

Fitting theory into reality in the ALTQ case

Kenjiro Cho

kjc@csl.sony.co.jp

Sony Computer Science Labs, Inc.

WIDE Project

Japan Advanced Institute of Science and Technology

is QoS (Quality of Service) still needed?

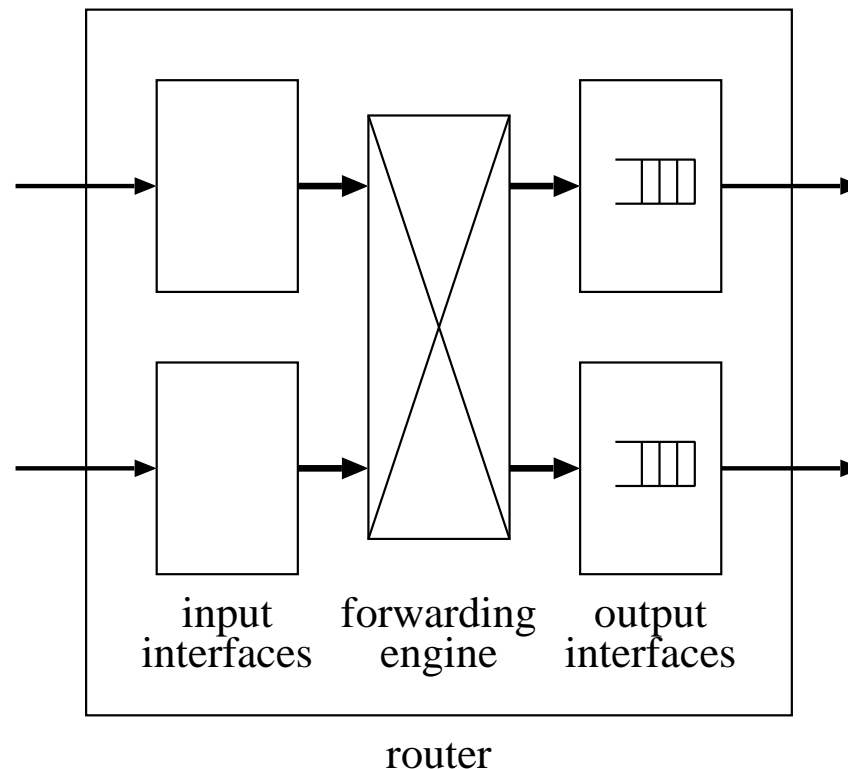
- some people claim
 - bandwidth became cheaper, we don't need QoS any more
- fat pipes do not solve all problems
 - bottlenecks will not go away
 - increasing inequality in bandwidth
 - congestion at bandwidth gaps
 - impact of p2p file sharing
 - p2p traffic consumes ISP backbone bandwidth
 - access link will become bottleneck again in a few years
 - access link technology: modem - ADSL - fiber?

trade-off between provisioning and control

- a wide range of spectrum
 - not "QoS vs. over-provisioning"
- balance between provisioning and controlling
 - at each level, among different levels
 - provisioning: provides headroom
 - controlling: provides protection mechanisms
- for properly provisioned network
 - QoS has effects similar to increasing capacity
 - shifts congestion starting point

QoS by packet scheduling at output queue

- QoS (Quality-of-Service) in the Internet
 - traffic control on output queue
 - assumption: packet switching is faster than link speed
 - provisioning becomes part of QoS
- out-of-scope
 - QoS in switch hardware, process scheduling, applications



Why ALTQ? (1/2)

- importance of software implementation research
- allows peers to reproduce results
 - comparable with
 - justification process in science
 - proof in math
 - power of computer, along with info sharing over Internet
 - ability to enable peers to reproduce and verify results
 - to accelerate cycle of innovation
- reproducible results are essential to robustness of results
 - programs are tested in a wide variety of contexts
 - bugs are found and fixed by others
- sharing software components leads to new ideas
 - good software inspires others
 - reuse of components

Why ALTQ? (2/2)

- QoS: elusive, confounding, confusing
 - too many conflicting concepts
 - needs tools and experiences
- QoS is hard to reproduce
 - research community lacked a common platform
 - start of the ALTQ project (on commodity PC and free UNIX)
- ALTQ
 - many experiments done with ALTQ
 - quality of ALTQ improved through feedback from users
 - stimulated many research projects
- we should learn from previous ideas
 - not only through papers
 - but also through working computers and Internet
- software implementation to reproduce research results
 - no less important than new discoveries for tech advancement

