Managing Traffic with ALTQ

Kenjiro Cho

Sony Computer Science Laboratories, Inc.
Tokyo, Japan 1410022
kjc@csl.sony.co.jp

Abstract

ALTQ is a package for traffic management. ALTQ includes a queueing framework and several advanced queueing disciplines such as CBQ, RED, WFQ and RIO. ALTQ also supports RSVP and diffserv. ALTQ can be configured in a variety of ways for both research and operation. However, it requires understanding of the technologies to set up things correctly. In this paper, I summarize the design trade-offs, the available technologies and their limitations, and how they can be applied to typical network settings.

1 Queueing Basics

Essentially, every traffic management scheme involves queue management. A large number of queueing disciplines have been proposed to date in order to meet contradictory requirements such as fairness, protection, performance bounds, ease of implementation or administration.

1.1 Queueing Components

Figure 1 illustrates queueing related functional blocks on a router. Each functional block could be needed to build a certain service but is not always required for other services. In fact, most routers currently in use do not have all the functional blocks.

Packets arrive at one interface of the router (ingress interface), and then, are forwarded to another interface (egress interface). A router could have functional blocks in the ingress interface to police incoming packets but the main functional blocks reside in the egress interface. The function of each block is described below.

Classifier Packet classifiers categorizes packets based on the content of some portion of the packet header. (e.g., addresses and port numbers). Packets matching some specified rule are classified for further processing.

Meters Traffic meters measure the properties of a traffic stream (e.g., bandwidth, packet counts). The measured characteristics are stored as flow state and used by other functions.

Markers Packet markers set a particular value to some portion of the packet header. The written values could be a priority, congestion information, an application type, or other types of information.

Droppers Droppers discard some or all of the packets in a traffic stream in order to limit the queue length, or as an implicit congestion notification.

Queues Queues are finite buffers to store backlogged packets. A queueing discipline could have multiple queues for different traffic classes.

Schedulers Schedulers select a packet to transmit from the backlogged packets in the queue.

Shapers Shapers delay some or all of the packets in a traffic stream in order to limit the peak rate of the stream. A shaper usually has a finite-size buffer, and packets may be discarded if there is not sufficient buffer space to hold the delayed packets.

A queueing discipline is, in general, defined as a set of the functional blocks at the egress interface, and usually consists of a specific queue structure, a scheduling mechanism and a dropper mechanism. However, the functional blocks described here are conceptual and a wide variety of combinations are possible.

1.2 Queueing Disciplines

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

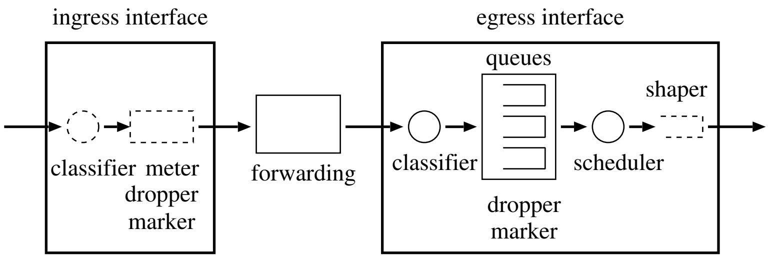

Figure 1: Queueing Architecture

Another important goal 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. A leaky bucket is a simple shaper mechanism with a finite buffer size. Another popular shaper mechanism is a token bucket that allows small bursts with a configurable burst size. The token bucket can accommodate traffic streams with bursty characteristics so it is more suitable for the current Internet traffic.

Yet another goal of a queueing discipline is congestion avoidance. TCP considers packet loss as a sign of congestion. A router can notify TCP of congestion implicitly by intentionally dropping a packet.

The following list describes representative queueing disciplines.

FIFO The simplest possible queueing discipline is FIFO (First-In-First-Out) that has only a single queue and a simple drop-tail dropper.

PQ PQ (Priority Queueing) has multiple queues associated with different priorities. A queue with a higher priority is always served first. Priority queueing is the simplest form of preferential queueing. However, low priority traffic easily starves unless there is a mechanism to regulate high priority traffic.

WFQ WFQ (Weighted Fair Queueing) [11, 4, 8] is a discipline that assigns an independent queue for each flow. WFQ can provide fair bandwidth allocation in times of congestion, and protects a flow from other flows. A weight can be assigned to each queue to give a different proportion of the network capacity.

SFQ SFQ (Stochastic Fairness Queueing) [10] is an approximation of WFQ. WFQ is difficult to implement because a large number of queues are required as the number of flows increases. In SFQ, a hash function is used to map a flow to one of a fixed set of queues, and thus, it is possible for two different flows to be mapped into the same queue.

