mbuf* after *IFQ_ENQUEUE()* is called so that the flags and the packet length are saved before calling *IFQ_ENQUEUE()*.

```
          ##old-style##                            ##new-style##
                                        |
int                                     | int
ether_output(ifp, m0, dst, rt0)         | ether_output(ifp, m0, dst, rt0)
{                                       | {
    ......                              |     ......
                                        |
    s = splimp();                       |     mflags = m->m_flags;
    if (IF_QFULL(&ifp->if_snd)) {       |     len = m->m_pkthdr.len;
        IF_DROP(&ifp->if_snd);          |     s = splimp();
        splx(s);                        |     IFQ_ENQUEUE(&ifp->if_snd, m,
        m_freem(m);                     |                 &pktattr, error);
        return (ENOBUFS);               |     if (error != 0) {
    }                                   |         splx(s);
    IF_ENQUEUE(&ifp->if_snd, m);        |         retuen (error);
                                        |     }
    ifp->if_obytes += m->m_pkthdr.len;|     ifp->if_obytes += len;
    if (m->m_flags & M_MCAST)           |     if (mflags & M_MCAST)
        ifp->if_omcasts++;              |         ifp->if_omcasts++;
    if ((ifp->if_flags                  |     if ((ifp->if_flags
         & IFF_OACTIVE) == 0)           |          & IFF_OACTIVE) == 0)
        (*ifp->if_start)(ifp);          |         (*ifp->if_start)(ifp);
    splx(s);                            |     splx(s);
    return (error);                     |     return (error);
}                                       | }
```

Figure 4.15: change to *if_output*

4.3.8 Modifications to Drivers

The following modifications are required to drivers in order to take advantage of the new output queue model. Unmodified drivers still works but only with the original FIFO queue.

The dequeue operation is the only one required in every driver. Other changes are required only when a driver has direct references to fields inside *struct ifqueue*.

dequeue operation

The traditional *IF_DEQUEUE()* macro should be replaced by *IFQ_DEQUEUE()* in the new model as shown in Figure 4.16. Some driver skips checking whether the dequeued mbuf is NULL when it knows the queue is not empty. However, in the new model, it is necessary to check NULL even when the queue is not empty since the queue could be non-work conserving.

```
          ##old-style##                             ##new-style##
                                      |
IF_DEQUEUE(&ifp->if_snd, m);          | IFQ_DEQUEUE(&ifp->if_snd, m);
                                      | if (m == NULL)
                                      |     return;
                                      |
```

Figure 4.16: dequeue operation in driver

empty check

If the driver checks *ifq_head* to see whether the queue is empty or not, *IFQ_IS_EMPTY()* should be used as shown in Figure 4.17.

```
          ##old-style##                             ##new-style##
                                      |
if (ifp->if_snd.ifq_head != NULL)     | if (!IFQ_IS_EMPTY(&ifp->if_snd))
                                      |
```

Figure 4.17: empty check in driver

Although *IFQ_POLL()* can be used for the same purpose, it could be costly for a complex scheduling algorithm since the *IFQ_POLL()* needs to run the scheduling algorithm to select the next packet. On the other hand, *IFQ_EMPTY()* checks only if there is any packet stored in the queue. It does not mean that a packet can be dequeued because a discipline could be non-work conserving.

poll-and-dequeue

If the driver polls the packet at the top of the queue and use it before dequeueing, *IFQ_POLL()* and *IFQ_DEQUEUE()* should be used as shown in Figure 4.18. It is guaranteed that *IFQ_DEQUEUE()* immediately after *IFQ_POLL()* returns the same packet.

```
            ##old-style##                                   ##new-style##
                                       |
 m = ifp->if_snd.ifq_head;             | IFQ_POLL(&ifp->if_snd, m);
 if (m != NULL) {                      | if (m != NULL) {
                                       |
     /* use m to get resources */      |     /* use m to get resources */
     if (something goes wrong)         |     if (something goes wrong)
         return;                       |         return;
                                       |
     IF_DEQUEUE(&ifp->if_snd, m);      |     IFQ_DEQUEUE(&ifp->if_snd, m);
                                       |
     /* kick the hardware */           |     /* kick the hardware */
 }                                     | }
                                       |
```

Figure 4.18: poll-and-dequeue in driver

purge operation

IFQ_PURGE() should be used to empty the queue because a non-work conserving queue cannot be emptied by a dequeue loop.

```
            ##old-style##                                   ##new-style##
                                       |
 while (ifp->if_snd.ifq_head != NULL) {|  IFQ_PURGE(&ifp->if_snd);
     IF_DEQUEUE(&ifp->if_snd, m);      |
     m_freem(m);                       |
 }                                     |
                                       |
```

Figure 4.19: purge operation in driver

eliminating prepend operations

If the driver uses *IF_PREPEND()*, it should be eliminated. A common use of *IF_PREPEND()* is to cancel the previous dequeue operation. However, there are queueing disciplines which cannot cancel the dequeue operation once the internal state is updated. Hence, the prepend operation must not used in the new model. The logic of the code should be converted to use the poll-and-dequeue method as shown in Figure 4.20.

```
            ##old-style##                           ##new-style##
                                        |
IF_DEQUEUE(&ifp->if_snd, m);            | IFQ_POLL(&ifp->if_snd, m);
if (m != NULL) {                        | if (m != NULL) {
                                        |
    if (something_goes_wrong) {         |     if (something_goes_wrong) {
        IF_PREPEND(&ifp->if_snd, m);    |
        return;                         |         return;
    }                                   |     }
                                        |
                                        |     /* at this point, the driver
                                        |      * is committed to send this
                                        |      * packet.
                                        |      */
                                        |     IFQ_DEQUEUE(&ifp->if_snd, m);
                                        |
    /* kick the hardware */             |     /* kick the hardware */
}                                       | }
                                        |
```

Figure 4.20: eliminating prepend operations in driver

attach routine

Once the driver is converted to the new style, add *IFQ_SET_READY()* in the attach routine. It sets a flag bit to indicate that this driver is already converted to the new style. ALTQ checks this flag bit to distinguish new-style drivers.

```
            ##old-style##                           ##new-style##
                                        |
                                        | IFQ_SET_READY(&ifp->if_snd);
if_attach(ifp);                         | if_attach(ifp);
                                        |
```

Figure 4.21: set a flag bit in driver

other macros

The following macros can be used to eliminate the rest of minor direct references to the fields inside *struct ifaltq*. *IF_DROP()* is renamed just for consistency.

```
           ##old-style##                                ##new-style##
                                      |
ifp->if_snd.ifq_maxlen = qsize;       | IFQ_SET_MAXLEN(&ifp->if_snd, qsize);
                                      |
IF_DROP(&ifp->if_snd);                | IFQ_INC_DROPS(&ifp->if_snd);
                                      |
ifp->if_snd.ifq_len++;                | IFQ_INC_LEN(&ifp->if_snd);
                                      |
ifp->if_snd.ifq_len--;                | IFQ_INC_LEN(&ifp->if_snd);
                                      |
```

Figure 4.22: other macros that can be used in driver

We have already converted more than 30 drivers to be conformant with the new output queue model. Most drivers require simple replacement of a few lines from the old-style to the new-style. But there are some drivers which are hard to convert.

For example, one driver tries to prepend a copy of the previously dequeued packet. Since a copy is to be prepended, the logic of the code cannot be converted easily to use the poll-and-dequeue approach. Still, such drivers are rare, and it should not be written in such a way in the first place. These drivers are too complicated to maintain and, in fact, not well-maintained.

The rules enforced by the new output queue model encourage simpler driver design, which will reduce work required to maintain drivers and be beneficial to the community.

4.3.9 Other Considerations

The queue operations described in this section are designed for the existing network drivers in order to make them capable of QoS with minimal changes. However, the output queue abstraction can be beneficial to other future extensions.

Queueing on network cards

There are network cards which have a processor and its firmware on the card. Such firmware could be customized to support functions which are traditionally implemented in the kernel. For example, TCP/UDP checksum offloading is already implemented into the Alteon Networks Tigon gigabit ethernet driver for FreeBSD [GCY99].

Therefore, it is possible in the future that a queueing discipline is implemented in a network card in order to offload packet scheduling onto the network card. With such a device, the kernel passes a packet to the device from *if_output*, and then, the device takes care of the rest of the job. Neither the outut queue nor queueing disciplines of ALTQ are necessary.

Still, the output queue abstraction is useful to accommodate such a device. ALTQ allows to customize the enqueue operation so that it can be modified to take advantage of a specific QoS feature supported by the device. The device QoS features can be managed through the ALTQ interface; for example, attaching and enabling a specific feature. It is also likely that a device can support limited functions; for example, a device could implement a packet scheduler but the kernel could be required to provide a classifier and buffer management.

Kernel thread support

A kernel thread implementation for symmetric multi-processor (SMP) would also benefit from the output queue abstraction. In the current non-thread safe systems, access to the output queue is serialized simply by disabling interrupts. Some form of locking is, however, required to take advantage of multi-threads.

As described in Section 4.1, there are a lot of places in the existing code which directly refer to fields inside *struct ifqueue*. To make them thread-safe, each of the references should be, in principle, protected by locking. Defining and enforcing the interface for accessing the output queue structure eliminates such misbehaving references and makes it easier to lock the structure.

A simple lock (e.g., mutex) can be used to lock the output queue structure. Most of the queue operations used in ALTQ do not have dependency so that it is sufficient to lock each operation. For exmaple, a thread-safe version of *IFQ_DEQUEUE()* can be written as in Figure 4.23 using the non-thread safe version shown as *_IFQ_DEQUEUE()*.

The only exception that needs locking across multiple operations is poll-and-dequeue. The poll-and-dequeue operation requires a polled packet should be dequeued by the successive dequeue operation. A possible approach is to use a recursive lock, and explicitly lock the structure during a poll-and-dequeue as shown in Figure 4.24. The number of drivers us-

```
#define IFQ_DEQUEUE(ifq, m)           \
        do {                          \
            IFQ_LOCK((ifq));          \
            _IFQ_DEQUEUE((ifq), (m)); \
            IFQ_UNLOCK((ifq));        \
        } while (0)
```

Figure 4.23: a simple lock for dequeue

ing poll-and-dequeue is limited, and they can be easily identified by use of *IFQ_POLL()* so that explicit locking for poll-and-dequeue is straightforward.

```
IFQ_LOCK(&ifp->if_snd);
IFQ_POLL(&ifp->if_snd, m);
if (m != NULL) {

    /* use m to get resources */
    if (something goes wrong) {
        IFQ_UNLOCK(&ifp->if_snd);
        return;
    }

    IFQ_DEQUEUE(&ifp->if_snd, m);

    /* kick the hardware */
}
IFQ_UNLOCK(&ifp->if_snd);
```

Figure 4.24: recursive locking for poll-and-dequeue

4.4 Traffic Conditioning Support

The traffic conditioning support provides mechanisms for DiffServ traffic conditioning and is placed in the input path [BBC+98, BSB99]. This is illustrated in Figure 4.25. The traffic conditioning block usually contains a number of traffic conditioning elements including a classifier, meters, markers and droppers.

Incoming packets are passed to the traffic conditioning block, and may be metered, marked, and could be dropped. As opposed to the output queueing, traffic conditioning actions are usually processed sequentially. Thus, the

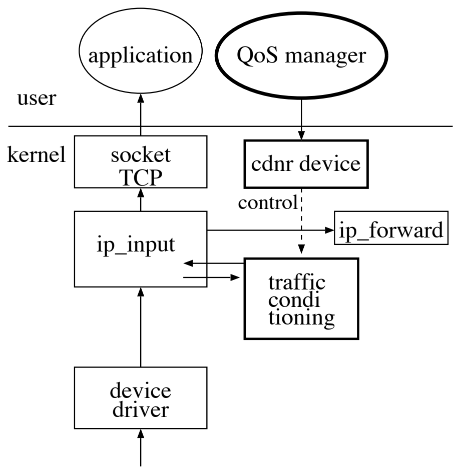


Figure 4.25: traffic conditioning support

traffic conditioning block can be supported by a simple hook.

Figure 4.26 shows a hook for traffic conditioning in *ip_input()*. *altq_input* is a pointer to a traffic conditioning function. If *altq_input* is not NULL, an incoming packet is passed to this function. Among traffic conditioning actions, the only action that affects the caller (*ip_input* in this example) is an action of dropping the packet. When the packet is dropped by traffic conditioning, the caller simply stops processing the packet.

```
void
ip_input(struct mbuf *m)
{
    ....

    if (altq_input != NULL && (*altq_input)(m, AF_INET) == 0)
        /* packet is dropped by traffic conditioner */
        return;

    ....
}
```

Figure 4.26: traffic conditioning hook in *ip_input*

Although ALTQ employs a simple hook for traffic conditioning, it is also possible to implement traffic conditioning into the input queue structure. In this case, the output queue structure of ALTQ can be applied to the protocol-dependent input queues of BSD UNIX. It can be done more easily than output queueing because input queueing in BSD UNIX does not involve with device drivers.

The advantage of this approach is that the architecture becomes more symmetric for both input and output, and a queueing discipline can be applied to the input traffic. However, the concept of traffic conditioning itself is not symmetric. Also, scheduling packets on the input side is not effective as long as the system bus is faster than the link speed. Although this assumption may not hold with gigabit ethernet, packet scheduling is not a solution for it. If the system bus of a router becomes a bottleneck, using more CPU power for packet scheduling does not solve the problem. Thus, our choice is to use a simple hook for the input path, and it is enough for traffic conditioning.

4.5 Module Management Support

The module management support provides mechanisms to control QoS mechanisms. Each QoS module corresponds to an output queueing block or a traffic conditioning block, and the framework handles a module as a unit of QoS mechanisms.

A module is controlled by *ioctl* system calls via an ALTQ device (e.g., */dev/altq/cbq*). Since each module has a different set of parameters, each module needs a different set of system calls. ALTQ is defined as a character device and each module is defined as a minor device of the ALTQ device.

To activate a module, a privileged user program opens the device associated with the module, then, attaches the module to an interface, configures it and enables it via the corresponding *ioctl* system calls. A different module can be attached to a different interface. When the module is disabled or closed, the system falls back to the original FIFO queueing.

4.5.1 Dynamic Loading and Unloading

Traditionally, the UNIX kernel is compiled and linked statically. Once the system boots, it is not possible to modify the kernel. This results in an unnecessarily large kernel which includes every possible modules and drivers, even though they are unlikely to be used.

Several modern versions of UNIX support dynamic loading of kernel modules. A module may be added or removed from a running kernel. Dynamic loading requires a runtime loader that performs relocation and binding of addresses when the module is loaded. Dynamic module support has several advantages. The system can boot with a small kernel, and loads only required modules at runtime. To upgrade a module, the kernel can unload the old module and load the new version without rebooting the entire system.

FreeBSD supports a dynamic kernel loader called KLD. In ALTQ, the kernel accesses QoS modules through a discipline switch table so that it is possible to dynamically load or unload QoS modules. The kernel boots with only the ALTQ framework built-in, and the QoS manager is responsible for dynamically loading required QoS modules before it activates QoS mechanisms in the kernel.

4.6 Summary

The ALTQ framework provides mechanisms to use alternative queueing disciplines and traffic conditioners in BSD UNIX. We have identified a number of problems of the current abstraction of output queueing, and proposed a new abstraction to support QoS. The new abstraction of output queueing is implemented into the ALTQ framework. The implementation is compatible with the existing code and allows to incrementally modify the existing drivers.

The ALTQ framework in the kernel can be further decomposed into three parts: output queueing support, traffic conditioning support, and module management support. The output queueing support provides mechanisms to support various queueing disciplines for output interfaces. The traffic conditioning support provides mechanisms to support various types of traffic conditioning for input interfaces. The module management support provides mechanisms to control QoS modules.

Chapter 5

Forwarding Mechanisms

QoS forwarding mechanisms perform actual QoS functions. QoS forwarding mechanisms are divided into two functional blocks: one is a traffic conditioning block in the input path and the other is an output queueing block on the output path. These blocks are further divided into components such as classifiers, markers and queueing disciplines.

The core component of a QoS forwarding mechanism is a queueing discipline that is the point of actual service differentiation. Our assumption of the system design is that forwarding performance of the system is much faster than the link speed so that packet scheduling is effective only at output queueing. Thus, a series of QoS processing can be sequentially applied to a packet in the traffic conditioning block. On the other hand, enqueue processing and dequeue processing are asynchronous in the output queueing block. As a result, the traffic conditioning block can be modeled as a sequential procedure but the output queueing block should be modeled as an event-driven state-machine.

In this chapter, we review the implemented forwarding mechanisms. The details of the mechanisms, simulation results, and analysis can be found elsewhere [Nag87, DKS89, Kes91, McK90, FJ95, WGC$^+$95, FJ93, Flo94, Cho99, SZ97, CF98].

5.1 FIFOQ (First-In First-Out Queueing)

FIFOQ is a simple tail-drop FIFO queue. FIFOQ is the simplest possible implementation of a queueing discipline in ALTQ, and can be used to compare with other queueing disciplines. FIFOQ can be also used as a template for those who want to write their own queueing disciplines.

Implementing a New Discipline

We assume that a queueing discipline is evaluated by simulation, and then ported onto ALTQ. The NS simulator [SS95] is one of a few simulators that support different queueing disciplines. The NS simulator is widely used in the research community and includes the RED and CBQ modules.

To implement a new queueing discipline in ALTQ, one can concentrate on the enqueue and dequeue routines of the new discipline. The FIFOQ implementation is provided as a template so that the FIFOQ code can be modified to put a new queueing discipline into the ALTQ framework. The basic steps are just to add an entry to the ALTQ device table, and then provide open, close, and ioctl routines. The required *ioctls* are attach, detach, enable, and disable. Once the above steps are finished, the new discipline is available on all the interface cards supported by ALTQ.

To use the added discipline, a privileged user program is required. Again, a daemon program for FIFOQ included in the release should serve as a template.

5.2 WFQ (Weighted-Fair Queueing)

WFQ [Nag87, DKS89, Kes91] is the best known and the best studied queueing discipline. In a broad sense, WFQ is a discipline that assigns a queue for each flow. A weight can be assigned to each queue to give a different proportion of the network capacity. As a result, WFQ can provide protection against other flows. In the queueing research community, WFQ is more precisely defined as the specific scheduling mechanism proposed by Demers et al. [DKS89] that is proved to be able to provide worst-case end-to-end delay bounds [Par92]. Our implementation is not WFQ in this sense, but is closer to a variant of WFQ, known as SFQ or stochastic fairness queueing [McK90]. A hash function is used to map a flow to one of a set of queues, and thus, it is possible for two different flows to be mapped into the same queue. In contrast to WFQ, no guarantee can be provided by SFQ.

5.2.1 Implementation

Our WFQ implementation allocates 256 queues per interface by default. An incoming packet is classified into one of the queues by a hash function. 3 hash functions are provided. The default hash function uses the destination IP address as a hash input so that packets heading to the same destination belong to the same queue. Another hash function uses 5 tuples (source

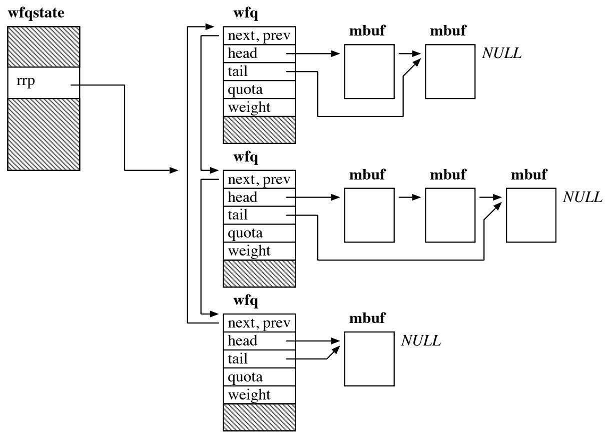


Figure 5.1: WFQ scheduler

address, destination address, source port, destination port, protocol). The last hash function is for debugging and uses only the source port.

Figure 5.1 shows the WFQ data structure. The *wfqstate* structure contains interface information, and the *wfq* structure contains queue information. In BSD UNIX, the *mbuf* structure is used to hold a packet. Backlogged packets are kept in a mbuf-chain. When a queue becomes backlogged, *wfq* is placed in a circular list that holds all backlogged queues for the interface. The *rrp* field of *wfqstat* is a round-robin pointer to schedule a queue among the backlogged queues.

The weighted-round robin scheduler adds a quota to a queue at every round. A queue is allowed to send a packet if its quota is positive. When the scheduler dequeues a packet from the queue, the length of the packet is subtracted from the quota. The weight of a queue determines the amount of the quota added to the queue at each round. When the number of backlogged packets exceeds the limit, a packet is discarded from the head of the longest queue.

5.3 CBQ (Class-Based Queueing)

CBQ was proposed by Jacobson and has been studied by Floyd [FJ95]. CBQ has given careful consideration to implementation issues, and is implemented as a STREAMS module by Sun, UCL and LBNL [WGC$^+$95]. Our CBQ code is ported from CBQ version 2.0 and enhanced.

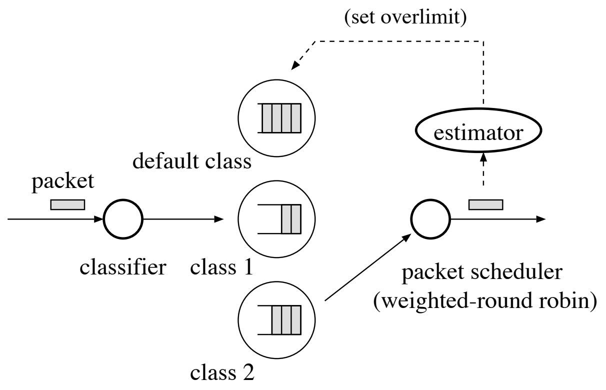


Figure 5.2: CBQ components

CBQ achieves both partitioning and sharing of link bandwidth by hierarchically structured classes. Each class has its own queue and is assigned its share of bandwidth. A child class can borrow bandwidth from its parent class as long as excess bandwidth is available.

Figure 5.2 shows the basic components of CBQ. CBQ works as follows: The classifier assigns arriving packets to the appropriate class. The estimator estimates the bandwidth recently used by a class. If a class has exceeded its predefined limit, the estimator marks the class as overlimit. The scheduler determines the next packet to be sent from the various classes, based on priorities and states of the classes. Weighted-round robin scheduling is used between classes with the same priority.

