Fitting Theory into Reality in the ALTQ case

Kenjiro Cho
Sony Computer Science Laboratories, Inc.

Abstract

The ALTQ project started as a challenge to implement various theoretical QoS mechanisms onto the existing open source operating systems running on commodity PC hardware. In the course of the ALTQ development, we have faced a number of limitations and complexities imposed by hardware and software. These practical issues are often overlooked by research people.

This paper examines gaps between the theoretical QoS models and the real system. One is in the queue operation model and the other is in the output buffer model. A new set of queue operations are defined in ALTQ as a trade-off between clean abstraction and compatibility with the existing drivers. A token backet regulator is added to the output queue model in order to control the number of packets buffered in network cards.

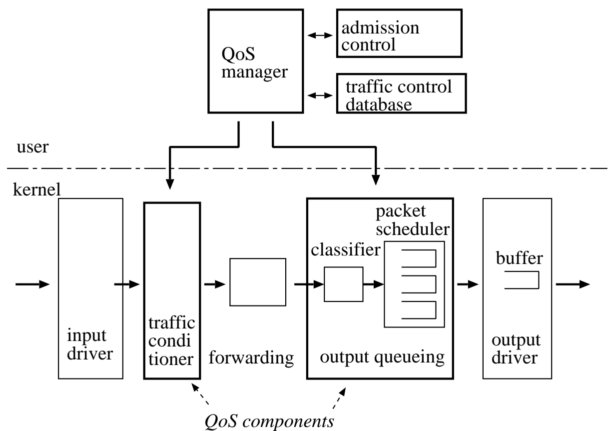

Figure 1: ALTQ traffic control model

1 Introduction

When implementing a theoretical model onto a real system, we sometimes encounter gaps between theory and practice. A real system has various limitations and complexities imposed by hardware and software. It is also well known that a significant portion of a well engineered program is devoted to error handling and exceptional processing. Therefore, the implementation of a theoretical model requires careful consideration of dealing with limitations and errors in the real system.

In addition, the BSD-based operating systems support a wide range of hardware and devices. For instance, a networking subsystem needs to run on various CPU types and work with a diverse set of network cards ranging from 32Kbps modems to Giga-bit Ethernet cards.

A large number of legacy subsystems and device drivers exist in the operating systems. Some of them are intentionally maintained for backward compatibility or for legacy hardware, and others are historical remains from the long incremental evolution.

It is an engineering challenge to redesign a theoretical work to fit into the existing operating system, and make it work for diverse hardware and usage scenarios. These engineering issues are often overlooked by research people. This paper examines gaps between theory and reality in the QoS models, and explains how ALTQ deals with the practical limitations.

2 ALTQ

ALTQ [Cho98, Cho01] is a framework for the BSD systems to support a wide variety of QoS (Quality of Service) components. The goal is to provide a well-engineered framework as well as practical QoS components for further research and experiments. ALTQ started as a platform for QoS related research but has evolved into a traffic management subsystem for daily operational use.

The ALTQ system consists of the three major components. The framework at the bottom takes care of interfaces to the operating system in order to make use of QoS mechanisms. The QoS components actually provide

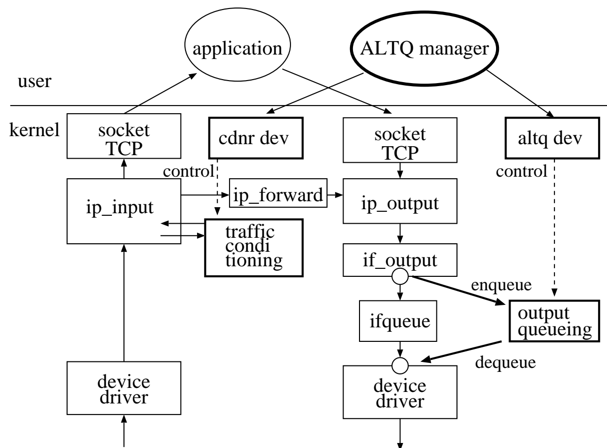

Figure 2: ALTQ system implementation model

QoS. The management tools in the user space take care of interfaces to human or other management software.

Figure 1 shows the system architecture of ALTQ. The key component for providing QoS is packet scheduling at the output queue. In a packet switching network, packets belonging to various flows are multiplexed together and queued for transmission at the output queue associated with the outgoing link. Thus, when arriving packets exceed the link bandwidth, the queued packets need to wait for the preceding packets to be transmitted. This queueing delay is a major source of jitter, fluctuations in the transmission delay.

In the best-effort Internet, arriving packets are transmitted usually by the first-come-first-serve strategy. A FIFO queue is the simplest form with a limited size of the buffer, and used in most router and host implementations. In FIFO, if the buffer becomes full, arriving packets are simply discarded.

On the other hand, it is possible to distinguish certain packets from others, and treat them differently. A classifier is a component to classify packets into different types, usually by looking into certain packet header fields such as source and destination addresses and port numbers. A queueing discipline implements a packet scheduler to provide differentiated treatment for different packets.