CBQ CBQ (Class Based Queueing) [7] 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. CBQ is non-work conserving and can regulate bandwidth use of a class. A child class can be configured to borrow bandwidth from its parent class as long as excess bandwidth is available.

RED RED (Random Early Detection) [6, 2] is a dropper mechanism that exercises packet dropping stochastically according to the average queue length. RED avoids traffic synchronization in which many TCPs lose packets at one time [5]. Also, RED makes TCPs keep the queue length short. RED is fair in the sense that packets are dropped from flows with a probability proportional to their buffer occupation. Since RED does not require per-flow state, it is considered scalable and suitable for backbone routers.

1.3 Issues in Queueing

Although there are a large number of mechanisms available for traffic management, there is no single mechanism that satisfies a wide range of requirements. Therefore, it is important to use appropriate mechanisms suitable for a purpose.

In addition, a mechanism can have quite different effects depending on how it is used. For example, WFQ for best effort traffic can provide fair bandwidth allocation. A certain portion of the link capacity can be reserved by configuring the weight and the classifier of WFQ. Further, the delay can be bounded by adding a token bucket to the traffic source.

Furthermore, it is not easy to combine different mechanisms in a coherent manner because different mechanisms are independently developed to meet the requirements of specific applications.

The issues in designing queueing disciplines are described below.

Overhead Most of the functional blocks are located in the packet forwarding path, and thus, adds some overhead to forwarding performance. A queueing discipline should require only a few simple operations to forward a packet in order to scale to a high-speed network. It is also preferable to be easily implemented in hardware.

Flow Definition A flow is a unit that a classifier identifies packets in a traffic stream. A flow can be a micro flow such as a single TCP session, or some type of aggregated flow.

Packets belonging to the same micro flow should be placed in the same queue in order to avoid packet reordering. Although TCP and other transport mechanisms can handle out-of-order packets, frequent out-of-order packets will considerably damage the transport performance.

Classifier Design The design of an efficient classifier is still an area of active research. A scalable algorithm is required as the number of filters or the number of active flows increases. Efficient handling of wild card filters is difficult because it needs to find a best match for multiple fields.

Classifiers are required not only by queueing but also by firewall, layer 4 forwarding, and traffic monitoring. Classifiers should be designed to be shared by other components.

To identify traffic types by port numbers, a classifier needs to check the transport header (e.g., TCP, UDP). However, port based classification is not always possible if a packet is fragmented or encrypted. Although IP fragments will decrease by the Path MTU Discovery, encrypted packets will be common with the widespread use of secure shells and IPsec.

Required States A queueing discipline needs to keep some state for each traffic class. The size of a state and the total number of states have a great impact to the scalability of a queueing discipline. It is believed that per-flow queueing is preferable for a small network or at an edge of a backbone but only aggregated-flow queueing is possible within a backbone network.

A related issue is how long a flow state is maintained. A queueing discipline could keep only the states of flows that have packets in the queue. On the other hand, a discipline would need information for a longer period to enforce a longer term rule.

Fairness Fairness is an important property to handle best effort traffic. However, there are different definitions of fairness and different targets for whom fairness is defined. Local fairness at a router does not necessarily lead to global fairness. Besides, network traffic is dynamic and constantly changing so that static fairness does not necessarily lead to fairness in a larger time scale.

Non-work Conserving Queues A work-conserving discipline is idle only when there is no packet awaiting service. A non-work conserving discipline, on the other hand, can delay packets in the queue; it can be considered as a form of a shaper. A non-work conserving queue is more complex to implement but is able to limit the peak rate, reduce jitters, and provide tighter performance bounds.

Statistical Guarantee Performance guarantee can be either deterministic or statistic. In general, deterministic guarantee requires a much larger fraction of the resources to be reserved than statistical guarantee. In practice, deterministic guarantee is difficult to implement because computer communication involves many mechanisms that do not have tight bounds.

2 Traffic Management

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.

2.1 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.

2.2 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.

2.3 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.

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.

Table 1: Queueing Overhead Comparison

	FIFO	FIFOQ	RED	WFQ	CBQ	CBQ +RED
(usec)	0.0	0.14	1.62	1.95	10.72	11.97

2.4 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 [3].

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.