CBQ Estimator

The CBQ estimator measures the bandwidth use of a class at every packet departure by computing the disparity between the target packet interval and the measured packet interval. If the router sends packets of size s from the class at precisely the link-sharing bandwidth b allocated to the class, then, the interdeparture time between successive packets would be

$$f(s,b) = s/b \tag{5.1}$$

When the "measured" interdeparture time is t, the discrepancy between the actual interdeparture time and the "allocated" interdeparture time is

$$\mathit{diff} = t - f(s,b) \tag{5.2}$$

The estimator computes *avg*, the exponentially-weighted moving average (EWMA) of *diff*, by

$$avg \leftarrow (1-w)avgidle + w * diff \tag{5.3}$$

Here, w is a weight of the EWMA. *avg* increases when the class uses more than its allocated bandwidth, and decreases when less than its allocated bandwidth. The estimator judges a class as "overlimit" when its *avg* becomes negative.

The parameter *maxidle* gives an upper bound for *avgidle*, and controls the burstiness allowed to a class. To permit a maximum burst of *maxburst* back-to-back packets, *maxidle* is set as follows:

$$maxilde \leftarrow t(1/p-1)\frac{1-g^{maxburst}}{g^{maxburst}} \tag{5.4}$$

for t the interpacket time for average sized packets sent back-to-back, p the fraction of the link bandwidth allocated to the class, and weight g.

The parameter *offtime* gives the time interval that an overlimit class must wait before sending another packet. *offtime* is determined by the steady-state burst size *minburst* for a class when the class is running over its limit. For a burst size of 1. *offtime* is set as follows:

$$offtime \leftarrow t(1/p-1) \tag{5.5}$$

For a steady-state burst size of $minburst + 1$ packets for $minburst \geq 1$, *offtime* is further modified as follows:

$$offtime \leftarrow offtime(1 + \frac{1}{1-g}\frac{1-g^{minburst}}{g^{minburst}}) \tag{5.6}$$

5.3.1 Implementation

Class Management

CBQ maintains a class hierarchy for an interface. Figure 5.3 illustrates how CBQ manages classes. The *rm_ifdat* structure contains interface information, and the *rm_class* structure contains class information. The *root* field of the *rm_ifdat* structure points to the root class of the class hierarchy. The *rm_class* structure has three fields to maintain the class hierarchy. *parent* points to its parent class. Only the root class has no parent and this field is set to *NULL*. *children* points to the first child in the child list. *next* points to the next class in the child list. The last class in the child list has the *next* field set to *NULL*.

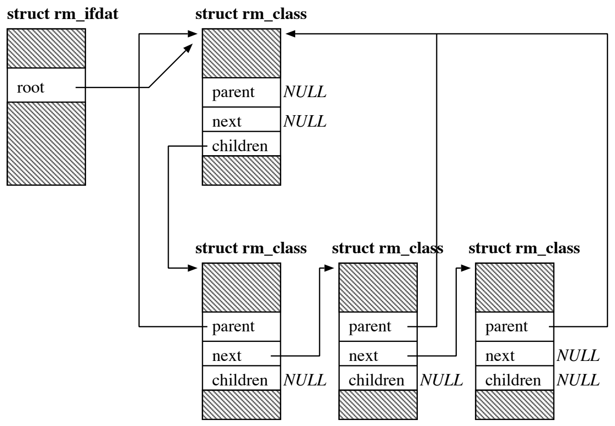


Figure 5.3: CBQ class hierarchy

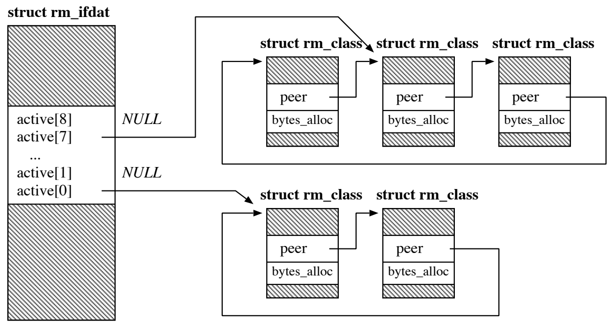


Figure 5.4: CBQ weighted-round robin scheduler

Scheduler

Figure 5.4 shows the CBQ scheduler that combines priority, weighted-round robin (WRR) and borrowing. CBQ has 8 priorities, and each class is assigned a fixed priority. When a class is backlogged, the class becomes active and is placed into a circular list of the corresponding priority. Active classes with the same priority are chained in the circular list, and the *peer* field in *rm_class* points to the next class in the circular list. When the queue of a class becomes empty, the class is removed from the circular list. The *active* field of rm_ifdat is an array which points to the next class to be scheduled for each priority.

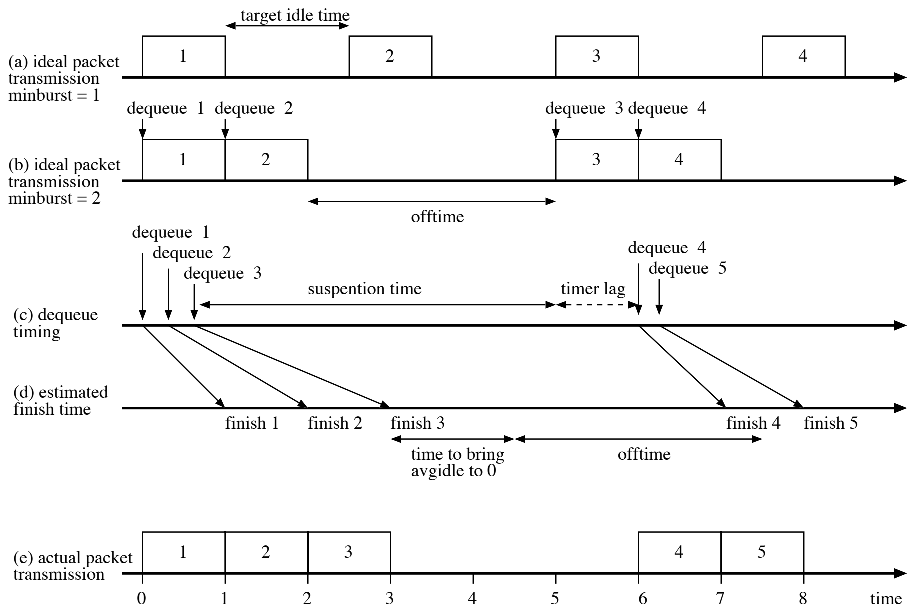

target idle time
(a) ideal packet transmission minburst = 1
1
2
3
4
dequeue 1
dequeue 2
dequeue 3
dequeue 4
(b) ideal packet transmission minburst = 2
offtime
dequeue 5
suspention time
timer lag
(c) dequeue timing
(d) estimated finish time
finish 1
finish 2
finish 3
finish 4
finish 5
time to bring avgidle to 0
(e) actual packet transmission
0
5
6
7
8
time

Figure 5.5: calculation of suspension time of a class

To dequeue a packet, the CBQ scheduler goes through the active lists from the highest priority to the lowest priority. For each priority, the scheduler employs WRR. A class can send a packet only when it is underlimit and has tokens larger than the next packet. The *bytes_alloc* field in rm_class holds tokens, and new tokens are added every time the scheduler visits the class. The amount of tokens given at a time is precomputed when a class is created. If the scheduler finds a class which can send a packet, the scheduler selects the class. If no class can send a packet within a priority, the scheduler goes to the next priority. If no class is found even for the lowest priority, the scheduler selects the first borrowable class it encounters during the search.

Estimator

Figure 5.5 shows how a class is regulated in our CBQ implementation. (a) shows ideal steady-state packet transmission with burst size 1 at the assigned rate. The target idle time of the class is the interval of this ideal transmission pattern. The average idle time is computed from the disparity between the target idle time and the measured idle time. Thus, the average idle time decreases when the class transmits more than its assigned rate, and increases when the class transmits less than its assigned rate.

(b) shows the effect of *minburst* that allows a bust size of n. In this example, n is 2. *offtime* is calculated by Equation (5.6) and used to suspend the class when it becomes overlimit. In a steady-state, the average idle time becomes 0 after sending 2 back-to-back packets, and the class is suspended for *offtime*. When the class is resumed, the average idle time becomes a positive value enough to send another 2 back-to-back packets.

The original CBQ design assumes that a back-logged packet is dequeued from a transmission complete interrupt for the previous packet. It also implies that CBQ can obtain the current time at both start and end of each packet transmission.

With modern network cards, however, CBQ cannot be notified at each transmission complete. Modern network cards have large transmission buffer or DMA descriptors to hold a number of packets. Many cards post interrupts only when the buffer becomes empty.

Thus, in our implementation, packet transmission complete time is estimated by software. When a packet is dequeued, its finish time is estimated from the link speed. The skew between the dequeue time and the estimated finish time reflects buffered packets in the network card, and is illustrated in (c) and (d).

When CBQ dequeues Packet 1, it sets up the network card for transmission, and then, computes the finish time as well as the average idle time of the class. If the average idle time is positive, CBQ can immediately send another packet from the same class. Assume that the average idle time becomes negative after the third packet. CBQ suspends the class at this point. The suspension time is the time required to bring the average idle time to 0 plus *offtime* in order to allow the class to send another 2 packets after resume.

There are two sources which affect the precision of the suspension time. One is the skew between the dequeue time and the estimated finish time. The skew changes by traffic from other classes and cannot be correctly estimated at the suspension time. The other is the timer lag. CBQ uses the kernel timer to resume a suspended class but the kernel timer has a limited granularity (10msec by default). Resume timing is always delayed to the next timer interval. Because of these limitations, actual suspension time is not accurate but the error is reflected to the average idle time and compensated later. It allows to respect the sending rate of a class in a longer term, and the mechanism is flexible and robust against other factors.

Timer Granularity

Timers are frequently used to set timeout-process routines. While timers in a simulator have almost infinite precision, timers inside the kernel are implemented by an interval timer and have limited granularity. Timer granularity and packet size are fundamental factors to packet scheduling.

To take one example, the accuracy of the bandwidth control in CBQ relies on timer granularity in the following way: CBQ measures the recent bandwidth use of each class by averaging packet intervals. CBQ regulates a class by suspending the class when the class exceeds its limit. To resume a suspended class, CBQ needs a trigger, either a timer event or a packet input/output event. In the worst case scenario where there is no packet event, resume timing is rounded up to the timer granularity. Most UNIX systems use 10 msec timer granularity as default, and CBQ uses 20 msec as the minimum timer.

Each class has a variable *maxburst* and can send at most *maxburst* back-to-back packets. If a class sends *maxburst* back-to-back packets at the beginning of a 20 msec cycle, the class gets suspended and would not be resumed until the next timer event—unless other event triggers occur. If this situation continues, the transfer rate becomes

$$rate = packetsize \times maxburst \times 8 \div 0.02$$

Now, assume that *maxburst* is 16 (default) and the packet size is the link MTU. For 10baseT with a 1500-byte MTU, the calculated rate is 9.6Mbps. For ATM with a 9180-byte MTU, the calculated rate is 58.8Mbps.

A problem arises with 100baseTX; it is 10 times faster than 10baseT, but the calculated rate remains the same as 10baseT. CBQ can fill only 1/10 of the link bandwidth. This is a generic problem in high-speed network when packet size is small compared to the available bandwidth. Because increasing *maxburst* or the packet size by a factor of 10 is problematic, a fine-grained kernel timer is required to handle 100baseTX. Current PCs seem to have little overhead even if timer granularity is increased by a factor of 10. The problem with 100baseTX and the effect of a fine-grained timer are illustrated in Section 7.3.

Depending solely on the kernel timer is, however, the worst case. In more realistic settings, there are other flows or TCP ACKs that can trigger CBQ to calibrate sending rates of classes.

Heuristic Algorithms

Queueing algorithms often employ heuristic algorithms to approximate the ideal model for efficient implementation. But sometimes properties of these heuristics are not well studied. As a result, it becomes difficult to verify the algorithm after it is ported into the kernel.

The *Top-Level link-sharing* algorithm of CBQ suggested by Floyd [FJ95] is an example of such an algorithm. The algorithm employs heuristics to control how far the scheduler needs to traverse the class tree. The suggested heuristics work fine with their simulation settings, but do not work so well under some conditions. It requires time-consuming efforts to tune parameters by heuristics. Although good heuristics are important for efficient implementation, heuristics should be carefully used and study of properties of the employed heuristics will be a great help for implementors.

5.4 RED (Random Early Detection)

RED was also introduced by Floyd and Jacobson [FJ93]. RED is an implicit congestion notification mechanism that exercises packet dropping or packet marking stochastically according to the average queue length. Since RED does not require per-flow state, it is considered scalable and suitable for backbone routers. At the same time, RED can be viewed as a buffer management mechanism and can be integrated into other packet scheduling schemes.

RED computes the average queue size as follows:

$$avg \leftarrow (1 - w_q)avgidle + w_q q \tag{5.7}$$

The weight w_q determines the time constant of the low-pass filter that allows transient changes of the instantaneous queue size.

avg is compared to two thresholds, a minimum threshold min_{th} and a maximum threshold max_{th}. When $avg < min_{th}$, no packet is dropped. When $avg \geq max_{th}$, all packets are dropped. When $min_{th} \leq avg < max_{th}$, packets are dropped with the probability that is a linear function of avg as shown in Figure 5.6. As avg varies from min_{th} to max_{th}, the packet-dropping probability p_b varies linearly from 0 to max_p.

$$p_b \leftarrow max_p(avg - min_{th})/(max_{th} - min_{th}) \tag{5.8}$$

Floyd also suggests the modified probability p_a that increases slowly as the count increases since the last dropped packet [FJ93]:

$$p_a \leftarrow p_b/(1 - count \times p_b) \tag{5.9}$$

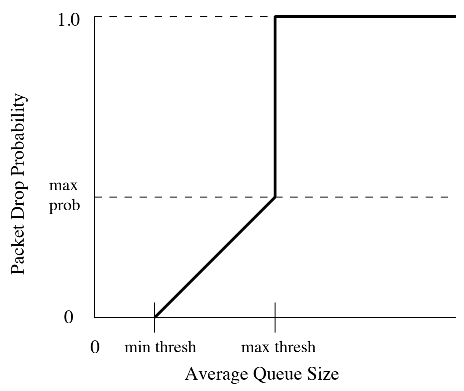


Figure 5.6: RED packet drop probability as a function of the average queue size

5.4.1 ECN (Explicit Congestion Notification)

Explicit Congestion Notification (ECN) [Flo94] is a congestion signaling mechanism under standardization process. The RED algorithm is used to select packets to mark so that ECN is experimentally supported in ALTQ as an extension of RED.

In the current Internet, packets are silently discarded under heavy congestion. TCP takes a packet loss as a congestion signal but TCP cannot distinguish packet loss by congestion from packet loss by corruption. By explicitly signaling a congestion notification, ECN provides performance improvement over a packet drop scheme, especially when packet loss leads to TCP timeout.

The ECN allows the sender to reduce the sending rate without losing a packet. The ECN mechanism is divided into the router mechanism and the end host mechanism.

The ECN proposal uses a 2-bit scheme in the IPv4 TOS field or in the IPv6 Traffic Class field. The ECT bit is set by the sender to indicate that it is an ECN-Capable Transport system. The CE bit is set by a router to indicate Congestion Experienced. An router runs the RED algorithm to select a packet and, if the ECT bit is set, the router marks the CE bit instead of discarding the packet.

The ECN proposal also uses 2 bits in the TCP flag field, the ECN-Echo bit and the CWR (Congestion Window Reduced) bit. At an initial TCP handshake, each host negotiates with its peer as to whether it will use

ECN. If both hosts agree to use ECN, the sender sets the ECT bit in all sending packets. When the receiver receives a packet with the CE bit set, the receiver sets the ECN-Echo bit in successive packets sent back to the sender. When the sender receives a packet with the ECN-Echo bit set, the sender reduces its congestion window using the Fast-Recovery algorithm. The sender reacts to ECN-Echo only once in the RTT interval to avoid over-reacting to ECN-Echo packets. After reducing the congestion window, the sender sets the CWR bit in the next packet to indicate it has adjusted the congestion window. The receiver stops setting the ECN-Echo bit when it receives a packet with the CWR bit set.

5.4.2 Implementation

Our implementation of RED is derived from the RED module in the NS simulator version 2.0 [SS95]. The average queue size is kept as a 32-bit fixed-point value because floating-point operations are not available in the kernel.

RED and ECN in ALTQ are integrated into CBQ and H-FSC so that RED and ECN can be enabled on a class queue basis.

One implementation issue of ECN is that IPv4 has a checksum in the header so that the checksum needs to be updated when the congestion experienced bit is modified. The algorithm to efficiently update the checksum is described in [Rij94]. On the other hand, IPv6 does not have a checksum assuming that the underlying link layer supports some form of error detection.

Precision of Integer Calculation

32-bit integer calculations easily overflow or underflow with link bandwidth varying from 9600bps modems to 1Gbps Gigabit Ethernet. In simulators, 64-bit double precision floating-point is available and it is reasonable to use it to avoid precision errors. However, floating-point calculation is not available or not very efficient in the kernel since the floating-point registers are not saved for the kernel (in order to reduce overhead). Hence, algorithms often need to be converted to use integers or fixed-point values. Our RED implementation uses fixed-point calculations converted from floating-point calculations in the NS simulator. We recommend performing calculations in the user space using floating-point values, and then bringing the results into the kernel. CBQ uses this technique. The situation will be improved when 64-bit integers become more commonly used and efficient.

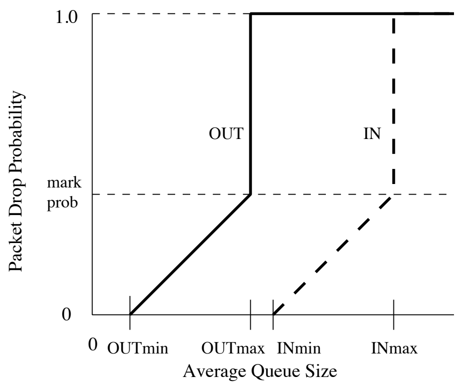


Figure 5.7: RIO

5.5 RIO (RED with IN and OUT)

Clark *et al.* proposed RIO [CF98] in order to discriminate against out-of-profile packets in times of congestion. RIO has 2 sets of RED parameters; one for in-profile packets and the other for out-of-profile packets. At the ingress of the network, profile meters tag packets as *in* or *out* based on contracted profiles for customers. Inside the network, *in* packets receive preferential treatment by the RIO dropper.

The RIO mechanism is a dual RED scheme; one for in packets and the other for *out* packets. Figure 5.7 shows the packet drop probability for *in* and *out* packets. As the average queue size grows, *out* packets are dropped first. If congestion persists even after dropping all *out* packets, *in* packets start experiencing packet loss.

It is possible to provision the network not to drop *in* packets at all by providing enough capacity for the total volume of *in* packets. Thus, RIO can be used to provide a service that statistically assures capacity allocated for users.

The original RIO has 2 different drop precedence values: *in* and *out*. However, the mechanism can be extended to support an arbitrary number of drop precedence levels. 3 drop precedence levels are defined for the Assured Forwarding of DiffServ [HBWW99]. Since adaptive flows are likely to stay under the medium drop precedence level under congestion, the medium drop precedence would protect adaptive flows from unadaptive flows.

The RIO implementation in ALTQ supports 3 drop precedence levels required to support Assured Forwarding.

5.6 Flow-valve

We have proposed the flow-valve, a safety-valve mechanism for RED to protect the network from misbehaving or overpumping flows and to promote end-to-end congestion control [Cho99]. The flow-valve can be regarded as an implementation of the concept known as a "RED penalty-box" but our focus is to protect router resources in times of congestion.

The flow-valve detects a traffic increase that goes beyond the control range of RED, and protect the local resources by forcing overpumping flows to back off. The flow-valve provides an incentive for end-to-end congestion control to keep the packet drop rate low under moderate congestion, and to conservatively back off under heavy congestion. Our simulation results demonstrate that the flow-valve can effectively protect the network from misbehaving flows and, at the same time, isolate undesirable behavior of conformant TCP.

5.6.1 Flow-valve mechanism

The flow-valve is a mechanism that detects a traffic increase that goes beyond the control range of RED, and cuts off the flow causing the overload to protect responsive flows and router resources. RED falls back to the simple Drop Tail behavior when the average queue length exceeds max_{th}, which means the traffic is getting out of control. It is likely that the traffic increase is caused by a flow not cooperating with others, and blocking the uncooperative flow will bring the queue length back in the proper range.

The flow-valve borrows many ideas from [FF99] but differs in that our focus is to engineer RED to work in the proper range even in the face of misbehaving flows. Our engineering challenge is to design a mechanism that works with a small number of samples or a transient condition, and approaches the theoretical model as the sample number increases or as the flow state becomes steady.

The flow-valve presents a model similar to the penalty-box but in a different light. That is, our model is a "safety-valve" instead of a "penalty-box", designed for easy protection and management of network resources at routers by employing two simple policies.

The first policy is to detect an "overpumping" flow instead of a "misbehaving" flow. "Overpumping" means that the sender is transmitting packets more than it should be as perceived by a router along the path. To identify misbehavior, a router needs evidence of misbehavior, which is the source of the difficulties in the penalty-box model. However, identifying overpumping is a decision local to the router experiencing congestion, and the router does not need to prove misbehavior on the sender side. A simple test, *the overpumping test*, is performed locally at each router to detect overpumping flows.

The second policy is to simply block a flow judged as overpumping at a router until the sender backs off exponentially. A simple test, *the backoff test*, is performed to observe exponential backoff.

One might think that blocking a flow is too simplistic and too damaging. However, it turns out that the penalty of a short blocking period is not so different from a passive measurement approach for most TCP implementations. Given the high packet drop rate of an overpumping flow, the probability of timeouts is already quite high without forcing it. The throughput of a TCP session is already severely damaged when the packet drop rate becomes this level.

In addition, a simple blocking scheme has several advantages over measuring rate reduction. The first advantage is quick reaction to traffic surge. Because our goal is to protect the network, the mechanism should respond without delay to offensive flows and a certain-to-work mechanism is needed to protect routers. A statistical approach is not suitable to this end.