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 proactive buffer management such as RED (Random Early Detection) [FJ93, BCC+98] discards packets before the buffer becomes completely full to signal a congestion sign and allow the senders to proactively reduce their sending rates.

Figure 2 shows the implementation model of ALTQ. ALTQ provides an alternative queueing discipline such as CBQ [FJ95] or H-FSC [SZ97] that is controlled by a user program through the pseudo-device interface.

3 Mismatches in Output Queue Models

ALTQ attempts to implement various theoretical proposals onto the off-the-shelf PC platforms. However, in the course of the ALTQ development, we have faced mismatches between theory and real implementations. This section reviews 2 examples of such mismatches. One is in the queue operation model in the existing device drivers, and the other is in the output buffer model in the PC-based hardware.

3.1 Queue Operation Model

A theoretical queue model usually has only 2 queue operations, enqueue and dequeue. In the existing implementations, however, there are other queue operations such as polling a queue or purging a queue. These operations are needed to cope with errors or resource exhaustion which is unavoidable in real implementations.

ALTQ defines a new set of queue operations to support different types of queueing disciplines. It is preferable not to require changes to the existing drivers since there are too many different network device drivers. However, it became apparent that we need to modify the drivers because the existing queue model is not flexible enough to accommodate non-FIFO queues.

There is a design trade-off between clean abstraction and compatibility with the existing software. In our case, we found that a few device drivers are too complex in their error recovery procedures so that it is not possible to cleanly abstract these exceptional situations. The design we adopted is to cover the most of the existing drivers but leave out a few of the exceptional drivers. For those exceptional drivers, we decided to rewrite the part of the drivers to fit them into the new operation model.

The new operation model supports the poll operation that is often used in the existing drivers. Some drivers poll the packet at the top of the queue to knows the required resources such as DMA descriptors and buffer memory. Since the poll operation is not defined in the existing queue macro in the BSD systems, the drivers directly refer to the internal filed of the *ifnet* structure.

In ALTQ, a new macro, *IFQ_POLL()*, is defined to peek at the next packet to be dequeued. The drivers that peek at the queue head are converted to use the poll-and-dequeue sequence. The code on the left side in Figure 3 shows the existing code directly accessing the internal field of the *ifnet* structure. The code on the right side shows the new style using *IFQ_POLL()*.

To support the poll operation, queueing disciplines are written to return the same packet when

```
            ##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 3: poll-and-dequeue in driver

```
            ##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: eliminating prepend operations in driver

IFQ_DEQUEUE() is called immediately after *IFQ_POLL()*.

On the other hand, some drivers put back an already dequeued packet using *IF_PREPEND()* when they fail to obtain enough resources such as DMA descriptors. However, the prepend operation assumes FIFO; it is difficult for a packet scheduler with multiple queues or with the internal state update at the dequeue operation. But the dequeue-and-prepend method can be converted to the poll-and-dequeue method easily in most of the drivers. Therefore, we decided not to support the prepend operation in ALTQ.

Figure 4 shows an example of the conversion from the dequeue-and-prepend method on the right side to the poll-and-dequeue method on the left side. In the new coding style, the next packet is dequeued only after the driver successfully obtains the required resources.

For most of the drivers, the modifications introduced by ALTQ are straightforward; they are simple macro replacements as in Figure 3. Some of the drivers need modifications in their program sequences as described in Figure 4. However, a few of the drivers have elaborate error recoveries that conflict with the new queue semantics. For example, one driver copies a packet into a single *mbuf cluster* when the dequeued packet is in a chained *mbuf* to reduce the required DMA descriptors. This new *mbuf* could be prepended to the queue again if an error occurs in another error recovery function. To make this driver to use the poll-and-dequeue method, we had to rewrite the code and simplify the error handling.

3.2 Output Buffer Model

The second mismatch is in the output buffer model. In the commonly used output buffer model in theory, there is a single queue for a given output link as shown in Figure 5 (a). Theoretical packet schedulers assume that it is enough to select the next packet to transmit among the packets waiting in the output queue.

A real PC system, however, has 2 distinct output buffers; one is implemented by software in the operat-