2.5 Building Services

So far, we have looked at the behavior of a single router. An end-to-end service quality can be obtained by con-

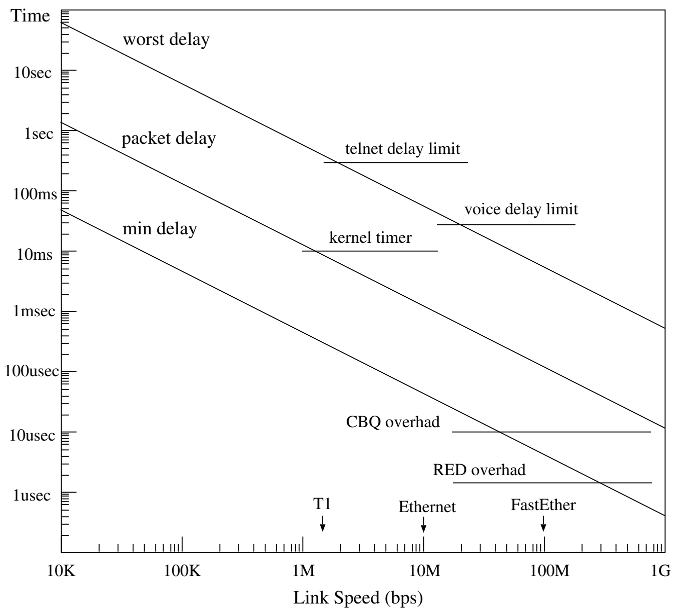

Figure 2: Queueing and Link Speed

catenating 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 [1]. 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.

3 ALTQ

ALTQ [3] is a framework for FreeBSD that introduces a variety of queueing disciplines. ALTQ provides a platform for traffic management related research. ALTQ also makes active queue management available for operational experience.

3.1 Design

The basic design of ALTQ is quite simple; the queueing interface is designed as a switch to a set of queueing disciplines as shown in Figure 3. To implement ALTQ, several fields are added to *struct ifnet*. The added fields are a discipline type, a common state field, a pointer to a discipline specific state, and pointers to discipline specific enqueue/dequeue functions.

The implementation policy of ALTQ is to make minimal changes to the existing code. The current kernel, however, does not have queueing abstraction enough to implement various types of queueing disciplines. As a result, there are many parts of the kernel code that assume FIFO queueing and the *ifqueue* structure.

Especially, it is problematic that many drivers directly use the *ifqueue* structure, *if_snd*, in the *ifnet* structure. These drivers must be modified but it is not easy to modify all the existing drivers. Therefore, we took an approach that allows both modified drivers and unmodified drivers to coexist so that we can modify only the drivers we need, and incrementally add supported drivers.

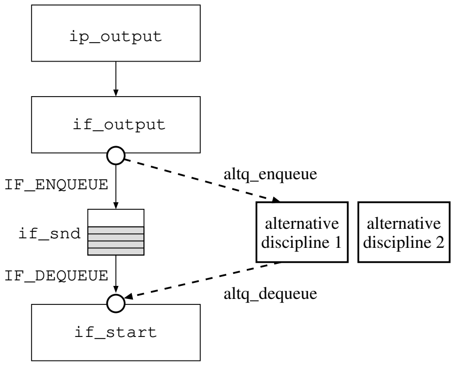

Figure 3: Alternate Queueing Architecture

3.1.1 Queueing Operations

In ALTQ, queueing disciplines have a common set of queue operations. Other parts of the kernel code manipulate a queue through 4 queueing operations; enqueue, dequeue, peek and flush. Drivers are modified to use only these operations, and not to refer to the *ifqueue* structure.

The enqueue operation is responsible not only for queueing a packet but also for other functions such as classifier and dropper that are required to enqueue a packet.

The dequeue operation returns the next packet to send. The main role of the dequeue operation is packet scheduling.

The peek operation is similar to the dequeue operation but it does not remove the packet from the queue. The peek operation can be used by a driver to see if there is enough buffer space or DMA descriptors for the next packet. ALTQ does not have a prepend operation since prepending a packet does not make sense if a discipline has multiple queues. Therefore, a driver should use a peek-and-dequeue policy if necessary.

The flush operation is used to empty the queue since non-work conserving queues cannot be emptied by a dequeue-loop.

3.1.2 Discipline Operations

Queueing disciplines are controlled by *ioctl* system calls via a queueing device (e.g., */dev/cbq*). ALTQ is defined as a character device and each queueing discipline is defined as a minor device of ALTQ.

There are 4 common operations to handle queueing disciplines; attach, detach, enable, and disable. The attach operation sets a queueing discipline to the specified interface. An interface can have one queueing discipline attached at a time. The attached discipline is not activated until the enable operation is performed. When the alternative queueing is disabled or closed, the system falls back to the original FIFO queueing.