The second advantage is that it is more effective in dissolving congestion. If the traffic load reaches the level that RED is no longer able to control, the congestion should be quickly dissolved.

The third advantage is bounded penalty. If a stochastic penalty were used, an unfortunate flow could be punished repeatedly. An exponential backoff mechanism is deterministic and can be observed in a fixed time period.

The fourth advantage is the emphasis on the backoff behavior. We believe that both timeouts and exponential backoff are essential to best effort traffic in order to avoid congestion collapse, though the importance of timeouts and exponential backoff has not been addressed much in previous research. There are many proposals for TCP to avoid timeouts and improve performance but those approaches do not help reduce the packet drop rate at a busy bottleneck link. Traditional TCP implementations are conservative in backing off at a high packet drop rate, which lowers the risk of congestion collapse. In some sense, more aggressive TCP implementations exemplify the lack of an incentive to reduce the packet drop rate.

The fifth advantage is the fairness of the penalty for conservative implementations. The flow-valve, by blocking the arriving packets and observing the backoff behavior, does not allow an aggressive retransmission policy to perform better than others. The only way to not be judged as overpumping is to keep the flow's packet drop rate low. Once judged as overpumping, the penalty is equal for all.

The flow-valve provides an incentive for congestion control; a user needs to keep the packet drop rate low under moderate congestion and to conservatively back off under heavy congestion.

In spite of the differences in our approach, the resulting mechanism is not so different from the original penalty-box model. The flow-valve can be regarded as an implementation of the penalty-box model.

Overpumping Test

The overpumping test is used to detect a flow causing overload. When the traffic is under the control of RED, the average queue length stays between min_{th} and max_{th} and the packet drop rate stays less than max_p. RED stochastically drops packets according to the traffic load and responsive flows control their sending rates in response to the packet loss. In such a dynamic traffic environment, a flow that adapts better to the network condition is likely to have a lower packet drop rate than a flow that adapts less because adaptive flows back off during congestion. RED is designed to keep the packet drop rate under max_p with cooperative TCP flows. Thus, if a flow's drop rate p_{avg} exceeds max_p, it is an indication that the flow is not adapting well or not adapting at all. Therefore, we set the packet drop rate threshold p_{th} to max_p and detects flows with $p_{avg} > p_{th}$. It is clear that, if we simply block flows whose packet drop rate is more than max_p, the RED packet drop rate never exceeds max_p.

However, we need to exclude flows using less than their share of the bandwidth because those flows suffer packet drops caused by other flows. Therefore, f_{avg} should be checked as a supplementary test. A fixed threshold could be used to check f_{avg} but a simple function of p_{avg} is used to calculate a reasonable threshold for the packet arrival rate. This function $f_{th}(p)$ is developed later based on a rough approximation of the TCP-friendly model. For now, suppose there is a reasonable function $f_{th}(p)$. Then, we can judge a flow to be overpumping when

$$(p_{avg} > p_{th})\ \ AND\ \ (f_{avg} > f_{th}(p_{avg})) \tag{5.10}$$

Note that the overpumping test is to detect the flow causing the overload, and thus, misbehaving flows could stay undetected since the flow-valve is triggered only when p_{avg} exceeds p_{th}. On the other hand, p_{avg} of a responsive flow could exceeds p_{th}. There are a number of possible reasons for that. For example, a large window size and a large RTT could lead to a burst of packet

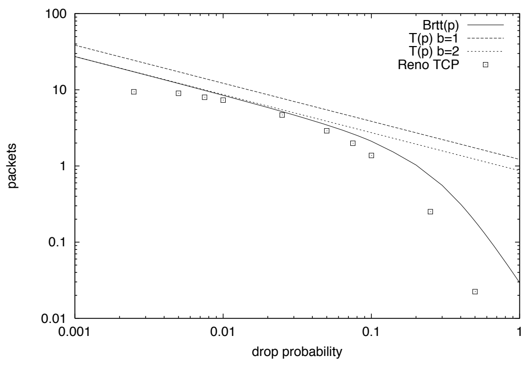


Figure 5.8: TCP throughput in packets per RTT

drops. It is also legitimate for TCP to sustain the sending rate at a high packet drop rate when packet drops are fairly uniformly distributed.

In some sense, the flow-valve compensates for the unfairness caused by the TCP mechanism. It is well known that a smaller round-trip time has a clear advantage over a larger one, and thus, a TCP flow with a small round-trip time could be very greedy. Another example is the first slow-start of a TCP session. In the first slow-start, TCP does not know the point (called *ssthresh*) to start the congestion avoidance algorithm, and often results in a burst of packet drops. The flow-valve works as a protection mechanism against such behavior of conformant TCP.

Backoff Test

The backoff test is used to free a blocked flow. A blocked flow is freed when the retransmission interval becomes more than a backoff threshold d_{th}. Since all the arriving packets are dropped, the sender will continue to double the retransmission interval until the retransmission interval reaches d_{th}. It can be easily detected by a drop timestamp t_{last}.

To observe the exponential growth in the retransmission interval, d_{th} should be an exponentially distributed random value. In practice, a fixed threshold can be used along with a coarse timestamp since rounding errors effectively provide randomization and it is not necessary to check retransmission intervals more than a few seconds.

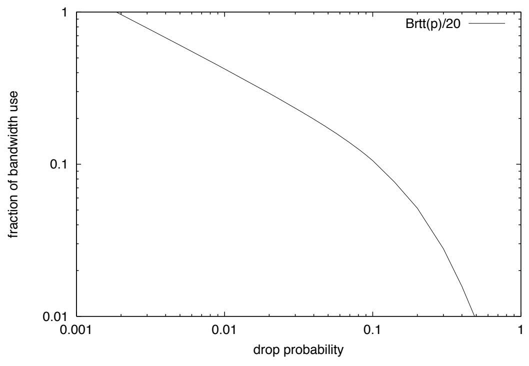


Figure 5.9: estimated bandwidth share as a function of packet drop rate

Packet Arrival Rate Function

For the overpumping test, we need a simple function of the flow's packet drop rate to estimate a reasonable bandwidth share in order to judge overpumping. We derive a rough approximation of the TCP throughput model using the knowledge of the queue state. However, our goal is to derive a simple approximation that can be used to judge overpumping and the function is not necessarily a precise model of TCP. Even if we had a precise model, it would not work with a small number of samples or transient conditions.

We use the analytical model proposed by Padhye *et al.* [PFTK98] to approximate the throughput of TCP. It is suitable for a large packet drop rate p since it assumes retransmission timeouts, exponential backoff and large RTTs. When the throughput of TCP is not limited by the maximum window size, the throughput in packets $B(p)$ is approximated by:

$$B(p) \approx \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 min(1, 3\sqrt{\frac{3bp}{8}})p(1+32p^2)} \tag{5.11}$$

The approximation assumes p is small but it is shown that the model fits well to the measurements over a wide range of p.

Equation (5.11) can be further simplified by eliminating variables other than p. TCP calculates the retransmission timeout value T_0 (also known as RTO) [Jac88, Jac90] by:

$$RTO = srtt + 4 \cdot rttvar \tag{5.12}$$

Thus, we can assume $T_0 > RTT$. We also assume $b = 2$ to reflect ack-every-other-packet policies. Then, the throughput in packets per RTT $B_{rtt}(p)$ is given by:

$$B_{rtt}(p) = B(p) \cdot RTT < \frac{1}{\sqrt{\frac{4p}{3}} + min(1, 3\sqrt{\frac{6p}{8}})p(1 + 32p^2)} \tag{5.13}$$

$B_{rtt}(p)$ shows how many packets a TCP session can transmit per RTT. $B_{rtt}(p)$ overestimates the throughput because we ignore $rttvar$ in (5.12) and the original model does not assume coarse timer granularity. $B_{rtt}(p)$ in Figure 5.8 reveals that TCP can send only two packets per RTT when $p = 0.1$. We also plot $T(p)$ derived from the original TCP-friendly test in [FF99] with $b = 1$ and $b = 2$. Since $T(p)$ does not assume timeouts, $T(p)$ differs from $Brtt(p)$ when $p > 0.01$. The numbers are simply calculated using the link latency in the simulation as RTT. The plot confirms that Equation (5.13) provides a good estimation for a wide range of p.

Next, we use $Brtt(p)$ to estimate the flow's share of the bandwidth using the knowledge of the queue state. We know that, if p is large, this router is a bottleneck for the flow. The flow's RTT includes the queueing delay at this node and the queueing delay at this node must be a significant fraction of the end-to-end delay. Let avg be the average queue length. The queueing delay at this node can be approximated by the packet service time for avg packets. A single flow is supposed to have less than $B_{rtt}(p)$ packets in the queue. Then, the flow's bandwidth share f becomes

$$f < \frac{B_{rtt}(p)}{(avg + \alpha)} \tag{5.14}$$

Equation (5.14) can be used to estimate the flow's share of the bandwidth. In Equation (5.14), α is an additional factor of the latency. In practice, it is reasonable to set several packets to α if we take into account the buffers inside the network interface cards. If the propagation delay of the attached link is known to be large, it can be added too. There are other factors that possibly contribute to RTT; queueing delay at other routers, (store-and-forward) forwarding delay at the routers, or congestion of the reverse path.

For the overpumping test, Equation (5.14) can be further simplified. When the overpumping test needs to check f_{avg}, the flow's packet drop rate p_{avg} already exceeds p_{th} and we need to check f_{avg} to exclude a flow using less than its share. If a flow is not using more than its share but the flow's packet drop rate is more than p_{th}, avg is supposed to be more than max_{th} since p_{th} is set to max_p. Therefore, f should be

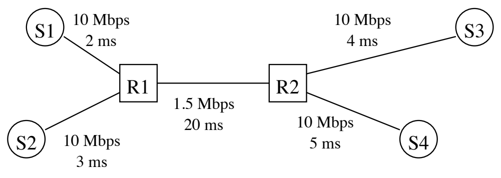


Figure 5.10: simulation network

$$f < \frac{B_{rtt}(p)}{(max_{th} + \alpha)} \tag{5.15}$$

Equation (5.15) approximates the reasonable share for a TCP-friendly flow as a simple function of p. For an efficient implementation, the function $f_{th}(p)$ could be implemented in a lookup table, or it could be a fixed threshold for $p = p_{th}$. Figure 5.9 shows Equation (5.15) with $max_{th} = 15$ and $\alpha = 5$, and it is in the range appropriate for the overpumping test; $f_{th}(0.05) = 0.17$, $f_{th}(0.1) = 0.1$, $f_{th}(0.2) = 0.05$ and $f_{th}(0.5) = 0.01$.

The function $f_{th}(p)$ is derived using several assumptions. The assumptions may not hold for some environments but the model is better than a heuristic fixed value since it can be easily verified. However, the function is supplementary to the overpumping test and errors in the estimation do not have a significant impact.

In this model, we assume all packets have the equal service time, implying that all packets are equal in size. The RED mechanism can be implemented in either the packet mode or the byte mode [FJ93] and the above model corresponds to the packet mode. In the byte mode, the average queue length counts the queue size in bytes so that it is straightforward to extended our model for the byte mode to take small packets into consideration.

5.6.2 Simulation Results

This section presents the simulation results to illustrate the behavior of the flow-valve. Two scenarios are used in the *ns* simulator (version 2.1b3) [SS95] with a simple topology in Figure 5.10. S1 and S2 are traffic sources and S3 and S4 are traffic sinks. R1 is a bottleneck router and RED and the flow-valve are enabled at R1. The RED parameters are configured with ($min_{th} = 5, max_{th} = 10, max_p = 0.1$). The queue size limit is 25. The flow-valve parameters are configured with ($p_{th} = 0.1, d_{th} = 1$). Reno TCP is used for the simulation.

In Figure 5.11 through Figure 5.13, graph (a) shows the sequence number of packets observed at R1. Sequence number n of Flow i is plotted at $((n \; mod \; 90)/100 + i)$ so that each flow corresponds to each main row and the sequence number wraps around every 90 packets. A packet is marked in gray when it dequeued, and a dropped packet is marked as 'X' in black. Graph (b) shows the bandwidth use of each flow measured at 0.25 second intervals in Figure 5.11 and 5.12, at 0.5 second intervals in Figure 5.13. The bandwidth use is normalized to the link bandwidth.

Test 1

Test 1 is a 25-second-long sequence that illustrates two typical scenarios; one is that an unresponsive flow is detected by the flow-valve during a traffic surge, and the other is that a high-bandwidth flow is quickly throttled by the flow-valve.

The following 4 flows, two ftp flows and two constant bit rate (CBR) flows, are used.

Flow 1	ftp from S1 to S3 (20 segment window size)
Flow 2	ftp from S2 to S3 (5 segment window size)
Flow 3	CBR from S2 to S4 (800Kbps, 1000B/packet)
Flow 4	CBR from S1 to S4 (1.6Mbps, 1000B/packet)

In the beginning of the scenario, two TCP flows, Flow 1 and Flow 2, share the bandwidth. The throughput of Flow 2 is limited by the small window size. At time 8, a CBR flow, Flow 3, starts up and the other two TCP flows reduce their sending rates. At time 15, another CBR flow, Flow 4 is invoked for 0.3 seconds to emulate a traffic surge. At time 20, Flow 4 starts up again, and it lasts longer this time.

Figure 5.11 shows how the original RED works in Test 1. After Flow 4 starts for the second time, the average queue length reaches max_{th} and RED falls back to the Drop Tail behavior. The TCP flows back off and the two CBR flows use up the available bandwidth.

Figure 5.12 shows the effect of the flow-valve in Test 1. When Flow 3 starts up, the estimated packet drop rate of Flow 3 jumps up, but then, gradually decreases as the other TCP flows reduce their sending rates. The flow-valve does not detect Flow 3 as overpumping at this point since the packet drop rate of Flow 3 is less than the threshold p_{th}. The RED average queue length stays between min_{th} and max_{th}, and the packet drop rate of the TCP flows stay at 2%–3%.

When Flow 4 is invoked to emulate a traffic surge, the sudden increase of the arriving packets causes the packet drop rate of Flow 3 to exceed p_{th}. Flow 3 is judged as overpumping and blocked from then on. This illustrates a typical behavior of an unresponsive flow in the flow-valve that an unresponsive flow manifests itself at a traffic increase. Since Flow 3 is CBR and never backs off, it is never freed. After Flow 3 is blocked, the two TCPs are able to use the full bandwidth again.

When Flow 4 starts again at time 20, Flow 4 is quickly detected as overpumping and blocked by the flow-valve. This illustrates the flow-valve's quick reaction to misbehaving flows. The average queue length is kept below max_{th} in the face of misbehaving flows.

Test 2

Test 2 is a 50-second-long sequence that illustrates interaction among 4 TCP flows. Flow 1 and Flow 2 are same as in Test 1. Flow 3 and Flow 4 emulate on/off sources with a large window size.

Flow 1 ftp from S1 to S3 (20 segment window size)

Flow 2 ftp from S2 to S3 (5 segment window size)

Flow 3 ftp from S1 to S4 (40 segment window size)
off period: 38–43

Flow 4 ftp from S2 to S4 (40 segment window size)
off period: 28–32, 37–43

When Flow 3 starts at time 7, it loses a burst of packets during the first slow-start and the estimated packet drop rate of Flow 3 exceeds the threshold p_{th}. Flow 3 is detected as overpumping but freed after it backs off. Flow 3 is never detected as overpumping again since it sets *ssthresh* at the first packet loss.

Figure 5.14 (a) shows the startup behavior of Flow 3. When the instantaneous queue length hits the limit, Flow 3 loses many packets and judged as overpumping. Flow 3 backs off exponentially and, at the 4th retransmission, the retransmission interval reaches the backoff threshold d_{th} and Flow 3 is freed.

At time 12, another flow, Flow 4, starts up and loses packets during the first slow-start as Flow 3 did. This time, the packet drop rate does not reach p_{th} and the flow-valve is not activated. Figure 5.14 (b) shows the startup behavior of Flow 4.

The difference between (a) and (b) in Figure 5.14 illustrates the impact of the backoff test to TCP. The penalty of the backoff test in (a) can be compared with a normal timeout in (b). The penalty of the normal timeout in (b) is smaller than the penalty of the flow-valve in (a) but they are in the same order. The difference will be smaller for most TCP implementations because of a larger minimum timeout value. It also suggests that, if a flow experiences multiple timeouts under heavy congestion, it is more harmful than a single penalty of the backoff test. It is also beneficial for a router; forcing exponential backoff is more effective in dissolving congestion than a few independent timeouts.

The packet loss at the first slow-start in this simulation is somewhat artificial by setting a small queue size limit. It is less likely to occur if the queue limit is big enough to absorb a transient increase of the instantaneous queue length. This setting is used to show a possible scenario in which a legitimate TCP is judged as overpumping and also to show the impact of the backoff test to a normal TCP.

The traffic pattern in Test 2 is highly dynamic but both the average queue length and the flow's packet drop rates are maintained within the proper range except the two first slow-starts mentioned above. It confirms that the flow-valve has no effect as long as RED works in the proper range.

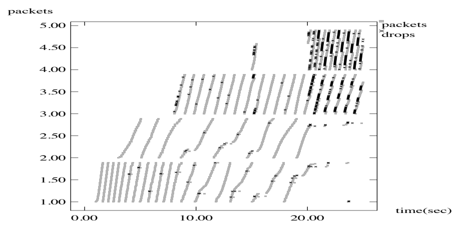
Summary

The flow-valve detects overpumping flows by a local decision and forces them to back off. Our design considerably simplifies an implementation of the penalty-box model.

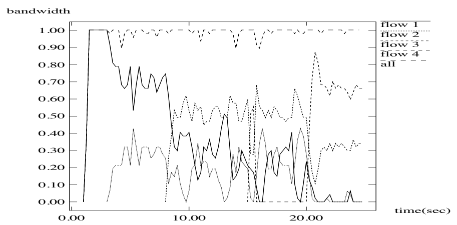
The flow-valve provides a simple rule; if a flow loses too many packet but still using too much bandwidth, the flow is forced to back off. The flow-valve provides an incentive for end-to-end congestion control to keep the packet drop rate low under moderate congestion, and to conservatively back off under heavy congestion.

Our simulation results have demonstrated that the flow-valve keeps traffic within the control range of RED even in the face of misbehaving flows. Further, it helps isolate undesirable behavior of conformant TCP.

The flow-valve has been successfully implemented onto ALTQ as an extension to the RED module. The flow-valve implementation on ALTQ is about 600 lines in C and is included in the ALTQ release.

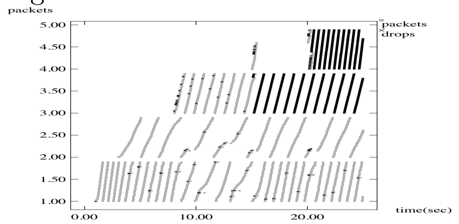


(a) sequence number of each flow

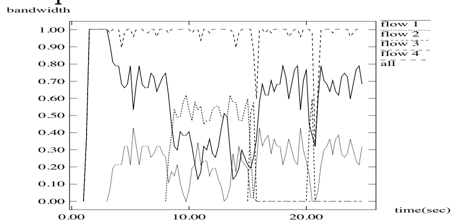


(b) bandwidth use of each flow

Figure 5.11: Test 1 with normal RED: unresponsive flows without flow-valve

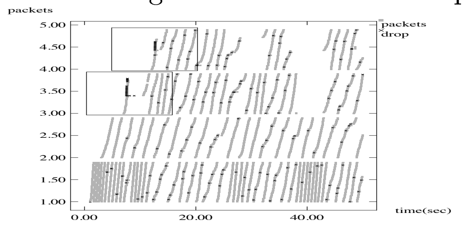


(a) sequence number of each flow

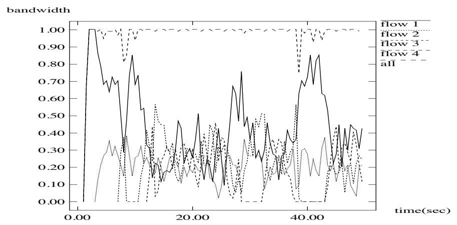


(b) bandwidth use of each flow

Figure 5.12: Test 1: unresponsive flows under flow-valve

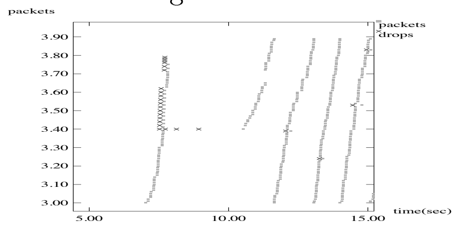


(a) sequence number of each flow

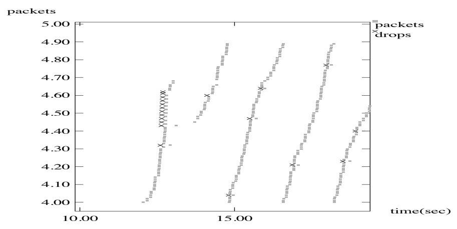


(b) bandwidth use of each flow

Figure 5.13: Test 2: 4 TCP sessions under flow-valve



(a) Flow 3: throttled by flow-valve



(b) Flow 4: normal timeout

Figure 5.14: Startup behaviors: flow-3 and flow-4 in Test 2

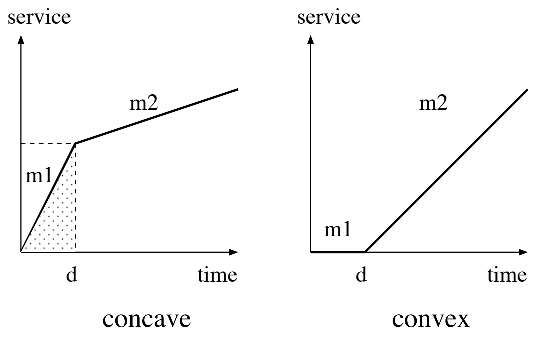


Figure 5.15: sample service curves

5.7 H-FSC (Hierarchical Fair Service Curve)

H-FSC [SZ97] supports both link-sharing and guaranteed real-time services. H-FSC employs a service curve based QoS model, and its unique feature is an ability to decouple delay and bandwidth allocation.