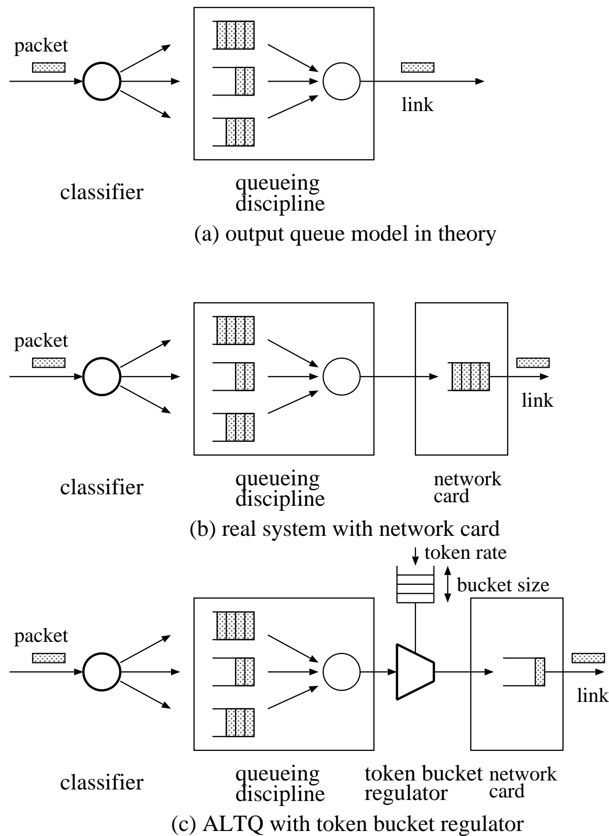

Figure 5: differences in output queue models

ing system and the other is implemented by hardware in the network card as shown in Figure 5 (b).

A large buffer in a network card has a significant impact to the performance of the packet scheduler such as increased delay, bursty dequeues, and a decreased number of interrupts.

Modern network cards support chained DMA; typically, 128 or 256 entries can be chained at a time. A chained DMA looks like a very large transmission buffer from packet scheduling. (Although the DMA structure is different from a normal buffer structure, it is treated as a device buffer in this section for simplicity.)

Also, many modern network cards generate interrupts only when the transmission of all the buffered packets is completed, and some cards allow the driver to program when to interrupt.

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. As a result, it creates a long waiting queue after packets are scheduled by the queueing discipline. From our experience, the buffer size of 10KB is enough for most cards to saturate a 100Mbps link but many drivers build a DMA chain as large as 200KB or much larger. The device buffer has an effect of inserting another large FIFO queue beneath a queueing discipline.

The first problem introduced by the large device buffer 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 other packets to be drained. Thus, even high priority packets end up with a very large delay if there is a large buffer in the network card.

The second problem is that, when the device buffer is large, packets are moved from the queue to the device buffer in a very bursty manner. When a large number of packets are dequeued at a time, the packet scheduler loses a chance to control the sending order. A packet scheduler is effective only when there are backlogged packets in the queue.

The third problem is that the scheduling timing is reduced as interrupts are reduced. 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. A packet scheduler needs to make use of interrupts to transmit backlogged packet in the queue. This is essential to making a queueing discipline work-conserving. Otherwise, the link is left idle while packets are backlogged 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 in the device. However, the problem becomes apparent when preferential scheduling is enabled.

The ideal transmission buffer size is the minimum amount required to fill up the link speed, but it depends on the network card, the link type, the CPU speed and other factors. 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 network card.

ALTQ employs token bucket regulators to solve the gap of the output buffer models as shown in Figure 5 (c). The purpose of the token bucket regulator is to limit the amount of packets buffered in the device in a device-independent manner. The token bucket regulator is implemented as a wrapper function of the dequeue operation.

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. The bucket size controls the amount of burst that can dequeued at a time, and controls a greedy device trying to dequeue packets as much as possible.

When 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. However, rate-limiting has a side-effect; it makes it harder to make use of interrupts. If the configured rate is smaller than the actual transmission rate, the rate limit would be still in effect at the time of the transmission complete interrupt, and the rate limiting falls back to the kernel timer to trigger the next dequeue. On the other hand, when the rate is set to the actual transmission rate or higher, transmission complete interrupts can trigger the next dequeue.

By default, ALTQ sets the bucket size heuristically according to the configured interface bandwidth. A user can manually adjust the bucket size to minimize the delay.

4 Conclusion

ALTQ started in 1997 as a platform for QoS related research. ALTQ has implemented several theoretical queueing models onto the existing BSD systems. ALTQ is integrated into NetBSD and OpenBSD official releases and widely used for traffic management.

Sometimes, the existing system does not agree with theoretical models. In this paper, we have reviewed 2 examples found in the development of ALTQ. One is the queue operation model in the existing device drivers, and the other is the output buffer model in the PC architecture.

Queue operations in theory do not include error handling but it is important in the real system. ALTQ redefines a new set of queue operations by looking into the usage in the existing drivers. Since ALTQ needs to work with a diverse set of hardware and software, we found it essential to have a good balance between clean abstraction and compatibility with the existing software.

The buffer model mismatch is a result of the evolution of network cards. Old ISA-based cards had only small on-board buffers without DMA capability, but modern PCI-based cards are capable of chained DMA. A large buffer in a network card is not assumed in theoretical models but it has a significant impact to the performance of packet scheduling. ALTQ minimizes the impact of large device buffers by adding a simple component, a token bucket regulator, to hide the architectural difference in a device independent manner.

It is important for researchers to learn from engineering experiences. When there is a gap between a theoretical model and the real system, the theoretical model may need to be reconsidered. In addition, such a difference might provide an opportunity for a new research area.

References

[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.

[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.

[Cho01] Kenjiro Cho. *The Design and Implementation of the ALTQ Traffic Management System.* PhD thesis, Keio University, January 2001.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–413, August 1993.

[FJ95] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[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.