Other than these operations, each queueing discipline usually needs discipline specific settings, which are also done via discipline specific *ioctls*.

3.2 Using ALTQ

ALTQ implements several queueing disciplines including CBQ, WFQ, RED, ECN, and RIO. For managing an operational network, CBQ will be the most appropriate discipline. CBQ is flexible to meet a wide range of requirements and the implementation is stable and well tested. Moreover, the CBQ implementation also integrates RED so that RED can be enabled for each CBQ class. The detailed mechanism of CBQ can be found elsewhere [7, 12, 3].

There are implementation issues when using ALTQ for different link speeds. These issues are described below.

3.2.1 Effect of Timer Resolution

Shapers are usually realized using timers, and thus, the resolution of a shaper is limited by the kernel timer resolution. The kernel timer resolution is 10msec in most UNIX systems. This implies that traffic becomes bursty if the packet transmission time of the link is less than the kernel timer resolution. On 10Mbps Ethernet, a 1500 byte packet takes 1.2msec so that 8 packets can be sent during a timer interval. 100Mbps FastEthernet is problematic since more than 80 packets can be sent during a timer interval. Therefore, it is desirable to use a higher resolution for the kernel timer. Current PCs seem to have little overhead even if the timer resolution is increased by a factor of 10.

CBQ employs a more elaborate scheme to limit bandwidth but it also has constraints from the timer resolution. In CBQ, an overlimit class is suspended until the state becomes underlimit again. A suspended class can be resumed from transmission complete interrupts but it relies on a timeout in case that the class is not resumed from interrupts. CBQ also limits the number of back-to-back packets by a variable *maxburst*. In the worst case scenario in which resuming is done only from timeouts, bandwidth of a class is limited by the timer resolution and *maxburst*. The default value of *maxburst* is 16; the value is selected to achieve 9.6Mbps on Ethernet with the default timer resolution. However, it is only 1/10 of the link capacity for FastEthernet. It is desirable to use a higher timer resolution for FastEthernet; 1 msec resolution achieves 96Mbps.

Note that the timer resolution affects only non-work conserving disciplines. Work-conserving disciplines do not need timers since packets are sent from transmission complete interrupts.

3.2.2 Difference in Network Cards

Some cards generate interrupts every time a packet is transmitted, and some generate interrupts only when the buffer becomes empty. It is generally believed that a smart network card should reduce interrupts to alleviate CPU burden. However, a queueing discipline could have finer grained control with frequent interrupts; it is a trade-off between CPU control and CPU load. There is an interesting report that CBQ works much better with an old NE2000 card that interrupts a lot and has small buffers.

3.2.3 Device Buffers

There is a similar trade-off in setting the buffer size in a network card. When delay is a concern on a slow link, large buffers in network cards could adversely affect queueing. For example, if a network card for a 128Kbps link has a 16KB buffer, the buffer can hold 1 second worth of packets. The device buffer has an effect of inserting another FIFO queue beneath a queueing discipline. This problem is invisible under FIFO but it 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, it seems that many drivers set an excessive buffer size.

3.3 Availability

A public release of ALTQ for FreeBSD, the source code along with additional information, can be found at http://www.csl.sony.co.jp/person/kjc/software.html.

4 Related Work

4.1 Dummynet

Dummynet [9] is another popular mechanism available for FreeBSD to limit bandwidth. 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 as described in Section 3.2.1. Although ALTQ shares the same limitation, ALTQ can take advantage of transmission complete interrupts. Dummynet currently does not have a mechanism for packet scheduling nor active buffer management. 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.

4.2 Linux Traffic Control

Linux has a traffic control (TC) framework since version 2.1. 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 minor difference found in the queue operations is that Linux TC defines "requeue" (prepend) instead of "peek". Linux TC employs a dequeue-and-requeue policy while ALTQ employs a peek-and-dequeue policy.

The 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 and ethernet bridging need 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.

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*.

5 Conclusion

There are increasing demands and expectations for network traffic management. Although a variety of technologies are available, there is no single mechanism that satisfies a wide range of requirements. It is important to understand advantages and limitations of different mechanisms.

It is also important to understand the system bottleneck for different link speeds. With a network ranging from a slow modem to a high-speed fiber, the system bottleneck shifts one place to another. The requirements for the hardware and the system configuration also change.

The behaviors of single queueing disciplines are well understood but interaction of different mechanisms, especially in operational settings, needs more study and experience. We hope ALTQ will be of use to gain experience in the field.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.aciri.org/floyd/papers.html.

[7] 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.aciri.org/floyd/papers.html.

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

[9] Rizzo L. 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/.

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

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

[12] 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.