5.7.1 H-FSC algorithm

H-FSC maintains 2 service curves; one for real-time criteria and the other for link-sharing criteria. A service curve of H-FSC consists of 2 segments as shown in Figure 5.15. *m1* and *m2* are slopes of the 2 segments and *d* is the x-projection of the intersection that specifies the length of the 1st segment. Intuitively, *m2* specifies the long term throughput guaranteed to a flow, while *m1* specifies the rate at which a burst is served. When the slope of the 1st segment is larger than that of the 2nd segment, it is called *concave*. A service curve is either convex or concave.

A concave service curve provides a bounded burst similar to a token-bucket. The triangular area made by the 1st segment is roughly corresponds to the depth of a token-bucket, and the slope bounds the peak rate. The difference is that the peak rate of a token-bucket is a upper bound of the sending rate and is often set to the wire speed. On the other hand, H-FSC guarantees the rate defined by the 1st segment, and thus, it may be less than the wire speed.

A convex service curve, on the other hand, suppresses the initial traffic volume. *m1* of a convex curve must be 0 in the current implementation. A linear service curve is a special case of a convex curve with a NULL 1st

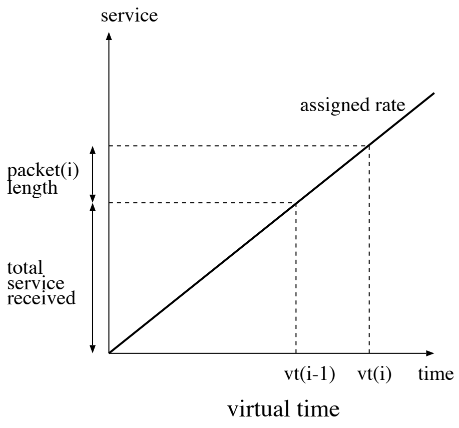


Figure 5.16: virtual time

segment. A linear service curve corresponds to a traditional virtual clock model and a good starting point for novice users.

Virtual Time

Each class keeps the total byte count already sent. When a class is backlogged, virtual time *vt* is calculated for the packet at the head of the class queue. *vt* is the x-projection of the service curve corresponding to $(total + packet\_len)$ as shown in Figure 5.16. As a result, *vt* of a class monotonically increases. By scheduling a class with the smallest *vt* among the backlogged classes, the bandwidth allocation becomes proportional to the service curve slope of each class.

A service curve is updated every time a class becomes backlogged. The update operation takes the minimum of (1) the service curve used in the previous backlogged period and (2) the original service curve starting at $(current\_time, total\_bytes)$. When a class has been idle long enough, the updated curve is equal to (2). On the other hand, when the class has been using bandwidth much more than its share, the updated curve is equal to (1). (1) and (2) can intersect when the class has been using bandwidth a little less than its share. In this case, the updated curve could have a different value of *d*. The operation is illustrated in Figure 5.17. It might be easier to see it as a half-filled token-bucket.

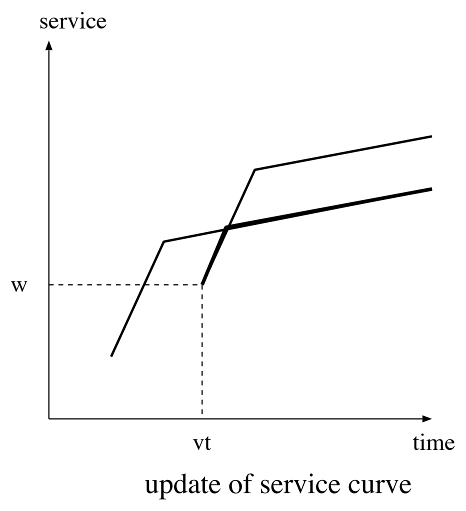


Figure 5.17: update of the virtual curve

H-FSC Scheduling

H-FSC has 2 independent scheduling mechanisms. Real-time scheduling is used to guarantee the delay and the bandwidth allocation at the same time. Hierarchical link-sharing is used to distribute the excess bandwidth available.

When dequeueing a packet, H-FSC always tries real-time scheduling first. If no packet is eligible for real-time scheduling, link-sharing scheduling is performed. H-FSC does not use class hierarchy for real-time scheduling.

Hierarchical Link-sharing

In H-FSC, only leaf classes have real packets but *vt* of an intermediate class is also maintained by summing up the total byte count used by its descendants. When dequeueing a packet, H-FSC's hierarchical scheduler walks through the class hierarchy from the root to a leaf class. At each level of the class hierarchy, the scheduler selects a class with the smallest *vt* among its child classes. When the scheduler reaches a leaf class, this leaf class is scheduled.

Note that the scheduler looks at only direct children at each level. Thus, the bandwidth allocation is proportional to the service curve slopes among the sibling classes but is not proportional among classes with different parents.

Real-time scheduling

As opposed to link-sharing scheduling, single consistent time is used for real-time scheduling. Each class keeps the cumulative byte count that is similar to the total byte count but only for packets scheduled by the real-time scheduling.

H-FSC computes the eligible time and deadline for each class. The eligible time and deadline are the x-projections of the head and tail of the next packet. A class becomes eligible for real-time scheduling when the current time becomes greater than the eligible time of the class. The real-time scheduler selects a class with the smallest deadline among eligible classes.

In the original H-FSC paper, a single service curve is used for both real-time scheduling and link-sharing scheduling. We have extended H-FSC to have independent service curves for real-time and link-sharing.

Decoupling service curves allows to independently control the guaranteed rate and the distribution of excess bandwidth. For example, it is possible to guarantee the minimum bandwidth of 2Mbps to 2 classes but the excess bandwidth is distributed with a different ratio.

It is also possible to set either of the service curves to be 0. When the real-time service curve is 0, a class receives only excess bandwidth. When the link-sharing service curve is 0, a class cannot receive excess bandwidth. Note that 0 link-sharing makes the class non-work conserving.

Note that, the link-sharing scheduling alone can guarantee the assigned bandwidth as long as the real-time service curve is equal to or smaller than the link-sharing service curve for all classes. But if the link-sharing service curve is smaller, assigned link-sharing bandwidth may not be provided.

5.7.2 Service Curve Based QoS Model

This section provides the formal definitions of the service curve based QoS model [Cru92, Cru95]. Each flow is associated with a service curve S_i which is a continuous non-decreasing function. A flow i is said to be guaranteed a service curve $S_i(\cdot)$, if for any time $t2$ when the flow is backlogged, there exists a time $t_1 < t_2$, which is the beginning of one of flow i's backlogged periods (not necessarily including t_2) such that the following holds

$$S_i(t_2 - t_1) \leq w_i(t_1, t_2) \tag{5.16}$$

where $w_i(t_1, t_2)$ is the amount of service received by flow i during the time interval $t_1, t_2]$. For packet systems, we restrict t_2 to be packet departure times.

A deadline is computed for each packet using a per flow deadline curve D_i. In an idealized fluid system, flow i's service curve is guaranteed if by any time t when flow i is backlogged, at least $D_i(t)$ amount of service is provided to flow i. Based on Equation (5.16)

$$D_i(t) = \min_{t_1 \in B_i(t)} (S_i(t - t_1) + w_i(t_1)) \tag{5.17}$$

where $B_i(t)$ is the set of all time instances, no larger than t, when flow i becomes backlogged, and $w_i(t_1) = w_i(0, t_1)$ is the total amount of service that flow i has received by time t_1. When flow i becomes backlogged for the first time, D_i is initialized to i's service curve $S_i(\cdot)$. Subsequently, whenever flow i becomes backlogged again at time a_i^k (the beginning of flow i's k-th backlogged period) after an idling period, D_i is updated according to the following:

$$D_i(a_i^k; t) = \min(D_i(a_i^{k-1}; t), S_i(t - a_i^k) + w_i(a_i^k)) \qquad t \geq a_i^k \tag{5.18}$$

The reason for which D_i is defined only fo $t \geq a_i^k$ is that this is the only portion that is used for subsequent deadline computations. Since D_i may not be an injection, its inverse function may not be uniquely defined. Here, we define $D_i^{-1}(a_i^k; t)$ to be the smallest value x such that $D_i^{-1}(a_i^k; x) = y$. Based on D_i, the deadline for a packet of length l_i at the head of flow i's queue can be computed as follows

$$d_i = D_i^{-1}(a_i^k; w_i(t) + l_i) \tag{5.19}$$

Intuitively, deadline d_i is computed from the amount of service already received $w_i(t)$ and the packet length l_i at the head of the queue. D_i^{-1} is to obtain the x-projection of the service curve from a given y value.

5.7.3 H-FSC Implementation

The implementation of the H-FSC algorithm is presented in this section. The code is taken from the H-FSC code in ALTQ but simplified by extracting only the relevant part to the H-FSC scheduler.

H-FSC Scheduler

The H-FSC scheduler has two criteria for scheduling: real-time criteria and link-sharing criteria. If there are eligible classes for real-time scheduling, the

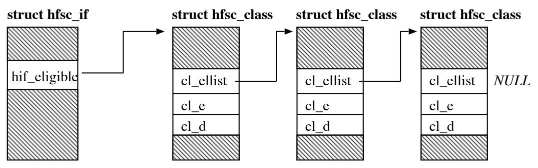


Figure 5.18: eligible list

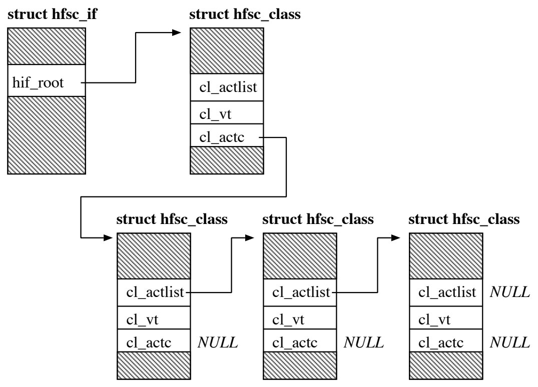


Figure 5.19: active child list

scheduler selects the class that has the minimum deadline among the eligible classes. If no eligible class is found, the link-sharing scheduling algorithm is applied to the hierarchical class tree.

Each class keeps three time values for scheduling. These values are the deadline, the eligible time and the virtual time, and they are computed using the corresponding service curves. The service curves are initialized when a class becomes backlogged, and the time values are updated every time the class sends a packet.

The real-time scheduling uses the deadline and the eligible time. When the eligible time of a class becomes smaller than the current time, the class becomes eligible. All eligible classes are kept in the eligible list for the interface. Figure 5.18 illustrates the eligible list. The *cl_e* field holds the eligible time of the class, and the *cl_d* field holds the deadline of the class. The eligible list is sorted by the deadline so that the scheduler always picks the first class in the eligible list.

When there is no eligible class, the scheduler uses the virtual time for link-sharing scheduling. Backlogged classes are kept in the active child list

```
/* runtime service curve */
struct runtime_sc {
    u_int64_t x;    /* current starting position on x-axis */
    u_int64_t y;    /* current starting position on x-axis */
    u_int64_t sm1;  /* scaled slope of the 1st segment */
    u_int64_t ism1; /* scaled inverse-slope of the 1st segment */
    u_int64_t dx;   /* the x-projection of the 1st segment */
    u_int64_t dy;   /* the y-projection of the 1st segment */
    u_int64_t sm2;  /* scaled slope of the 2nd segment */
    u_int64_t ism2; /* scaled inverse-slope of the 2nd segment */
};
```

Figure 5.20: *struct runtime_sc*

of the parent class. Figure 5.19 illustrates the active child list. The *cl_vt* field holds the virtual time of the class. The active child list is sorted by the virtual time so that the scheduler always picks the first class in the active child list. The scheduler walks through the class tree from the root class to a leaf class selecting the first active child at each level.

Runtime Service Curve

The time values, *cl_e*, *cl_d* and *cl_vt*, are computed from the runtime service curves. H-FSC requires a high resolution wall-clock for service curve computation. A higher clock resolution provides better precision to the scheduler. Our implementation uses a machine dependent high-resolution clock available on modern CPU architecture. On the Intel Pentium CPU, we use the TimeStamp Counter (TSC) that is a 64-bit counter driven by the CPU clock. For example, the TSC of a 200MHz CPU has 5 nsec resolution. Another advantage of using the machine dependent clock is that it takes only one machine cycle to read the clock.

H-FSC requires to compute the x-projection of the service curves for every packet, and this arithmetic operation is costly. In order to reduce the cost of arithmetic operations, service curve values are converted to the internal representation. The unit of time is converted to the clock resolution, that is, the unit time is 5 nsec on on 200MHz CPU and 2 nsec on 500MHz CPU. The advantage of this scheme is that TSC values do not need to be normalized, and can be directly used to compute the x-projection of a service curve, although time scale depends on the CPU clock frequency. Another optimization is to avoid divide operations for computing the x-projection by keeping the inverse of the slope.

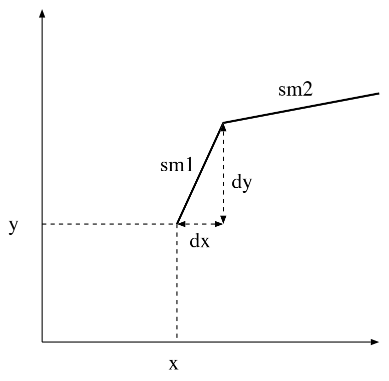


Figure 5.21: runtime service curve

Figure 5.20 shows the implementation information contained in the *runtime_sc* structure to describe a runtime service curve illustrated in Figure 5.21.

Each field in *runtime_sc* structure is 64 bit long to avoid arithmetic overflow in coordinate computation. The unit of the y axis of the service curve coordinates is byte. The unit of the x axis is the unit time of the machine dependent high resolution clock. *x* and *y* show the current starting position. *dx* and *dy* show the length of the first segment. *sm1* and *sm2* show the scaled slopes of the 2 segments. *ism1* and *ism2* are inverse values of *sm1* and *sm2*.

Several functions are provided to manipulate a service curve. The *rtsc_y2x* function computes the x-projection of the service curve for a given y value. *rtsc_y2x()* checks if the given y value is covered by the first segment or the by the second segment, and then, computes the x-projection using the segment that covers the y value. The *rtsc_min* function performs updating a service curve by taking the minimum of the service curve used in the previous backlog period and the original service curve starting at the current time and the already received service.

init_ed and update_ed Functions

The *init_ed* function is shown in Figure 5.22.

```
void init_ed(cl, next_len)
    struct hfsc_class *cl;
    int next_len;
{
    u_int64_t cur_time = read_machclk();

    /* update the deadline curve */
    rtsc_min(&cl->cl_deadline, cl->cl_rsc, cur_time, cl->cl_cumul);

    /* update the eligible curve.
     * for concave, it is equal to the deadline curve.
     * for convex, it is a linear curve with slope m2.
     */
    cl->cl_eligible = cl->cl_deadline;
    if (cl->cl_rsc->sm1 <= cl->cl_rsc->sm2)
        cl->cl_eligible.dx = cl->cl_eligible.dy = 0;

    /* compute e and d */
    cl->cl_e = rtsc_y2x(&cl->cl_eligible, cl->cl_cumul);
    cl->cl_d = rtsc_y2x(&cl->cl_deadline, cl->cl_cumul + next_len);

    ellist_insert(cl);
}
```

Figure 5.22: *init_ed*

init_ed initializes the two real-time service curves: the deadline service curve and the eligible service curve. *read_machclk()* reads the current time in a machine-dependent fashion. *rtsc_min()* performs updating the service curve by taking the minimum of the service curve used in the previous backlog period and the original service curve starting at the current time and the cumulative work.

In the case of a concave service curve, the eligible curve becomes equal to the deadline curve. In the case of a convex service curve, the eligible curve becomes a linear curve with slope *m2* [SZ97].

Then, the eligible time and the deadline for the next packet is calculated using the updated service curves by *rtsc_y2x()*. Finally, the class is placed onto the eligible class list.

```
void update_ed(cl, next_len)
    struct hfsc_class *cl;
    int next_len;
{
    cl->cl_e = rtsc_y2x(&cl->cl_eligible, cl->cl_cumul);
    cl->cl_d = rtsc_y2x(&cl->cl_deadline, cl->cl_cumul + next_len);

    ellist_update(cl);
}
```

Figure 5.23: *update_ed*

The *update_ed* function is shown in Figure 5.23. The eligible time and the deadline for the next packet are computed from the current service curves. *rtsc_y2x()* computes the x-projection of the service curve for a given y value. *ellist_update()* re-orders the eligible class list by the deadline value.

The *update_d* function is a variant of *update_ed*, and updates only the deadline. *update_d()* is called when the class is scheduled by the link-sharing criteria. Because the deadline depends on the length of the next packet, the deadline needs to be recomputed. On the other hand, the eligible time does not change and is left untouched.

init_v and update_v Functions

The *init_v* function is shown in Figure 5.24.

```
void init_v(cl, len)
    struct hfsc_class *cl;
    int len;
{
    struct hfsc_class *min_cl, *max_cl;

    while (cl->cl_parent != NULL) {
        if (cl->cl_nactive++ > 0)
            break;  /* already active */

        min_cl = actlist_first(cl->cl_parent->cl_actc);
        if (min_cl != NULL) {
            /* set vt to the average of the min and max classes.
             * if the parent's period didn't change, don't decrease vt.
             */
            max_cl = actlist_last(cl->cl_parent->cl_actc);
            vt = (min_cl->cl_vt + max_cl->cl_vt) / 2;
            if (cl->cl_parent->cl_vtperiod == cl->cl_parentperiod)
                vt = max(cl->cl_vt, vt);
            cl->cl_vt = vt;
        } else
            cl->cl_vt = 0; /* no packet is backlogged.  set vt to 0 */

        /* update the virtual curve */
        rtsc_min(&cl->cl_virtual, cl->cl_fsc, cl->cl_vt, cl->cl_total);

        cl->cl_vtperiod++;  /* increment vt period */
        cl->cl_parentperiod = cl->cl_parent->cl_vtperiod;
        if (cl->cl_parent->cl_nactive == 0)
            cl->cl_parentperiod++;

        actlist_insert(cl);
        cl = cl->cl_parent;  /* go up to the parent class */
    }
}
```

Figure 5.24: *init_v*

init_v function initializes the link-sharing service curve. *init_v* is called when a leaf class becomes backlogged. The *while* loop goes through the parent classes and, if the parent has no active child, it initializes the parent too.

The newly-activated child class is inserted to the active child list of the parent. The initial virtual time is set to the average of the minimum vt and the maximum vt of the active sibling classes in order to bound the discrepancy between any two active sibling classes. If the parent is in the same active period since the last time this child became inactive, the virtual time should not be decreased because the other active siblings have not received equivalent service while this child was inactive.

Once the initial vt is set, the link-sharing service curve is initialized by *rtsc_min*. The newly-activated child is placed onto the active class list of the parent class.

```
void update_v(cl, len)
    struct hfsc_class *cl;
    int len;
{
    while (cl->cl_parent != NULL) {
        cl->cl_total += len;
        if (cl->cl_fsc != NULL) {
            cl->cl_vt = rtsc_y2x(&cl->cl_virtual, cl->cl_total);
            actlist_update(cl);  /* update the vt list */
        }
        cl = cl->cl_parent;  /* go up to the parent class */
    }
}
```

Figure 5.25: *update_v*

The *update_v()* function in Figure 5.25 computes a new vt value. *rtsc_y2x()* computes the x-projection of the current link-sharing service curve from the current total service value. *actlist_update()* re-order the active child list by the vt values. The *while* loop goes through the parent classes until it reaches the root class.

set_active and set_passive Functions

The *set_active* function is shown in Figure 5.26. *set_active()* is called when a class becomes backlogged to initialize the service curves of the class. If *cl_rsc* is NULL, the scheduler does not use real-time scheduling for this class. Similarly, if *cl_fsc* is NULL, the scheduler does not use link-sharing scheduling for this class.

```
void set_active(cl, len)
    struct hfsc_class *cl;
    int len;
{
    if (cl->cl_rsc != NULL)
        init_ed(cl, len);
    if (cl->cl_fsc != NULL)
        init_v(cl, len);
}
```

Figure 5.26: *set_active*

The *set_passive* function is shown in Figure 5.27. *set_passive()* removes the class from the eligible list, and from the active child list. For the active class list, the *while* loop visits the parent classes and, if there are no other active child, the parent is also removed from the active child list.

```
void set_passive(cl)
    struct hfsc_class *cl;
{
    if (cl->cl_rsc != NULL)
        ellist_remove(cl);

    if (cl->cl_fsc != NULL) {
        while (cl->cl_parent != NULL) {
            if (--cl->cl_nactive == 0)
                actlist_remove(cl);
            else
                break;  /* still has active children */
            cl = cl->cl_parent;  /* go up to the parent class */
        }
    }
}
```

Figure 5.27: *set_passive*

hsfc_enqueue Function

The *hfsc_enqueue* function is shown in Figure 5.28.

```
int hfsc_enqueue(ifp, m, pktattr)
    struct ifnet *ifp;
    struct mbuf *m;
    struct altq_pktattr *pktattr;
{
    struct hfsc_if *hif;
    struct hfsc_class *cl;

    cl = pktattr->pattr_class;
    hfsc_addq(cl, m);
    if (qlen(cl->cl_q) == 1)
        set_active(cl, m_pktlen(m));
    return (0);
}
```

Figure 5.28: *hfsc_enqueue*

hfsc_enqueue enqueues the packet to the class queue. The corresponding class is obtained from the classifier result set in *pktattr*. The next step is to call *hfsc_addq()* to enqueue the packet to the class. Finally, if the queue length becomes 1, this is the start of a new backlog period so that *set_active()* is called to initialize the service curves.

hsfc_dequeue Function

The *hfsc_dequeue* function is shown in Figure 5.29.

```
struct mbuf *hfsc_dequeue(ifp)
    struct ifnet *ifp;
{
    struct hfsc_if *hif;
    struct hfsc_class *cl;
    struct mbuf *m;
    int len, next_len, realtime = 0;

    if ((cl = ellist_get_mindl(hif->hif_eligible)) != NULL) {
        realtime = 1;
    } else {
        cl = hif->hif_rootclass;
        while (is_a_parent_class(cl))
            cl = actlist_first(cl->cl_actc);
    }
    m = hfsc_getq(cl);

    len = m_pktlen(m);
    update_v(cl, len);
    if (realtime)
        cl->cl_cumul += len;
    if (!qempty(cl->cl_q)) {
        if (cl->cl_rsc != NULL) {
            next_len = m_pktlen(qhead(cl->cl_q));
            if (realtime)
                update_ed(cl, next_len);
            else
                update_d(cl, next_len);
        }
    } else
        set_passive(cl);
    return (m);
}
```

Figure 5.29: *hfsc_dequeue*

hfsc_dequeue() is called to send a packet to the network so that it implements the scheduling algorithm to select a packet to send. *hfsc_dequeue()* decides which scheduling criteria should be used, selects a class based on the criteria, remove a packet from the selected class, and updates the service curves of the class.

When there is an eligible class, the real-time criteria is used for schedul-

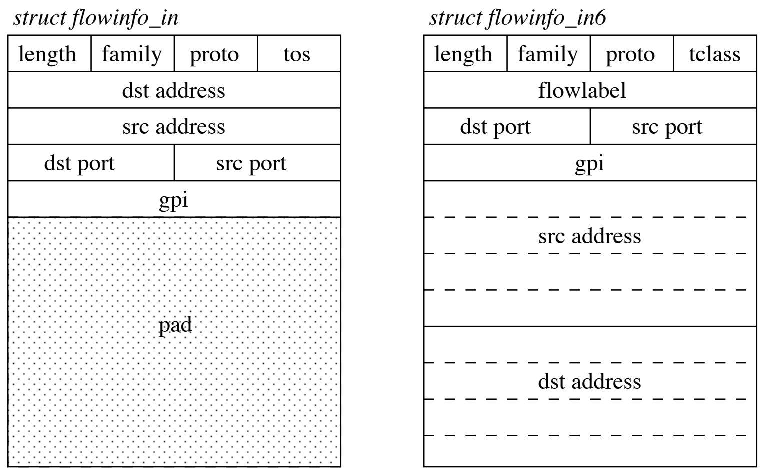


Figure 5.30: flow information for IPv4 and IPv6 used by classifier

ing. *ellist_get_mindl()* tries to find an eligible class that has the minimum deadline among the eligible classes. If there is no eligible class, the link-sharing criteria is used for scheduling. The scheduler walks through the class hierarchy from the root class to a leaf class. At each level, the scheduler selects the class with the minimum vt by actlist_first(). At this point, one class is selected to dequeue a packet, and hfsc_getq() is called to remove a packet from the class queue.

The rest of the block updates the service curves of the class, and maintains the eligible class list and the active class list. update_v() updates the link-sharing service curve. When the real-time criteria is used, the packet length is added to the cumulative work of the class. If the class queue is not empty, it updates the real-time service curves for the next packet in the class queue. If the queue becomes empty, set_passive() is called to remove the class from the eligible class list and the active child list.

5.8 Classifier

A packet classifier divides packets into different classes. To classify a packet, the classifier performs filter-matching by comparing packet header fields (e.g., IP addresses and port numbers) for each packet. Efficient implementation of a classifier is still under active research [BGP$^+$94, DDPP98, SVSW98, GM99].

In our implementation, classifiers support both IPv4 and IPv6. Figure 5.30 shows flow information for IPv4 and IPv6 used by the classifier. When performing filter matching, the classifier extracts these fields from a packet, and compares with packet filters.

Filters are hashed by the destination addresses in order to reduce the number of filter matching operations. However, if a filter doesn't specify a destination address, the filter is put onto the wildcard-filter list. When classifying a packet, the classifier tries the hashed list first, and if no matching is found, it tries the wildcard list.

In this implementation, per-packet overhead grows linearly with the number of wildcard filters. A classifier could be implemented more efficiently, for example, using a directed acyclic graph (DAG) [BGP$^+$94].

5.9 Traffic Conditioning

5.9.1 Traffic Conditioning Components

Our implementation of a traffic conditioning block follows the conceptual model of Diffserv routers described in [BSB99]. A traffic conditioning block is attached to an input interface, and contains a group of traffic conditioning components including classifier, meter, and action elements.

Traffic conditioning elements can be divided into 2 groups: simple action elements and fan-out elements. A simple action element takes only one action to a packet. Possible actions are (1) pass a packet (2) drop a packet (3) mark a certain DSCP value to a packet. On the other hand, a fan-out element has N different actions, and selects one of the actions according to some conditions. Classifiers and meters are fan-out elements.

ALTQ supports the following traffic conditioning element types:

- simple action elements

  pass: a pass action takes no action to a packet.

  mark: a mark action sets a DSCP value to the DS field of a packet.

  drop: a drop action discards a packet.

- fan-out elements

  tbmeter: a token bucket meter [BSB99] measures conformance to a token bucket profile, and takes either an in-profile action or an out-of-profile action. A token bucket profile has two parameters, a rate and a depth.

trTCM: a 2-rate 3-color maker [HG99] has 2 token bucket profiles, one for a committed rate and the other for a peak rate, and takes one of 3 actions. A trTCM takes a green action when the input is under the committed profile, a yellow action when the input is more than the committed profile but under the peak profile, a red action otherwise.

TSWTCM: a time-sliding window 3-color marker [FS99] has 3 parameters, a committed rate, a peak rate, and an averaging interval. A TSWTCM measures conformance to a profile, and stochastically takes one of green, yellow and red actions.

A top level conditioner that holds all traffic conditioning elements on an interface implements a classifier. A top level conditioner first classifies an incoming packet, and then, passes the packet to the corresponding elements. When an element is a fan-out element, the packet is further passed to one of the next elements. A traffic conditioning process for a packet terminates when it reaches a simple action element and a final action for the packet is determined.

ALTQ currently does not support shaping elements described in [BBC+98, BSB99]. A shaping element substantially differs from other traffic conditioning elements since it cannot be processed in a sequential execution pass. If a shaper is required, it can be implemented as part of a non-work conserving queueing discipline at an output interface instead of a traffic conditioning element.

Traffic Conditioning Actions

struct tc_action in Figure 5.31 implements traffic conditioning actions. *struct tc_action* consists of an action code and value pair. An action code represents an action type such as mark and drop, and an action value is a code dependent parameter. A mark action takes a DSCP value as a parameter. Both a drop action and a pass action do not have a parameter. Mark, drop and pass actions are simple actions.

A handle action connects an action to another element, and takes a handle (identifier) of a next element as a parameter. A next action is a kernel internal representation of a handle action for efficient execution, and its parameter is a pointer to the next element instead of the handle of the element. A handle action is used in user programs but the kernel converts the handle of an element to a pointer to the element and holds it as a next action type.

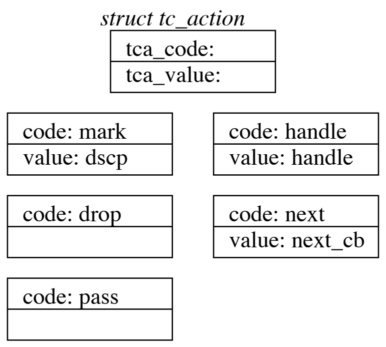


Figure 5.31: traffic conditioning actions

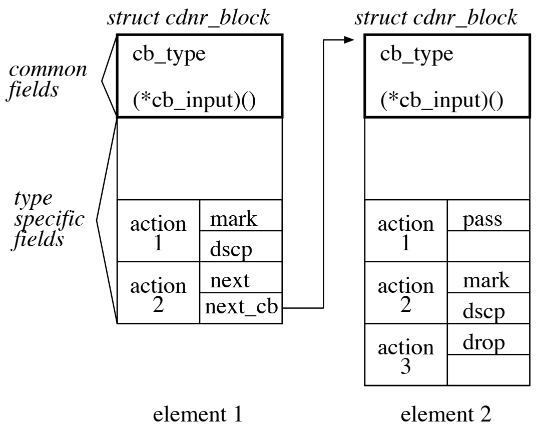


Figure 5.32: traffic conditioner block

Traffic Conditioner Block

Simple traffic conditioning elements are self-contained within *struct tc_action* but other traffic conditioning elements such as meters require a structure called *conditioner block*. *struct cdnr_block* consists of common fields and type specific fields as shown in Figure 5.32.

The common fields include a conditioner type field and a pointer to an input function. The input function is invoked when a packet is passed to the traffic conditioning element. The input function performs type specific process and returns a result action.

The type specific fields of *struct cdnr_block* include internal state of the el-

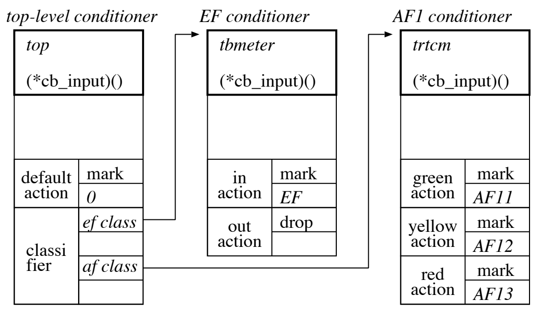


Figure 5.33: traffic conditioner example

ement and result actions. Figure 5.32 shows examples of conditioner blocks. Element 1 has two actions: "action 1" is a mark action and "action 2" is a next action connected to Element 2. Element 2 has three actions: a pass action, a mark action and a drop action.

Figure 5.33 shows a more concrete example in which a top level conditioner holds two conditioner blocks: one for Expedited-Forwarding (EF) and the other for Assured-Forwarding (AF). The top level conditioner implements a classifier. There are two classes defined for the classifier, one for EF and the other for AF. The EF conditioner is a token bucket meter that marks the EF DSCP value for in-profile packets, and drops out-of-profile packets. The AF conditioner is a trTCM that marks three drop precedence values.

When a packet is passed to the top level conditioner, the top level conditioner first classifies the packet. If it does not match either the EF class or the AF class, the default action is taken. In this case, the default DSCP value 0 is marked. If a packet belongs to the EF class, the classifier returns a pointer to the EF conditioner block. The packet is passed to the input function of the EF conditioner, and then, the packet is marked or dropped. If a packet belongs to the AF class, the classifier returns a pointer to the AF conditioner block. The packet is passed to the input function of the AF conditioner, and then, a precedence value is set to the packet.

5.10 Summary

QoS forwarding mechanisms are realized on top of the ALTQ framework. ALTQ implements a wide variety of queueing disciplines. We have reviewed FIFOQ, WFQ, CBQ, RED, ECN, Flow-valve, H-FSC and RIO available in

ALTQ. Each discipline trades off implementation complexity with the ability to provide good performance. ALTQ also implements other forwarding mechanisms such as classifiers and traffic conditioners. Our experiences with these forwarding mechanisms prove the flexibility of the ALTQ framework.

Since QoS components are integrated into the ALTQ framework, it becomes possible to combine components that were originally proposed and implemented independently. It also becomes possible to examine issues in implementing QoS mechanisms proposed only through theory or simulation. We have identified limitations and issues in implementing QoS components such as effects of device buffer and timer granularity.

Chapter 6

Management Mechanisms

QoS management mechanisms are a set of tools and libraries used for traffic management. For example, a setup tool is needed to configure queueing disciplines by combining QoS components and setting appropriate parameters to the components. A monitoring tool is indispensable for operation. A sophisticated QoS manager requires admission control and policy control so that it needs a library to interface with admission control and policy control. Simple management tools are useful in the early stage of QoS deployment but, as the technology matures, more elaborate systems will be used in this area. As the ALTQ users grow in size and diversity, the importance of the management tools is increasing.

6.1 QoS Manager

A QoS manager, called *altqd*, is a stand-alone user program to manage ALTQ queueing disciplines and other QoS mechanisms.

At startup, *altqd* reads a configuration file and sets up forwarding mechanisms in the kernel accordingly. Then, *altqd* waits for events that could trigger updates of QoS components. *altqd* maintains the states of the QoS components installed into the kernel so that admission control can be done by *altqd* without consulting the kernel. A client program can retrieve information about QoS components by querying to *altqd*.

In order to control QoS components from other programs (e.g., rsvpd), most of the *altqd* code is implemented as a library called *libaltq*.

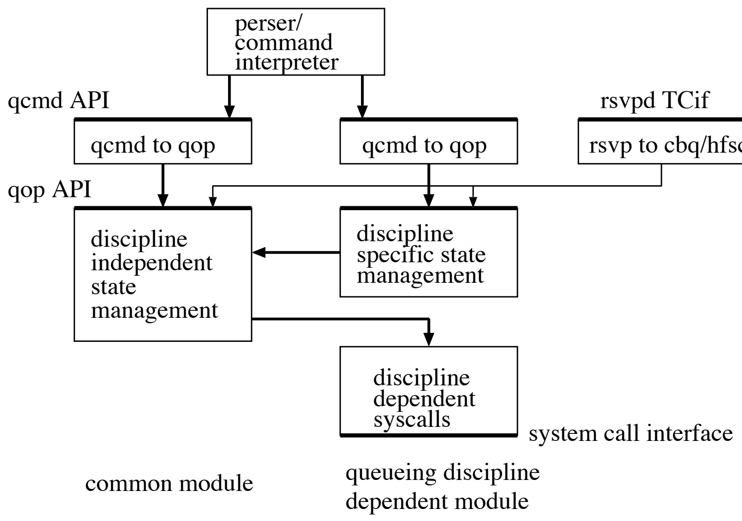


Figure 6.1: ALTQ API

6.2 ALTQ Library

The ALTQ library, *libaltq*, consists of three layers: the parser/interpreter layer, the Queue Command (QCMD) API layer and the Queue Operation (QOP) API layer as shown in Figure 6.1. The QCMD API layer and the QOP API layer are divided into the common modules and the discipline dependent modules.

The parser/interpreter layer implements a parser to read a configuration file and to interpret line commands from a terminal when in the terminal mode. The parser/interpreter layer is built on top of the QCMD API.

The QCMD API is a simplified version of the QOP API. Most parameters in the QOP API are pointers to structures whereas most parameters in the QCMD API are strings. The QCMD API is designed mainly for the parser but applications that do not need efficiency or detailed information can also benefit from the QCMD API.

The QOP API is an application programming interface of ALTQ queueing disciplines and provides a uniform interface to different queueing disciplines. The API provides:

- uniform handling of different disciplines including classes and filters.

- a single configuration file for different disciplines on multiple interfaces

- memory management
- error handling

The QOP API layer consists of a common module and a set of discipline specific modules. Many queue operations are discipline independent (e.g., enabling ALTQ), and thus, done through the common module. The common module also manages memory allocation and deallocation, name mapping of interfaces, classes and filters.

On the other hand, some operations are intrinsically discipline specific since they need discipline specific parameters. These operations are supported by a discipline specific module.

Discipline specific modules are also responsible for system calls since system calls are also discipline dependent.

Additional sets of APIs can be built on top of the QOP API. For example, an interface module for RSVP is built on the QOP API. The RSVP interface module translates the RSVP traffic control parameters into discipline specific parameters. Implementing DiffServ PEP (policy enforcement point) would require an approach similar to RSVP.

Interface Operations

The interface operations are add, delete, enable, disable and clear. The add operation is discipline dependent. The add operation attaches a specified queueing discipline to an interface. The delete operation detaches the queueing discipline from the interface. It also deletes all classes and filters associated with the interface. The enable operation activates the attached queueing discipline on the interface. The disable operation inactivates the queueing discipline. The clear operation reinitializes the queueing discipline.

Class Operations

The class operations are add, delete, and modify. The add and modify operations are discipline dependent. The add operation creates a specified class for an interface. The delete operation removes a specified class from the interface. It also deletes all filters associated with the class. The modify operation changes class parameters.

Filter Operations

The filter operations are add and delete. The add operation creates a classifier filter for the specified class. It also tries to detect a situation in which a new filter conflicts with a previously-defined filter entry, and returns an error if a conflict is detected. The delete operation removes a specified filter from the class.

Traffic Conditioner Operations

The ALTQ library also supports traffic conditioners. Traffic conditioners are handled in a way similar to classes but differ in that they are placed at an input interface. Another important difference is that dependencies are in the reverse order. In a class hierarchy, classes are created from the root node to a leaf node, and removed from a leaf node to the root node. On the other hand, in a traffic conditioner hierarchy, conditioners are created from a leaf node to the root node, and removed from the root node to a leaf node.

Error Handling

The API functions return an error on a failure. The API errors are categorized into five groups: system errors, class errors, filter errors, admission errors, and policy errors.

Admission Control

Each discipline specific module implements admission control. Both CBQ and H-FSC require that the sum of the bandwidth assigned to child classes cannot exceed the bandwidth assigned to the parent class. It is checked when a class is created, and returns an admission control error if it fails.

6.3 Monitoring Tools

Tools to monitor the counters and other statistics are indispensable for development and operations. Routers have various counters that can be monitored through the SNMP protocol. With a variety of QoS components, monitoring these counters is vital to both development and operations.

All the QoS components implemented in ALTQ have counters. The counters as well as other information that could be used for debugging can be read through the *ioctl* system call.

altqstat is a user program to display statistics of a queueing discipline and traffic conditioning elements. When it starts, *altqstat* obtains the queueing discipline type attached to an interface, and then, calls the discipline specific module to obtain and display statistics.

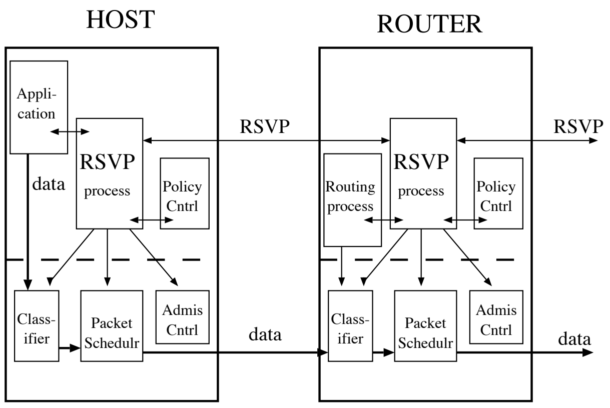


Figure 6.2: RSVP implementation model

In addition, *altqstat* can display the current class hierarchy of queueing disciplines and traffic conditioners by communicating with *altqd*. A HTTP-style request/response protocol is used to retrieve the information from *altqd* through a UNIX domain socket.

Regarding SNMP support, the MIB (Management Information Base) for Diffserv is still in the middle of a standardization process at IETF at this writing. Once the DiffServ MIB is standardized, ALTQ will support SNMP using an SNMP agent program publicly available for UNIX.

6.4 RSVP

rsvpd is a daemon program that handles RSVP signaling messages in the RSVP release from ISI [ISI]. *rsvpd* is based on the RSVP implementation model in Figure 6.2. The actual implementation of *rsvpd* is shown in Figure 6.3.

rsvpd is designed to make use of a system dependent traffic control module, if it is available. The kernel traffic control interface is defined to abstract different traffic control systems.

To provide a traffic control module to *rsvpd* is one of the initial goals of ALTQ since no traffic control module was available for BSD UNIX. Naturally, the ALTQ design was influenced by the kernel traffic control interface of *rsvpd*. For example, (1) a traffic control module is bound to an interface. (2) a traffic control module can be divided into classifier, packet scheduler, and admission control. (3) traffic control operations are categorized into fil-

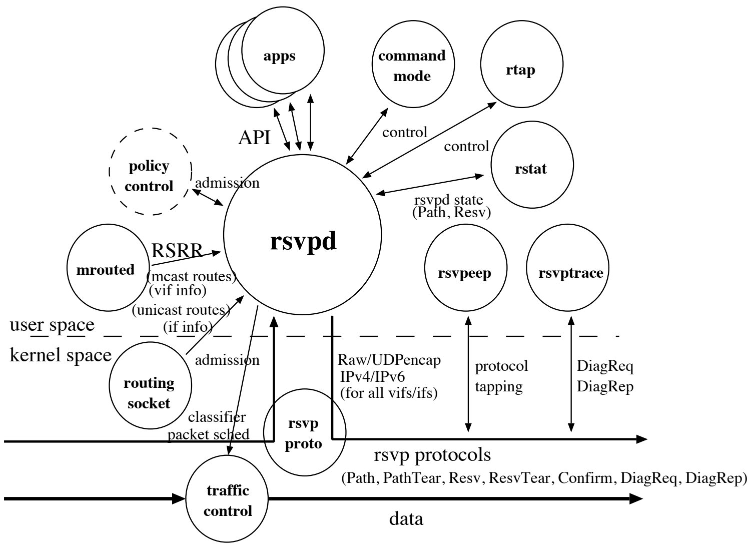


Figure 6.3: RSVP daemon implementation

ter operations and flowspec operations: filter operations work on the classifier and flowspec operations work on the packet scheduler and the admission control. (4) a filter or a reservation can be dynamically made, modified, and destroyed.

The kernel traffic control interface of *rsvpd* is listed below.

TC_init initializes the traffic control module for a given interface.

TC_AddFlowspec makes a reservation for a given flow using the corresponding flowspec. In ALTQ, the flowspec is first passed to the admission control and, if successful, the flowspec is translated to discipline-specific parameters, and then, a new class is created for the reservation.

TC_ModFlowspec modifies a flowspec of a given flow. In ALTQ, the new flowspec is first passed to the admission control and, if successful, the class for the reservation is updated to match the new flowspec.

TC_DelFlowspec deletes flow for specified handle; it also deletes all corresponding filter specs. In ALTQ, the class for the reservation is deleted.

TC_AddFilter adds a filter for an existing flow. In ALTQ, the filterspec is translated to ALTQ filter parameters, and the new filter is set to the class.

TC_DelFilter deletes existing filter. In ALTQ, the corresponding filter is deleted.

TC_Advertise updates OPWA (One Pass With Advertising) ADSPEC.

6.5 DiffServ

A DiffServ network can be built by configuring QoS components provided by ALTQ. Because a DiffServ network can be statically configured, it does not require a dedicated program like *rsvpd* for RSVP.

In order to build a DiffServ network with ALTQ, one needs to design a network; (1) fix the network topology. (2) select PHBs to use. (3) assign DSCPs to the PHBs. (4) provision resources (e.g., bandwidth) for PHBs. (5) allocate resources to each PHB. Then, traffic conditioners and queueing disciplines should be configured accordingly.

6.5.1 Expedited-Forwarding PHB

The Expedited-Forwarding (EF) PHB [JNP99] is intended to build a low loss, low latency, low jitter, assured bandwidth, end-to-end service through DS domains. Such a service appears to the endpoints like a point-to-point connection or a "virtual leased line".

Creating such a service requires:

1. Configuring nodes so that the aggregate has a well-defined minimum departure rate. (the minimum departure rate should be independent of the intensity of other traffic at the node.)
2. Conditioning the aggregate (via policing and shaping) so that its arrival rate at any node is always less than the node's configured minimum departure rate.

The EF PHB provides the first part of the service. The network boundary traffic conditioners provide the second part.

The requirement of the EF PHB for a queueing discipline is that the queueing discipline should guarantee a rate configured for EF traffic. In ALTQ, either CBQ or H-FSC can be used to realize the EF PHB.

For traffic conditioning, a token bucket meter can be used. In ALTQ, a token bucket meter should be configure to mark the EF DSCP for in-profile packets and to discard out-of-profile packets.

The network should be provisioned in such a way that its arrival rate at any node is always less than the node's configured minimum departure rate.

6.5.2 Assured Forwarding PHB Group

Assured Forwarding (AF) PHB Group [HBWW99] is a set of PHBs that offer a high level of assurance that packets will be delivered as long as the behavior aggregate conforms to a given service profile. The network will accept excess traffic from the behavior aggregate but the excess traffic may have a higher probability of being discarded. The AF PHB Group defines 4 independent classes; a class is assigned to a behavior aggregate. Within each class, 3 levels of drop precedence are defined. Packets in the same behavior aggregate are delivered in arriving order regardless of their drop precedence levels.

The requirement of the AF PHB group for a queueing discipline is that the queueing discipline should support bandwidth allocation for 4 classes, with 3 drop precedence levels within each class. In ALTQ, either CBQ or H-FSC with the RIO dropper can be used to realize the AF PHB group.

The requirement for traffic conditioning is that the traffic conditioner should compare packets of a behavior aggregate with a 2-level traffic profile, and then, mark one of 3 DSCP values into a packet accordingly. In ALTQ, either trTCM or TSWTCM can be used as an AF traffic conditioner.

6.6 Summary

QoS management mechanisms are a set of tools and libraries used for traffic management, and control QoS forwarding mechanisms implemented in the kernel. In the early stage of the ALTQ deployment, they were simple development tools. As the ALTQ users grow in size and diversity, the importance of the management tools is increasing. QoS management mechanisms also include important future research themes such as policy servers and QoS monitoring.

A QoS manager, called *altqd*, is a stand-alone user program to manage ALTQ QoS mechanisms. *altqd* reads a configuration file and sets up queueing disciplines accordingly. In order to control queueing discipline from other programs (e.g., rsvpd), most of the code is implemented as a library.

The ALTQ library allows to handle queueing disciplines at high level in a discipline independent manner. The ALTQ library consists of three layers: the parser/interpreter layer, the Queue Command (QCMD) API layer and the Queue Operation (QOP) API layer. The QCMD API layer and the QOP API layer are divided into the common modules and the discipline dependent modules. The ALTQ library is also used to realize RSVP-capable or DiffServ-capable nodes.

Chapter 7

Results

In this section, we present the performance of the implemented disciplines. Note that traffic management becomes more important at a bottleneck link, and thus, its performance does not necessarily correspond to the high-speed portion of a network.

We use primarily CBQ and H-FSC to illustrate the traffic management performance of ALTQ since CBQ and H-FSC are more complex and interesting than other implemented disciplines. CBQ is non-work conserving, needs a classifier, and uses the combination scheduling of priority and weighted-round robin. H-FSC is basically work conserving but can be configured to be non-work conserving, needs a classifier, and uses a unique service-curve based scheduling. However, we do not attempt to outline the details specific to CBQ or H-FSC.

7.1 Test System Configuration

We have measured the performance using three PentiumIII machines (all 700MHz with 440BX chipset) running FreeBSD-4.1/altq-3.0a. Figure 7.1 shows the test system configuration. Host A is a source, host B is a router, and host C is a sink. CBQ or H-FSC is enabled only on the interface of host B connected to host C. The link between host A and host B is 155Mbps ATM. The link between host B and host C is either 155M ATM, 10baseT, 100baseTX, or 128K serial line. When 10baseT is used, a dumb hub is inserted. When 100baseTX is used, a direct connection is made by a cross cable, and the interfaces are set to the full-duplex mode. Efficient Network Inc. ENI-155p cards are used for ATM, Intel EtherExpress Pro/100B cards are used for 10baseT and 100baseTX. RISCom/N2 cards are used for a

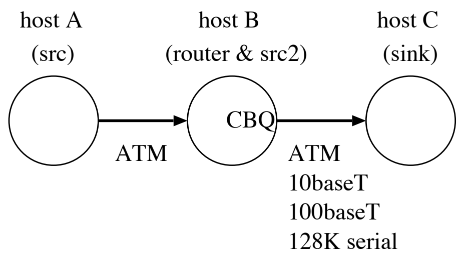


Figure 7.1: test system configuration

synchronous serial line.

The Netperf benchmark program [Jon93] is used with ±5.0% confidence interval at 99% confidence level. We use TCP to measure the packet forwarding performance under heavy load. In contrast to UDP, which consumes CPU cycles to keep dropping excess packets, TCP quickly adapts to the available bandwidth, and thus does not waste CPU cycle. However, a suitable window size should be selected carefully according to the end-to-end latency and the number of packets queued inside the network. Also, one should be careful about traffic in the reverse direction, since ACKs play a vital role in TCP. Especially with shared media (e.g., Ethernet), sending packets could choke TCP ACKs.

7.2 Overhead

The overhead introduced by H-FSC or CBQ consists of four steps: (1) extract flow information from an arriving packet. (2) classify the packet to the appropriate class. (3) select an eligible class for sending next. (4) update the state of the class. There are many factors which affect the overhead: structure of class hierarchy, priority distribution, number of classes, number of active classes, rate of packet arrival, distribution of arrival, and so on. Hence, the following measurements are not intended to be complete.

7.2.1 Throughput Overhead

Table 7.1 compares TCP throughput of H-SFC and CBQ with that of the original FIFO queueing, measured over different link types. A small configuration with three classes is used to show the minimum overhead of H-FSC and CBQ. There is no other background traffic during the measurement.

Table 7.1: H-FSC and CBQ throughput

Link Type	orig. FIFO (Mbps)	HFSC (Mbps) (overhead %)	CBQ (Mbps) (overhead %)
ATM	133.60	132.47 (0.85%)	132.46 (0.85%)
10baseT	6.73	6.67 (0.89%)	6.66 (1.04%)
100baseTX	94.38	94.31 (0.07%)	94.33 (0.05%)
loopback			
MTU 16384	1028.59	956.12 (7.05%)	857.01 (16.68%)
MTU 9180	763.15	694.91 (8.94%)	584.56 (23.40%)
MTU 1500	393.23	320.07 (18.60%)	243.99 (37.95%)

From Table 7.1, no significant overhead is observed because packet processing can overlap the sending time of the previous packet. As a result, use of H-FSC or CBQ does not affect the throughput.

The measurements over the software loopback interface with different MTU sizes are also listed in the table. These values show the limit of the processing power and the overhead of H-FSC and CBQ in terms of CPU cycle. H-FSC does have about 9% overhead with 9180-byte MTU, and about 19% overhead with 1500-byte MTU. CBQ does have about 23% overhead with 9180-byte MTU, and about 38% overhead with 1500-byte MTU.

Figure 7.2 plots the measured TCP throughput in the loopback interface over a wider range of MTU size. The overhead in bits per second does not change much with different MTU sizes because the overhead is per packet and does not affect the performance gain by larger packet sizes.

A larger packet size has a clear advantage in performance, especially when a sophisticated packet scheduling is used. It also shows that a current PC can handle more than 300Mbps with bi-directional loopback load. That is, a PC-based router has processing power enough to handle multiple 100Mbps-class interfaces; CPU load will be much lower with physical interfaces since DMA can be used.

As Table 7.1 and Figure 7.2 show, the overhead of H-FSC is much smaller than that of CBQ. H-FSC was implemented much later than CBQ with various improvements from our experience with CBQ. The improvements include optimization for the ALTQ structure, avoiding integer overflow check

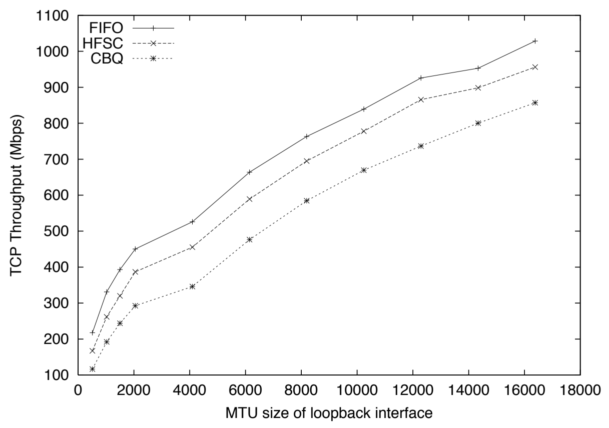


Figure 7.2: MTU size and TCP throughput on a loopback interface

by using 64bit when necessary, use of the Pentium Timestamp Counter for time measurement. However, the core algorithm of CBQ is simpler than H-FSC. The performance gain in the H-FSC implementation seems to come from optimizations in the peripheral code rather than the core scheduling algorithm.

7.2.2 Latency Overhead

Table 7.2 shows the overhead in latency over ATM and 10baseT. In this test, request/reply style transactions are performed using UDP, and the test measures how many transactions can be performed per second. The rightmost two columns show the calculated average round-trip time (RTT) and the difference in microseconds. Again, both H-FSC and CBQ have three classes, and there is no background traffic.

The increase of RTT by H-FSC or CBQ is almost constant regardless of packet size or link type, since a packet scheduling has a per-packet overhead. The overhead per packet is about 3 microseconds for H-FSC, and 10 microseconds for CBQ.

7.2.3 Scalability Issues

Both H-FSC and CBQ are designed such that a class tree has relatively small number of classes; a typical class tree would have less than 20 classes. Still, it is important to identify the scalability issues. Although a full test of

Table 7.2: HFSC and CBQ latency

Link Type	queue type	request/ response (bytes)	trans. per sec	calc'd RTT (usec)	diff (usec)
ATM	FIFO	1, 1	5479.20	182.51	
	HFSC		5421.41	184.45	1.95
	CBQ		5206.64	192.06	9.55
	FIFO	64,64	4396.19	227.47	
	HFSC		4357.87	229.47	2.00
	CBQ		4196.85	238.27	10.80
	FIFO	1024,64	2487.56	402.00	
	HFSC		2469.88	404.88	2.88
	CBQ		2418.33	413.51	11.51
	FIFO	8192,64	597.44	1673.81	
	HFSC		596.19	1677.32	3.51
	CBQ		593.24	1685.66	11.85
10baseT	FIFO	1,1	3335.35	299.82	
	HFSC		3307.62	302.33	2.51
	CBQ		3242.09	308.44	8.63
	FIFO	64, 64	2495.90	400.66	
	HFSC		2484.67	402.47	1.81
	CBQ		2445.08	408.98	8.33
	FIFO	1024,64	792.72	1261.48	
	HFSC		790.82	1264.51	3.03
	CBQ		786.82	1270.94	9.46

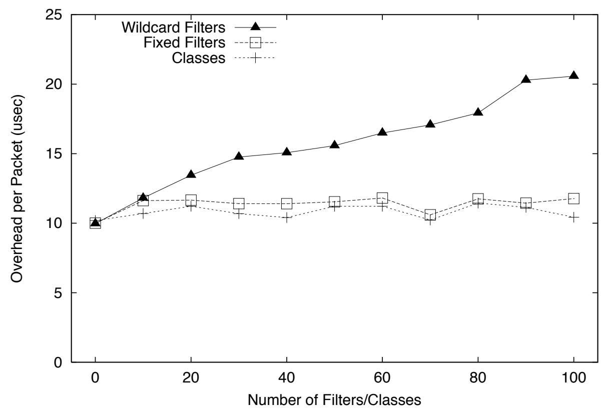


Figure 7.3: effect of number of filters/classes

scalability is difficult, the following measurements provide some insight into it. Figure 7.3 shows how the latency changes when we add additional filters or classes; up to 100 filters or classes are added. The values are differences in calculated RTT from the original FIFO queueing measured over ATM with 64-byte request and 64-byte response. CBQ is used for the test. H-FSC shares the same classifier with CBQ so that the result can be applied to H-FSC as well.

The "Wildcard Filters" plot and "Fixed Filters" plot in the graph show the effect of two different types of filters. To classify a packet, the classifier performs filter-matching by comparing packet header fields (e.g., IP addresses and port numbers) for each packet. In our implementation, class filters are hashed by the destination addresses in order to reduce the number of filter matching operations. However, if a filter doesn't specify a destination address, the filter is put onto the wildcard-filter list. When classifying a packet, the classifier tries the hashed list first, and if no matching is found, it tries the wildcard list. In this implementation, per-packet overhead grows linearly with the number of wildcard filters. A classifier could be implemented more efficiently, for example, using a directed acyclic graph (DAG) [BGP+94].

On the other hand, the number of classes doesn't directly affect the packet scheduler. As long as classes are underlimit, the scheduler can select the next class without checking the states of the other classes. However, to schedule a class which exceeds its share, the scheduler should see if there is a class to be scheduled first. Note that because the maximum number of overlimit classes is bound by the link speed and the minimum packet size, the overhead will not grow beyond a certain point.

When there are overlimit classes, it is obvious that CBQ performs much better than FIFO. It is shown in Section 7.6.

7.2.4 Overhead of Other Disciplines

The latency overhead can be used to compare the minimum overhead of the implemented disciplines. Table 7.3 shows the per-packet latency overhead of the implemented disciplines measured over ATM with 64-byte request and 64-byte response. The values are differences in calculated RTT from the original FIFO queueing.

"tbr" is a token bucket regulator without any queueing discipline. All queueing disciplines include the overhead of a token bucket regulator. "TBM", "trTCM, "tswTCM" are a token bucket meter, a two-rate three-color marker, and a time-sliding window three-color marker. The difference of the origi-

Table 7.3: per-packet overhead comparison (usec)

tbr 0.11	FIFOQ 0.17	HFSC 1.89	CBQ 10.73
WFQ 0.70	RED 5.90	HFSC+RED 2.29	CBQ+RED 10.96
TBM 0.75	RIO 13.51	HFSC+RIO 15.30	CBQ+RIO 23.42
trTCM 0.89	tswTCM 2.00	HFSC+RIO+trTCM 16.01	CBQ+RIO+trTCM 24.07

nal FIFO and our FIFOQ is that the enqueue and dequeue operations are macros in the original FIFO but they are function calls in ALTQ.

Note that the latency values shown in Table 7.3 are the minimal latency when there is no background traffic. The values will be considerably different when there is other traffic. However, there is no easy way to obtain or compare those numbers.

Impact of Latency Overhead

Network engineers seem to be reluctant to put extra processing on the packet forwarding path. But when we talk about the added latency, we should also take queueing delay into consideration. For example, a 1KB packet takes 800 microseconds to be put onto a 10Mbps link. If two packets are already in the queue, an arriving packet could be delayed more than 1 millisecond. Thus, several microseconds can be negligible when there is competing traffic. The dominant factor in end-to-end latency is normally queueing delay, and thus, sophisticated queueing is worth it.

7.3 Bandwidth Allocation

Figure 7.4, 7.5 and 7.6 shows the accuracy of bandwidth allocation of CBQ over different link types. TCP throughputs were measured when a class is allocated 5% to 95% of the link bandwidth. The plot of 100% shows the throughput when the class can borrow bandwidth from the root class. As the graphs show, the allocated bandwidth changes almost linearly over ATM, 10baseT and a serial line. However, considerable deviation is observed over 100baseTX, especially during the range from 15% to 55%.

The problem in the 100baseTX case is the timer granularity problem described in section 5.3.1. The calculated limit rate is 9.6Mbps, and the

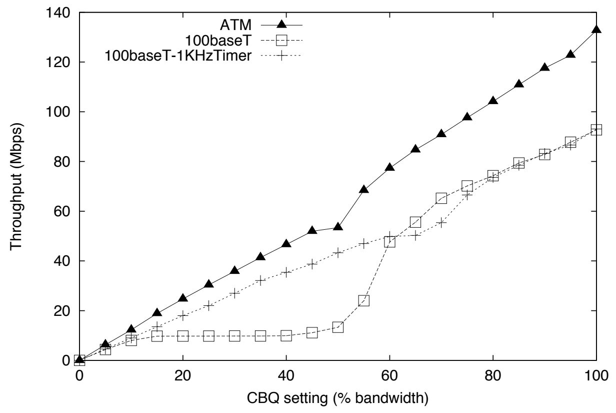


Figure 7.4: bandwidth allocation over ATM/100baseTX

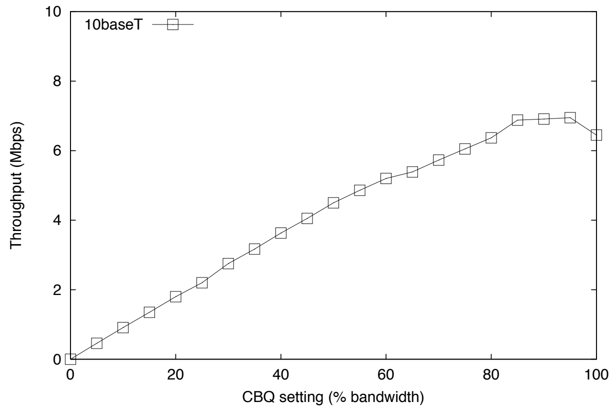


Figure 7.5: bandwidth allocation over 10baseT

throughput in the graph stays at this limit up to 55%. Then, as the sending rate increases, packet events help CBQ scale beyond the limit. To back up this theory, we tested the performance of the kernel whose timer granularity is modified from 10ms to 1ms. With this kernel, the calculated limit rate is 96Mbps. The result, shown as *100baseTX-1KHzTimer*, is satisfactory, which also agrees with theory. Note that the calculated limit of the ATM case is 58.8Mbps, and we can observe a slight deviation at 50%, but packet events help CBQ scale beyond the limit. Also, note that 10baseT shows sat-

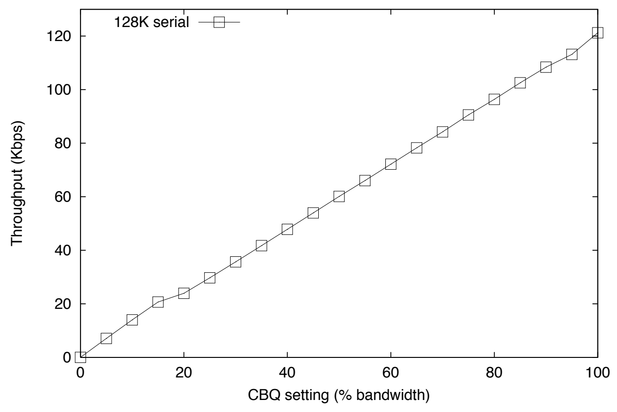


Figure 7.6: bandwidth allocation over 128K serial

uration of shared-media, and performance peaks at 85%. The performance of 10baseT drops when we try to fill up the link.

7.4 Bandwidth Guarantee

Figure 7.7 illustrates the success of bandwidth guarantee by CBQ over ATM. Four classes, one each allocated 10Mbps, 20Mbps, 30Mbps and 40Mbps, are defined. A background TCP flow matching the default class is sent during the test period. Four 20-second-long TCP flows, each corresponding to the defined classes, start 5 seconds apart from each other. To avoid oscillation caused by process scheduling, class-0 and class-2 are sent from host B and the other three classes are sent from host A. All TCP connections are trying to fill up the pipe, but the sending rate is controlled by CBQ at host B.

The *cbqprobe* tool is used to obtain the CBQ statistics (total number of octets sent by a class) every 400 msec via *ioctl*, and the *cbqmonitor* tool is used to make the graph. Both tools are included in the release.

As we can see from the graph, each class receives its share and there is no interference from other traffic. Also note that the background flow receives the remaining bandwidth, and the link is almost fully utilized during the measurement.

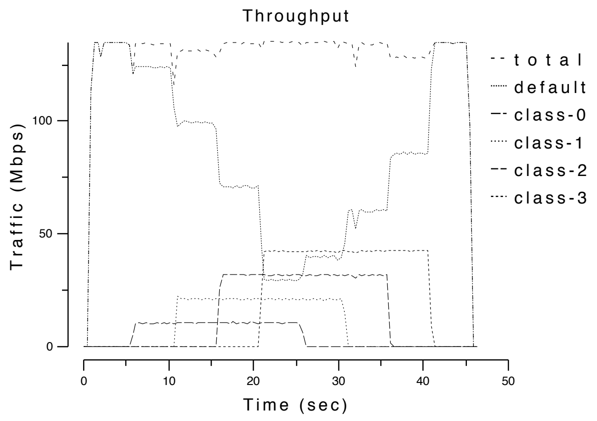


Figure 7.7: CBQ bandwidth guarantee

7.5 Link Sharing by Borrowing

Link sharing is the ability to correctly distribute available bandwidth in a hierarchical class tree. Link-sharing allows multiple organizations or multiple protocols to share the link bandwidth and to distribute "excess" bandwidth according to the class tree structure. Link-sharing has a wide range of practical applications. For example, organizations sharing a link can receive the available bandwidth proportional to their share of the cost. Another example is to control the bandwidth use of different traffic types, such as telnet, ftp, or real-time video.

The test configuration is similar to the two agency setting used by Floyd [FJ95]. The class hierarchy is defined as shown in Figure 7.8 where two agencies share the link, and interactive and non-interactive leaf classes share the bandwidth of each agency. In the measurements, Agency X is emulated by host B and agency Y is emulated by host A. Four TCP flows are generated as in Figure 7.9. Each TCP tries to send at its maximum rate, except for the idle period. Each agency should receive its share of bandwidth all the time even when one of the leaf classes is idle, that is, the sum of class-0 and class-1 and the sum of class-3 and class-4 should be constant.

Figure 7.10 shows the traffic trace generated by the same method described for Figure 7.7. The classes receive their share of the link bandwidth and, most of the time, receive the "excess" bandwidth when the other class in the same agency is idle. High priority class-4, however, receives more

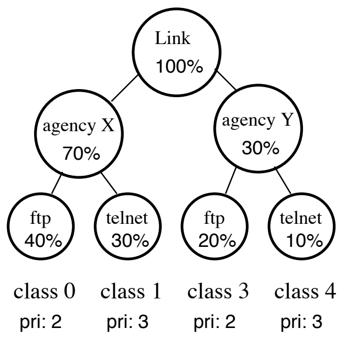


Figure 7.8: class configuration

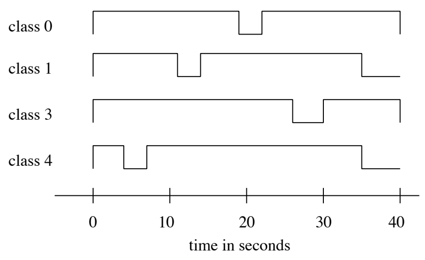


Figure 7.9: test scenario

than its share in some situations (e.g., time frame:22–25). The combination of priority and borrowing in the current CBQ algorithm, especially when a class has a high priority but a small share of bandwidth, does not work so well as in the NS simulator [FJ95].

The borrowing algorithm of CBQ does not have a clear rule to distribute the "excess" bandwidth. The first borrowable class found by the scheduler is selected, and the round robin pointer is advanced when a borrowable class is scheduled. This works reasonably well within the same priority. However, a high priority class is preferred when there are multiples borrowable classes with different priorities.

To confirm the cause of the problem, we tested with all the classes set to the same priority. As Figure 7.11 shows, the problem of class-4 is improved. Note that, even if interactive and non-interactive classes have the same priority, interactive classes are likely to have much shorter latency because interactive classes are likely to have much fewer packets in their queues.

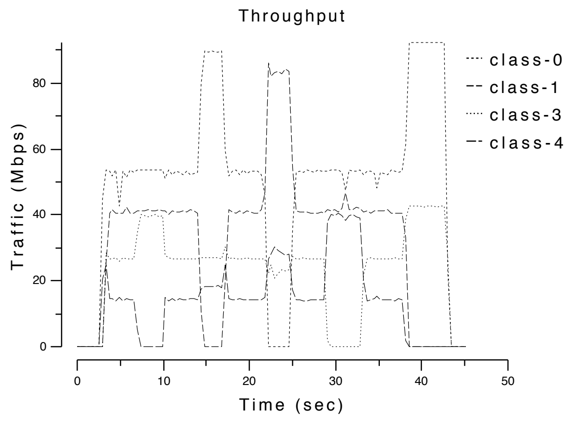


Figure 7.10: link-sharing trace

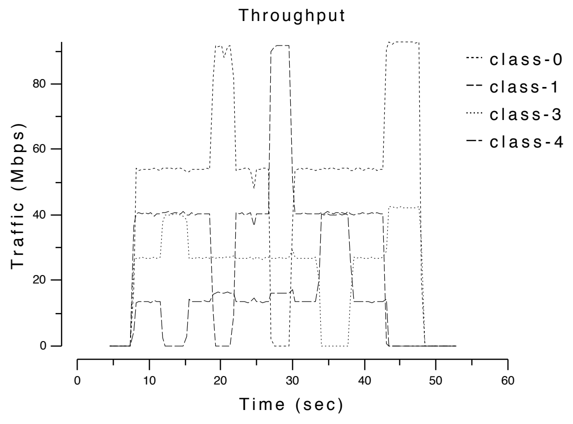


Figure 7.11: link-sharing trace with same priority

7.6 Delay Differetiation

In this test, the latency differentiation between two classes is measured. The method is similar to one in 7.2.2 but with background traffic as shown in Figure 7.12. The link between host B and host C is 10baseT set to the full-duplex mode.

The background traffic is generated from host B to host C using TCP

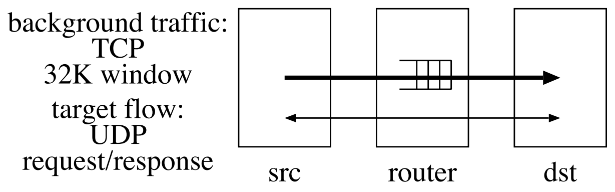


Figure 7.12: latency test environment with competing traffic

Table 7.4: latency with competing traffic

discipline type	condition	trans. per sec	calc'd RTT (msec)
HFSC	no background traffic	2465.46	0.41
	no differentiation	31.80	31.45
	differentiated class	347.50	2.88
CBQ	no background traffic	2431.87	0.41
	no differentiation	31.86	31.39
	differentiated class	325.06	3.08
	diff class w/ priority	346.76	2.88

with 32KB window size. This background traffic creates a constant backlog at the output interface of host B. If packets for the entire 32KB window is backlogged, it will create about 26 millisecond queueing delay.

The target traffic is generated from host A to host C. It is 64-byte request/reply style transactions using UDP, and the test measures how many transactions can be performed per second.

H-FSC and CBQ are tested with different conditions. The result is shown in Table 7.4. When there is no background traffic the target flow can achieve more than 2400 transactions per second for both H-FSC and CBQ.

The "no differentiation" condition means that both the target traffic and the background traffic is put into the same class, that is, they are served in a FIFO order. The number of transactions drops to 32 per second. The calculated RTT is 31 milliseconds.

In the "differentiated class" condition, a separated class is assigned to the target flow. The target class in CBQ has the same priority with the background class. In this setting, the target traffic and the background traffic are served by the weighted-round robin. Also, the target class is assigned a higher priority in the "diff class w/ priority" condition. In this setting, the target class will be always served first, and the performance becomes close to H-FSC.

With a separated class assigned, the number of transactions improves to 347 per second. The calculated RTT is about 2.9 milliseconds. Since a 1500-byte packet takes 1.2 milliseconds on 10bseT, the result shows that the average latency of the target traffic is less than the delay of 3 packets.

It was necessary to configure the token bucket regulator properly to achieve low latency for the target traffic. This is explained in the next section.

7.7 Token Bucket Regulator

In this test, the effects of parameter settings of a token bucket regulator are explored. A token bucket regulator has 2 parameters: token rate and bucket size. In Table 7.5, a TCP stream with the 32KB window size is used to measure the throughput. The link between host B and host C is 100baseT set to the full-duplex mode.

The FIFOQ is used as a queueing discipline to show the number of backlogged period. "packets" and 'period" in the table show the number of packets and the number of backlogged period observed by FIFOQ. The backlogged period is incremented every time the empty queue becomes backlogged. Thus, as long as FIFOQ has a constant backlog, the period does not increase. The period count in the table also includes counts for the test setup and report packets.

The first raw in the table shows a case without a token bucket regulator. The measure throughput is 94Mbps. The period count is close to the number of packets, which indicates that the queue is empty most of the time due to the large buffer in the device. That is, packets are queued in the device buffer and not in FIFOQ.

The token rates of 100M, 98M and 94M are used to illustrate the relation between the token rate and the real transmission rate. The measured transmission rate without a token bucket regulator is about 94Mbps seen from the application. 94Mbps at the application level corresponds to about 98Mbps at the interface level since the packet size includes the headers of TCP, IP and Ethernet. Thus, the token rate of 100Mbps is a little higher than the actual transmission rate, 98Mbps is equal to the transmission rate, and 94Mbps is a little lower than the transmission rate.

When the bucket size is too small, the driver cannot buffer enough packets in order to fill the pipe. When the bucket size is only 1KB, the throughput is less than 40Mbps for all the token rates. As the bucket size increases,

Table 7.5: effects of token bucket parameters

token rate (bps)	bucket size (bytes)	throughput (bps)	packets	periods
(none)	–	94.32M	161505	150981
100M	1K	39.44M	68192	14104
	2K	94.32M	161507	119480
	4K	94.31M	161500	2963
	8K	94.35M	161561	20161
	16K	94.34M	161556	99004
	32K	94.35M	161567	161567
	64K	94.33M	161543	161543
	128K	94.32M	161517	161517
98M	1K	33.65M	58195	2836
	2K	90.02M	154803	98
	4K	93.77M	160574	50
	8K	93.88M	160788	245
	16K	93.98M	160993	736
	32K	94.06M	161076	7092
	64K	94.03M	161096	12239
	128K	94.08M	161112	30749
94M	1K	32.62M	56406	21
	2K	88.53M	152213	25
	4K	89.83M	153905	43
	8K	90.05M	154216	115
	16K	90.17M	154417	573
	32K	90.18M	154491	1137
	64K	90.22M	154497	1991
	128K	90.26M	154564	2408
50M	1K	7.17M	12416	15
	2K	30.75M	52905	20
	4K	31.17M	53511	21
	8K	33.52M	57444	21
	16K	33.56M	57484	39
	32K	44.23M	75747	4346
	64K	48.01M	82232	18367
	128K	48.06M	82306	20050

the throughput reaches 94Mbps for the token rate of 100Mbps and 98Mbps. The throughput of the 94Mbps token rate is limited up to 90Mbps that is the corresponding rate at the application level.

The result shows that the bucket size of 4KB is enough for 100Mbps 100baseTX, at least for this driver. The bucket sizes more than 4KB do not seem to contribute to the throughput.

On the other hand, the period counts reveal the number that the queue

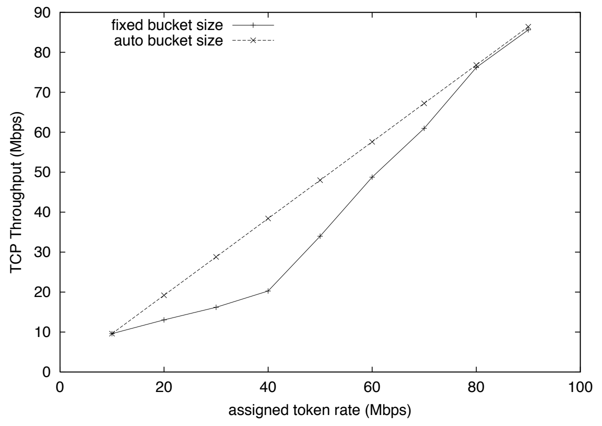


Figure 7.13: rate limiting by token bucket regulator

becomes empty. The token rates of 98Mbps and 94Mbps clearly show small period counts. As the bucket size increases, the queue becomes empty more frequently.

The token rate of 100Mbps is less effective than the 98Mbps or 94Mbps case. This can be explained as follows: The bucket size limits the amount of packets that the driver dequeues at a time. Still, a token rate higher than the transmission rate allows that packets are dequeued faster than the driver can transmit in a longer term. As a result, excessive packets can accumulate in the device buffer. Thus, fewer packets are left in FIFOQ for a given TCP window size.

This situation is not so visible when only throughput is measured. However, the problem becomes clear when latency is measured. In the latency test in Table 7.4, the token rate was set to 9.4Mbps that is slightly lower than the actual transmission rate. If the token rate is set to 10Mbps, the latency degrades considerably due to the delay caused by the device buffer.

The token rate of 50Mbps illustrates a situation where the token rate is much smaller than the actual transmission rate and the driver cannot dequeue packets from transmission complete interrupts. Because packets are dequeued only at a kernel timer interval, the target throughput cannot be achieved unless the bucket size is large enough to cover the timer interval.

The required bucket size in bytes for a given target rate in bits-per-second and a timer frequency is:

$$size = rate/8/Hz$$

When the target rate is 10Mbps and the timer frequency is 100Hz, the required bucket size is about 12KB. When the target rate is 50Mbps, and the timer frequency is 100Hz, the required bucket size becomes about 61KB.

Figure 7.13 shows the measured throughput for varying target rate. The "fixed bucket size" shows the throughput of a 12KB bucket size. The "auto bucket size" shows the bucket size is calculated by the above equation. As the graph shows, the throughput cannot reach the target rate with the 12KB bucket size, but is satisfactory with the auto-scaled bucket size.

The results presented in this section show the importance of the correct setting of the token bucket parameters. The token rate should be set to a value close to the actual transmission rate, and the bucket size should be set to a small value but big enough not to affect the throughput.

Although it is required to find out the correct setting, it also proves the design of a token bucket regulator. That is, a device-independent and tunable mechanism is needed to control the behavior of a driver.

7.8 Summary

In this section, the test results are presented. The performance of the ALTQ system is the combined performance of the framework and QoS forwarding mechanisms to use. Thus, we have evaluated the performance of the system including the framework and QoS forwarding mechanisms.

The overhead in throughput and latency are examined using a simple setting without background traffic. The performance of bandwidth allocation and guarantee, and the link-sharing ability are confirmed. Then, the latency with competing traffic is presented. The parameter setting of a token bucket regulator is also discussed.

Through these tests, we have confirmed that the ALTQ system is able to provide various types of QoS with a minimal overhead.

Chapter 8

ALTQ Applications

8.1 Traffic Management Issues

There are people arguing that there are no need for QoS control since bandwidth will be cheap and abundant in the future. However, traffic management is not a choice between QoS and non-QoS but a wide rage of spectrum. For example, at one extreme, every single packet could be precisely controlled at every router. At the other extreme, packets could be transferred even without flow control. However, both approaches are too expensive to realize and to manage so that they have no practical importance.

For a properly provisioned network, queue management could be considered as a precaution in case of congestion. It also works as a protective measure against misbehaving flows, misconfiguration, or misprovisioning. The effect of active queue management will not be so visible for such a properly provisioned network. However, it will virtually shift the starting point of congestion so that the effect is similar to increasing the link capacity.

Traffic management needs a good balance between controlling and provisioning at each level and among different levels. It is important to find a balance point that is cost-effective as well as administratively easy to manage.

8.1.1 Time Scale of Traffic Management

Traffic management consists of a diverse set of mechanisms and policies. Traffic management includes pricing, capacity planning, end-to-end flow control, packet scheduling, and other factors. These cover different time scales and complement one another.

The time scale of queueing is a packet transmission time. Queueing is effective to manage short bursts of packets. End-to-end flow control in turn manages the rate of a flow in a larger time scale. An important role of end-to-end flow control is to keep the size of packet bursts small enough to be manageable by queueing. To this end, large capacity itself is of no use for managing bursts in the packet level time scale. On the contrary, widening gap in link speed makes bursts larger and larger so that it makes managing traffic more important, especially at bandwidth gap points.

8.1.2 Controlling Bottleneck Link

Typically, bottleneck points are entries of WAN connections and they are the source of packet loss and delay. Queue management is most effective at those points.

Congestion is often caused by a small number of bulk data sessions (e.g., web images, ftp) so that isolating such sessions from other types of traffic will significantly improve network performance. It also serves as a protective measure. On the other hand, RED will substantially improve the performance of cooperative TCP sessions.

There are network administrators trying to keep the link utilization as high as possible. However, queueing theory tells us that the system performance drastically drops if the link utilization becomes close to 100%. It is a phenomenon that a queue is no longer able to absorb fluctuations in packet arrivals. Ideally, the link capacity should be provisioned so that the average link utilization is under a certain point, say 80%.

A difficulty in deploying queue management is that queueing manages only outgoing traffic and the beneficiaries are on the other side of a link. Queueing is not appropriate for managing incoming traffic because the queue is almost always empty at the exit of a bottleneck. In order to manage incoming traffic, queue management should be placed at the other end of the WAN link but most organizations do not have control over it.

8.1.3 Queueing Delay

Network engineers tend to focus on the forwarding performance. That is, how many packets can be forwarded per second, or how long it takes to forward a single packet. However, once the forwarding overhead becomes less than a packet transmission time, the throughput reaches the wire speed by a pipeline effect. Although further cutting down the overhead improves the delay, it has no effect if the queue is not empty.

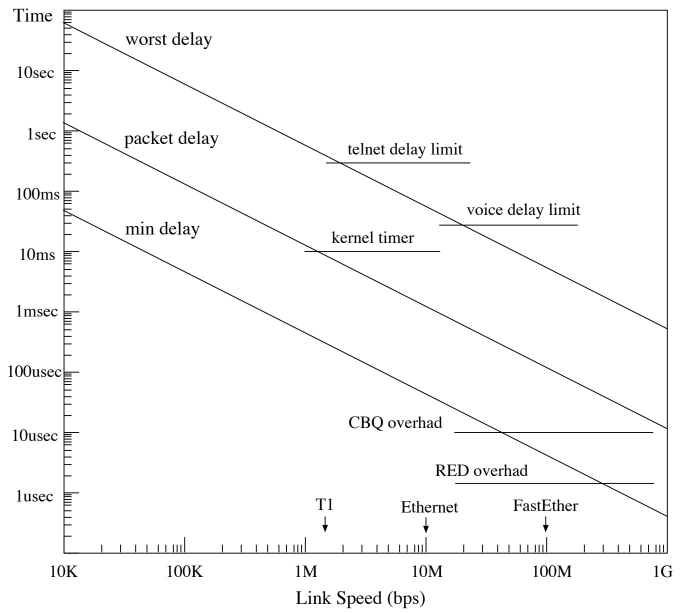


Figure 8.1: queueing and link speed

On the other hand, queueing delay (waiting time in the queue) is by orders of magnitude larger than the forwarding delay. It implies that, if there is a bottleneck, high-speed forwarding does not improve the delay because most of the delay comes from queueing delay. Thus, we should pay closer attention to queueing delay, once the throughput reaches the wire speed.

8.1.4 Impact of Link Speed

It is important to understand how the effects and the overheads of queueing are related to the link speed. To illustrate the issues involved, Figure 2 plots packet transmission time and queueing delay on varying link speed in log-log scale. **min delay** and **packet delay** show the required time to transmit a packet at the wire speed with the packet size of 64 bytes and 1500 bytes, respectively. These are the minimum time required to forward a packet by a store-and-forward method. **worst delay** shows the worst case queueing delay when the queue is full, assuming that the maximum queue length is 50 (the default value in BSD UNIX) and all packets are 1500-byte long. On the other hand, Table 1 shows the per-packet overhead of different queueing disciplines measured on a PentiumPro 200MHz machine [Cho98].

The per-packet overhead of queueing is independent of link speed. By a simplistic analysis, queueing overhead would be negligible if the per-packet overhead is less than **min delay**, and could be acceptable if the per-packet overhead is less than **packet delay**. The overhead of CBQ is 10usec. It would be negligible up to 40Mbps and acceptable even at 1Gbps. The overhead of RED is 1.6usec. It would be negligible up to 300Mbps.

On the other hand, the delay requirement of an application is also independent of link speed. If an interactive telnet session needs the latency to be less than 300 msec, preferential scheduling is required for link speed less than 1.5Mbps. If a voice stream needs the latency to be less than 30 msec, preferential scheduling is required for link speed less than 20Mbps.

Although there are other performance factors and the analysis is simplistic, it illustrates the effects of the link speed on queueing. In summary, queueing does not have significant overhead for commonly used link speed. Preferential scheduling improves interactive response on a slow link, and improves real-time traffic on a medium speed link.

8.1.5 Building Services

So far, we have looked at the behavior of a single router. An end-to-end service quality can be obtained by concatenating router behaviors along the communication path. For example, a traffic stream from user A to user B can be controlled such a way that the average rate is 1Mbps, the peak rate is 3Mbps and the packet delay is less than 1msec.

However, to make useful services, a network as a whole should be properly configured in a consistent way. In order to guarantee a service quality, it is necessary to configure all routers along the path and control all incoming traffic to these routers.

The diffserv working group at IETF is trying to establish a framework for various types of differentiated services [BBC$^+$98]. In the diffserv model, a network that supports a common set of services is called “DS domain”. A DS domain should be built in such a way that all incoming packets are policed at the boundary. Incoming packets are classified, measured and marked according to the user contract. These boundary actions are called “traffic conditioning”. Inside a DS domain, internal routers (called DS interior nodes) perform preferential packet scheduling using only the packet header field (DS field) that has been marked at the boundary.

Traffic management mechanisms can be simpler in a closed network that can police all incoming traffic at the network boundary. For example, a simple priority queueing discipline can provide a premium service if the amount of incoming premium traffic is limited to a small fraction of the capacity. On the other hand, most current IP networks do not follow such a closed network model so that no firm assumption can be made about incoming traffic.

8.2 Discussion

One of our goals is to promote the widespread use of UNIX-based routers. Traffic management is becoming increasingly important, especially at network boundaries that are points of congestion. Technical innovations are required to provide smoother and more predictable network behavior. In order to develop intelligent routers for the next generation, a flexible and open software development environment is most important. We believe UNIX-based systems, once again, will play a vital role in the advancement of technologies.

However, PC-UNIX based routers are unlikely to replace all router products in the market. Non-technical users will not use PC-UNIX based routers. High-speed routers or highly-reliable routers require special hardware and will not be replaced by PCs. Still, the advantage of PC-UNIX based routers is their flexibility and availability in source form, just like the advantage of UNIX over other operating systems. We argue that we should not let black boxes replace our routers and risk loosing our research and development environment. We should instead do our best to provide high-quality routers based on open technologies.

There are still many things that need to be worked out for the widespread use of PC-UNIX based routers. Network operators may worry about the reliability of PC-based routers. However, a reliable PC-based router can be built if the components are carefully selected. In most cases, problems are disk related troubles and it is possible to run UNIX without a hard disk by using a non-mechanical storage device such as ATA flash cards. Another reliability issue lies in PC components (e.g., cooling fans) that may not be selected to be used 24 hours a day. There are a wide range of PC components and the requirements for a router are quite different from those for a desktop business machine. Some of the rack-mount PCs on the market are suitable for routers but SOHO routers need smaller chassis. We need PC hardware packages targeted for router use.

By the same token, we need software packages. It is not an easy task to configure a kernel to have the necessary modules for a router and make it run without a hard disk or a video card. Although there are many tools for routers, compilation, configuration, and maintenance of the tools are time consuming. Freely-available network administration tools seem to be weak but could be improved as PC-based routers become popular.

In summary, the technologies required to build quality PC-UNIX based routers are already available, but we need better packaging for both hardware and software. The networking research community would benefit a great deal if a line of PC-based router packages were available for specific scenarios, such as dial-up router, boundary router, workgroup router, etc.

Chapter 9

Related Work

Our idea of providing a framework for queueing disciplines is not new. Nonetheless there have been few efforts to support a generic queueing framework. Research queueing implementations in the past have customized kernels for their disciplines [GF95, SZ97]. However, they are not generalized for use of other queueing disciplines. ALTQ took an initiative in this area, and motivated other research efforts [DDPP98, Alm99, MKJK99].

ALTQ implements a switch to a set of queueing disciplines, which is similar to the protocol switch structure of BSD UNIX. A different approach is to use a modular protocol interface to implement a queueing discipline. STREAMS [Rit84] and x-kernel [PHOR90] are such frameworks and the CBQ release for Solaris is actually implemented as a STREAMS module. Although it is technically possible to implement a queueing discipline as a protocol module, a queueing discipline is not a protocol and the requirements are quite different. It is more flexible to abstract the whole QoS block than to abstract each QoS component since various types of QoS components can be combined with a variety of connections. One of the contributions of this paper is to have identified the requirements of a generic queueing framework.

A large amount of literature exists in the area of process scheduling but we are not concerned with process scheduling issues. Routers, as opposed to end hosts which run real-time applications, do not need real-time process scheduling because packet forwarding is part of interrupt processing. For end hosts, process scheduling is complementary to packet scheduling.

9.1 Dummynet

Dummynet [Riz97] is another popular mechanism available for FreeBSD to limit bandwidth. However, dummynet is an extension of a firewall mechanism, and thus, it does not address the system architecture design to support QoS.

Dummynet is originally designed to emulate a link with varying bandwidth and delay, and realized as a set of 2-level shapers; the first level shaper enforces the bandwidth limit, and the second level shaper enforces the specified delay.

Dummynet has several advantages over ALTQ. Dummynet is implemented solely in the IP layer so that it is device independent and no modification is necessary to drivers. Because dummynet is a set of software shapers, dummynet can be used both on the input path and on the output path. In addition, the classifier of dummynet is integrated into *ipfw* (the firewall mechanism of FreeBSD) so that it can be configured as part of firewall rules. Dummynet also works with the Ethernet bridging mechanism.

On the other hand, there are disadvantages. The shaper mechanism is realized solely by the kernel timer so that the shaper resolution is limited to the kernel timer resolution. Although ALTQ shares the same limitation, ALTQ can take advantage of transmission complete interrupts. Dummynet does not work with the *fastforwarding* mechanism that bypasses the normal IP forwarding path.

In summary, dummynet is good for simple bandwidth limiting on moderate (Ethernet class) link speed, and it is easy to configure. There are great demands for bandwidth control that fall into this category.

9.2 Linux Traffic Control

Linux has a traffic control (TC) framework since version 2.1 [Alm99]. The implemented queueing disciplines include CSZ, PQ, CBQ, RED and SFQ.

Linux TC is similar to ALTQ in a number of ways. The Linux TC framework has a switch of queueing disciplines and defines a set of queue operations. One difference found in the queue operations is that Linux TC defines “requeue” (prepend) instead of “poll”. Linux TC employs a dequeue-and-requeue policy while ALTQ employs a poll-and-dequeue policy. The problem of dequeue-and-requeue is that it is hard for some disciplines to cancel a decision once the internal state is updated. Some of Linux TC disciplines just store a requeued packet and provide this packet for the next dequeue without running the scheduler, even though it may not be a right packet to dequeue at this time.

Although the architecture or the system design of Linux TC is not documented, it seems to try to abstract a packet scheduler and the interface to the kernel becomes complicated. On the other hand, ALTQ hides the existence of a packet scheduler within a simple output queue blackbox model. ALTQ also emphasizes backward-compatibility and IPv6 support.

Other architectural differences come from the kernel architecture. That is, Linux has a network device layer and its *sk_buff* has rich fields.

Linux has a common network device layer that handles link type specific processing and acts as an upper half of a driver. Queueing is done within this device layer so that TC requires changes only in this layer.

In BSD UNIX, there is no common code path between the network layer and network device drivers. Operations are performed only through *struct ifnet*. As a result, the ALTQ support is scattered in *if_output* and *if_start*. Note that it is not only ALTQ but also BPF needs supporting code in device drivers. In Linux, they are also supported in the common network device layer.

Linux's *sk_buff* has many fields and have almost all information about a packet. A classifier can easily access network layer or transport layer information. On the other hand, *mbuf* of BSD UNIX carries no information about a packet. Though this design is good for enforcing network stack layering, a classifier needs to extract information from a packet itself by parsing headers.

These architectural differences illustrate the difference in their design philosophy. The network code of BSD UNIX has been successful with this minimalist approach. BSD UNIX is written in such a way that implemented functions are carefully narrowed down. Thus, extending the existing code is harder and requires a careful design, which in turn leads to a careful design of the extension itself.

However, BSD UNIX might need to redesign the current abstraction in the future. An abstracted network device will make extensions easier and keep drivers simpler. There are other possible extensions to the interface level such as sub-interfaces for VLAN and virtual interfaces for multi-link. Also, various optimizations will be possible if packet information can be tagged to *mbuf*.

9.3 Summary

ALTQ was one of the first efforts trying to provide a generic queueing framework. In this section, ALTQ is compared with dummynet and Linux Traffic Control. The design of ALTQ is based on the proposed system architecture to support QoS, and it is carefully implemented into BSD UNIX. ALTQ also emphasizes backward-compatibility and IPv6 support.

Chapter 10

Conclusion

In this thesis, we have presented the design and implementation of the ALTQ traffic management system. ALTQ was designed to meet the needs of new network applications that require quality-of-service.

ALTQ started as a small project to prove our QoS framework concept but has evolved into a full-fledged QoS package; a wide range of QoS features have been added since its first public release in March 1997.

The current version of ALTQ runs on several versions of FreeBSD, NetBSD and OpenBSD. A large number of network device drivers are supported. H-FSC, CBQ, RED (including ECN), RIO, WFQ, BLUE, FIFOQ and DiffServ traffic conditioners are implemented.

ALTQ is integrated into the KAME IPv6 stack and fully supports IPv6 as well as IPv4. ALTQ is now being developed under the KAME CVS repository so that new features and bug fixes are committed first to the KAME repository, and an ALTQ release is created out of the KAME respository. Also, ALTQ has been integrated into the development tree of NetBSD so that ALTQ will be part of the standard NetBSD release.

Many people are using ALTQ as a research platform or a testbed. Although we do not know how many ALTQ users are out there, our ftp server has recorded thousands of downloads over the last 4 years.

The performance of our implementation is quite satisfactory, but there are still many things to be worked out such as scalability issues in implementation and easier configuration for users.

10.1 Contributions

We have proposed a QoS system architecture that consists of QoS system framework, QoS forwarding mechanisms, and QoS management mechanisms. The architecture design conceals details and differences of QoS components within larger QoS blocks placed on input and output interfaces. ALTQ is a practical implementation model of this architecture, and fits well into the existing networking stack.

The system framework of ALTQ presents flexible abstractions to support a wide variety of queueing disciplines, and provides the interface between QoS components and the underlying operating system. We have identified the problems in the current abstraction of output queueing, and then, introduced a new abstraction of output queueing in order to provide service differentiation by packet scheduling and buffer management. The flexibility of the framework design is proved through implementing a number of QoS components onto the framework. The proposed framework keeps backward-compatibility not to break hundreds of the existing drivers so that incremental transition to the new model is possible.

QoS forwarding mechanisms built on top of the framework perform actual QoS functions such as packet scheduling, buffer management, classifier, and traffic conditioning. Since QoS components are integrated into the ALTQ framework, it becomes possible to combine components that were originally proposed and implemented independently. It also becomes possible to examine issues in implementing QoS mechanisms proposed only through theory or simulation. We have identified limitations and issues in implementing QoS components such as effects of device buffer and timer granularity. The performance of the implemented forwarding mechanisms has proved that theoretical work can be applied to the real-world.

QoS management mechanisms are a set of tools and libraries used for traffic management, and control QoS forwarding mechanisms implemented in the kernel. In the early stage of the ALTQ deployment, they were simple development tools. As the ALTQ users grow in size and diversity, the importance of the management tools is increasing. QoS management mechanisms also include important future research themes such as policy servers and QoS monitoring.

We have evaluated the performance of the system including the framework and QoS forwarding mechanisms since the performance of the ALTQ system is the combined performance of the framework and QoS forwarding mechanisms to use. We have confirmed that the ALTQ system is able to provide various types of QoS with a minimal overhead.

ALTQ has been a flexible and well-engineered platform for QoS related research. ALTQ allows researchers to easily implement new queueing disciplines without knowing the details of kernel programming. ALTQ provides missing components to developers of QoS based systems that assume traffic control in the underlying platform.

It is essential to the Internet research to obtain feedback from field experiences. ALTQ provides a set of tools to gain operational experiences.

Many research projects have been using ALTQ, which shows that ALTQ has stimulated many other research activities in the field.

10.2 Final Remarks

We have identified the requirements of a generic QoS framework and the issues of implementation. Then, we have demonstrated, with a number of forwarding mechanisms, that simple extension to BSD UNIX and minor fixes to drivers are enough to incorporate a variety of QoS mechanisms. The main contribution of ALTQ is engineering efforts to make quality-of-service available for researchers and network operators on commodity PC platforms and UNIX. Our performance measurements clearly show the feasibility of traffic management by PC-UNIX based routers.

As router products have been proliferating over the last decade, network researchers have been losing research testbeds available in source form. We argue that general purpose computers, especially PC-based UNIX systems, have become once again competitive router platforms because of flexibility and cost/performance. Traffic management issues require technical innovations, and the key to progress is platforms which new ideas could be easily adopted into. ALTQ is an important step in that direction.

10.3 Future Research

ALTQ presents a major step forward from the current networking systems. Although much has been accomplished, there are still plenty of tasks that need attention. Future research could be done in the following areas:

- **Auto-configuration and dynamic management:**
  Configuring QoS components even for a single node is not easy. It requires careful design of a network, understanding of the underlying mechanisms, and appropriate selection of parameters. A subtle mistake in the setting could have a significant effect on traffic. As

the size of a network grows, it soon becomes impossible to manually configure QoS components. An automatic configuration process, or a self-managing agent is vital to widespread use of traffic management.

- **Monitoring and measurements:**
  Quality of service involves monitoring the quality. It is necessary to monitor a set of quality metrics. To meet the requirements of network operations, a diverse set of monitoring and measurement tools are needed. Again, automation is vital to widespread use of traffic management.

- **Policy Management:**
  In a self-managing QoS-capable network, admission control is required. Admission control is normally based on a set of policies. A number of constrains are placed on allowing a higher level of service to a specific user. A policy-based admission control architecture as well as a policy exchange protocol are being studied at IETF. However, there is much to be developed and solved in this field.

The ALTQ system is designed to be flexible for additional enhancements, and will be served as a platform for research in these areas. It is our challenge to allow users to take full advantage of the benefits of quality-of-service support in the Internet.

Bibliography

[Alm99] W. Almesberger. Linux network traffic control – implementation overview. In *Proceedings of 5th Annual Linux Expo*, pages 153–164, Raleigh, NC, May 1999.

[BBC$^+$98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, Internet Engineering Task Force, December 1998.

[BCC$^+$98] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, K. L. Peterson, S. Shenker Ramakrishnan, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, Internet Engineering Task Force, April 1998.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. RFC 1633, Internet Engineering Task Force, June 1994.

[BGP$^+$94] Mary L. Baily, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkan. Pathfinder: A pattern-based packet classifier. In *Proceedings of Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, November 1994.

[BSB99] Y. Bernet, A. Smith, and S. Blake. A conceptual model for diffserv routers. Internet Draft, Internet Engineering Task Force, October 1999.

[CF98] D. Clark and W. Fang. Explicit allocation of best effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

[Cho98] Kenjiro Cho. A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers. In *USENIX 1998 Annual Technical Conference*, pages 247–258, June 1998.

[Cho99] Kenjiro Cho. Flow-valve: Embedding a safety-valve in red. In *Global Internet Symposium, Globecom*, December 1999.

[Cob54] A. Cobham. Priority assignment in waiting line problems. *Operations Research*, 2:70–76, 1954.

[Cru92] R. Cruz. Service burstiness and dynamic burstiness measures: A framework. *Journal of High Speed Networks*, 1(2):105–127, 1992.

[Cru95] R. Cruz. Quality of service guaranteed in virtual circuit switched network. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, August 1995.

[DDPP98] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins - a software architecture for next generation routers. In *Proceedings of SIGCOMM '98 Symposium*, pages 229–240, Vancouver, British Columbia, Canada, September 1998.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM '89 Symposium*, pages 1–12, Austin, Texas, September 1989.

[FF99] Sally Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transaction on Networking*, 7(4):458–472, August 1999.

[FJ91] S. Floyd and V. Jacobson. Traffic phase effects in packet-switched gateways. *Computer Comunication Review*, 21(2):26–42, April 1991.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–413, August 1993. Also available from `http://www-nrg.ee.lbl.gov/floyd/papers.html`.

[FJ95] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995. Also available from `http://www-nrg.ee.lbl.gov/floyd/papers.html`.

[Flo94] Sally Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5), October 1994. Also available from `http://www-nrg.ee.lbl.gov/floyd/papers.html`.

[FS99] Wenjia Fang and Nabil Seddigh. A time sliding window three colour marker (tswtcm). Internet Draft, Internet Engineering Task Force, October 1999.

[GCY99] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. In *USENIX 1999 Annual Technical Conference: FREENIX Track*, June 1999.

[GF95] Amit Gupta and Domenico Ferrari. Resource partitioning for real-time communication. *IEEE/ACM Transactions on Networking*, 3(5), October 1995. Also available from `http://tenet.berkeley.edu/tenet-papers.html`.

[GM99] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of SIGCOMM '99 Symposium*, page 8, Harvard University, September 1999.

[HBWW99] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding phb group. RFC 2597, Internet Engineering Task Force, June 1999.

[HG99] J. Heinanen and R. Guerin. A two rate three color marker. RFC 2698, Internet Engineering Task Force, September 1999.

[ISI] The RSVP Project at ISI. `http://www.isi.edu/rsvp/`.

[Jac88] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review*, 18(4):314–329, August 1988.

[Jac90] V. Jacobson. Berkeley tcp evolution from 4.3-tahoe to 4.3-reno. In *the Eighteenth Internet Engineering Task Force*, Vancouver, Canada, September 1990.

[JNP99] V. Jacobson, K. Nichols, and K. Poduri. An expedited forwarding phb. RFC 2598, Internet Engineering Task Force, June 1999.

[Jon93] Rick Jones. *Netperf: A Benchmark for Measuring Network Performance*. Hewlett-Packard Company, 1993. Available at `http://www.cup.hp.com/netperf/NetperfPage.html`.

[Kes91] Srinivasan Keshav. On the efficient implementation of fair queueing. *Internetworking: Research and Experience*, 2:157–173, September 1991.

[KKK90] C. R. Kalmanek, H. Kanakia, and Srinivasan Keshav. Rate controlled servers for very high-speed networks. In *Proceedings of Globecom*, pages 12–20 (paper 300.3), San Diego, California, December 1990. IEEE.

[Kle75] Leonard Kleinrock. *Queueing Systems — Theory*, volume 1. Wiley-Interscience, New York, New York, 1975.

[Kle76] Leonard Kleinrock. *Queueing Systems — Computer Applications*, volume 2. Wiley-Interscience, New York, New York, 1976.

[KSS01] Martin Karsten, Jens Schmitt, and Ralf Steinmetz. Implementation and Evaluation of the KOM RSVP Engine. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'2001)*. IEEE, April 2001.

[LNO96] T. V. Lakshman, A. Neidhardt, and T. J. Ott. The drop from front strategy in tcp and in tcp over atm. In *Proceedings of INFOCOM*, 1996.

[LTWW93] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *Proceedings of SIGCOMM '93 Symposium*, pages 183–193, San Francisco, California, September 1993. ACM. Also available as `ftp://thumper.bellcore.com/pub/wel/sigcomm93.ps.Z`.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the*

4.4 BSD Operating System. Addison-Wesley Publishing Co., 1996.

[McK90] P. E. McKenney. Stochastic fairness queueing. In *Proceedings of INFOCOM*, San Francisco, California, June 1990.

[MKJK99] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click moduler router. In *Proceedings of SOSP'99*, pages 217–231, Kiawah Island Resort, SC, December 1999.

[Nag87] John Nagle. On packet switches with infinite storage. *IEEE Trans. on Comm.*, 35(4), April 1987.

[Par92] Abhay Parekh. A generalized processor sharing approach to flow control in integrated services networks. LIDS-TH 2089, MIT, February 1992.

[PFTK98] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: A simple model and its empirical validation. In *Proceedings of SIGCOMM '98 Symposium*, pages 303–314, Vancouver, Canada, September 1998.

[PHOR90] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. The x-kernel: A platform for accessing internet resources. *Computer*, 23(5):23–34, May 1990.

[RG99] F. Risso and P. Gevros. Operational and performance issues of a cbq router. *ACM Computer Communication Review*, 29(5):47–58, October 1999.

[Rij94] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.

[Rit84] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[Riz97] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *Computer Comunication Review*, 27(1):31–41, April 1997. Also available from `http://www.iet.unipi.it/~luigi/`.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. RFC 2212, Internet Engineering Task Force, September 1997.

[SS95] McCanne S. and Floyd S. NS (Network Simulator). `http://www-nrg.ee.lbl.gov/ns/`, 1995.

[SVSW98] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of SIGCOMM '98 Symposium*, pages 191–202, Vancouver, British Columbia, Canada, September 1998.

[SZ97] Ion Stoica and Hui Zhang. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proceedings of SIGCOMM '97 Symposium*, pages 249–262, Cannes, France, September 1997.

[WGC+95] Ian Wakeman, Atanu Ghosh, Jon Crowcroft, Van Jacobson, and Sally Floyd. Implementing real-time packet forwarding policies using streams. In *Proceedings of USENIX '95*, pages 71–82, New Orleans, Louisiana, January 1995.

[Wro97] J. Wroclawski. The use of rsvp with ietf integrated services. RFC 2210, Internet Engineering Task Force, September 1997.

[ZDE+93] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7:8–18, September 1993. Also available from `http://www.isi.edu/rsvp/pub.html`.

[ZK91] Hui Zhang and Srinivasan Keshav. Comparison of rate-based service disciplines. In *Proceedings of SIGCOMM '91 Symposium*, pages 113–121, Zurich, Switzerland, September 1991.

Appendix A

List of Research Projects using ALTQ

ALTQ is distributed under a BSD-style license, and thus, is freely available. A public release of ALTQ, the source code along with additional information, can be found at `http://www.csl.sony.co.jp/~kjc/software.html`

Since its first public release in March 1997, ALTQ has been used for a large number of research projects around the world. The following list contains research projects using ALTQ.

1. Hui Zhang's group at CMU: the H-FSC implementation in ALTQ is a result of collaboration with Hui Zhang's group.
   `http://www.cs.cmu.edu/~hzhang/`

2. Darwin project at CMU: The Darwin (Resource Management for Application-Aware Networks) project uses ALTQ as a traffic control platform.
   `http://www.cs.cmu.edu/~darwin/`

3. CAIRN Project: The CAIRN project builds a large scale research testbed using PC-based routers. ALTQ has been used by CAIRN as one of platforms
   `http://www.cairn.net/`

4. KOM RSVP: RSVP implementation by Martin Karsten. KOM RSVP is a RSVP implementation at Darmstadt University of Technology, Germany. It uses ALTQ as a traffic control module [KSS01].
   `http://www.kom.e-technik.tu-darmstadt.de/rsvp/`

5. ISI RSVP: The RSVP group at ISI uses ALTQ as one of traffic control modules.
   `http://www.isi.edu/rsvp/`

6. KAME IPv6: ALTQ is integrated into the IPv6 implementation by the KAME Project. ALTQ is being developed in the KAME development tree.
   `http://www.kame.net/`

7. INRIA IPv6: ALTQ is integrated into the IPv6 implementation by INRIA.
   `ftp://ftp.inria.fr/network/ipv6/`

8. WIDE Project: ALTQ is used for diffserv within the WIDE Project.
   `http://www.wide.ad.jp/`

9. Moon Bear Project: Moon Bear Project is a joint project of three universities in Japan, and working on QoS policy management, QoS routing, and traffic measurement. ALTQ is used as a platform.
   `http://www.moon-bear.net/`

10. Ayame Project at Japan Advanced institute of Science and Technology: Ayame Project is working on a network architecture based on MPLS. ALTQ is used as a platform.
    `http://www.ayame.org/`

11. RT-HDI (Real-Time Human Device Interaction) Project at Keio University: ALTQ is used as a network QoS platform.
    `http://www.mkg.sfc.keio.ac.jp`

12. ECN research at UCLA: Hariharan Krishnan and Lixia Zhang use ALTQ for their research on ECN.
    `http://www.cs.ucla.edu/~hari/software/ecn/ecn.html`

13. RSVP tunnels at UCLA: Andreas Terzis and Lixia Zhang use ALTQ for their research on RSVP tunnels.
    `http://www.cs.ucla.edu/~terzis/`

14. Two Tier Architecture for Resource Allocation at UCLA: Lixia Zhang's group is working on a Bandwidth-Broker implementation on top of ALTQ.
    `http://irl.cs.ucla.edu/twotier`

15. NASA Goddard Space Flight Center: various QoS tests by George Uhl.
`http://corn.eos.nasa.gov/notebooks/ip-0007.html`
`http://www.nren.nasa.gov/CFP/uhl.html`

16. HICID at UCL: The High Performance Interactive Conferencing and Information Distribution Project at University College London. Risso and Gevros evaluated our CBQ implementation in ALTQ in [RG99].
`http://www.cs.ucl.ac.uk/staff/jon/arpa/altq2/fulvio-cbq-2.html`
`http://www.cs.ucl.ac.uk/staff/jon/hicid/hicid.html`

17. Mobile IP Project at NUS: Mobile IP Project at NUS ported ALTQ onto Linux to support RSVP.
`http://mip.ee.nus.edu.sg/`

18. Course Projects at Harvard: ALTQ is used for a course projects at Harvard.
`http://www.eecs.harvard.edu/cs96/`

19. Blue at U. Michigan, W. Feng, D. Kandlur, D. Saha, K. Shin
ALTQ was used to implement BLUE. `http://www.eecs.umich.edu/~wuchang/blue/`

20. UNC DiRT: The Distributed real-time system group at the University of North Carolina at Chapel Hill uses ALTQ as a research platform.
`http://www.cs.unc.edu/~jeffay/dirt/`

21. NIST Switch Project: NIST switch project builds multilayer label switching nodes useing ALTQ.
`http://snad.ncsl.nist.gov/itg/nistswitch/description/nistswitch.html` y

22. Universita' di Napoli Federico II: Simon Pietro Romano and Raffaele D'Albenzio use ALTQ for their QoS research platform.
`http://www.grid.unina.it/individual/sprom/`

23. adserv: differentiated service implementation by Octavio Medina.
`ftp://ftp.rennes.enst-bretagne.fr/pub/reseau/medina/adserv-0.1.tar.gz`