ALTQ goals

- to provide QoS platform
 - flexibility for further research
 - stability as a platform
 - performance to prove QoS in the Internet
- to explore system architecture to support QoS
 - ALTQ: design and implementation
 - system framework
 - abstractions of QoS blocks
 - interfaces QoS blocks into the existing OS
 - forwarding mechanisms
 - QoS components on packet forwarding path
 - perform actual QoS functions
 - management mechanisms
 - control QoS components
 - functions not required on forwarding path

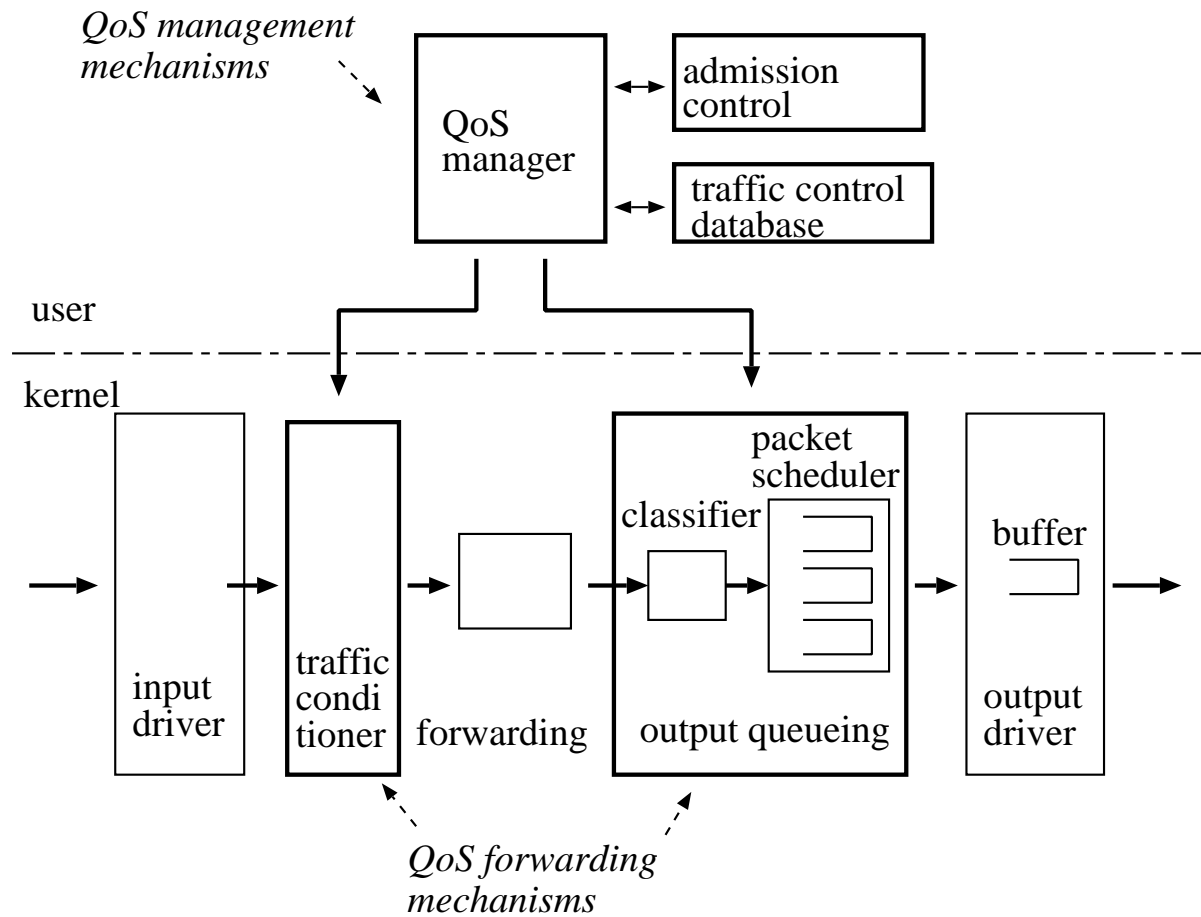
related work

- custom queueing implementation [GF95,SZ97]
 - not generalized for a framework
 - ALTQ is the first QoS framework in wide use
 - follow-on projects[DDPP98,Alm99,MKJK99]
- switch to different QoS blocks
 - similar to protocol switch in BSD UNIX
 - differs from module based approach
 - STREAMS[Rit84], x-kernel[PHOR90]
- process scheduling
 - complementary to packet scheduling
- dumynet[Riz97]
 - extension of firewall rules
- Linux TC[Alm99]
 - packet scheduler framework similar to ALTQ

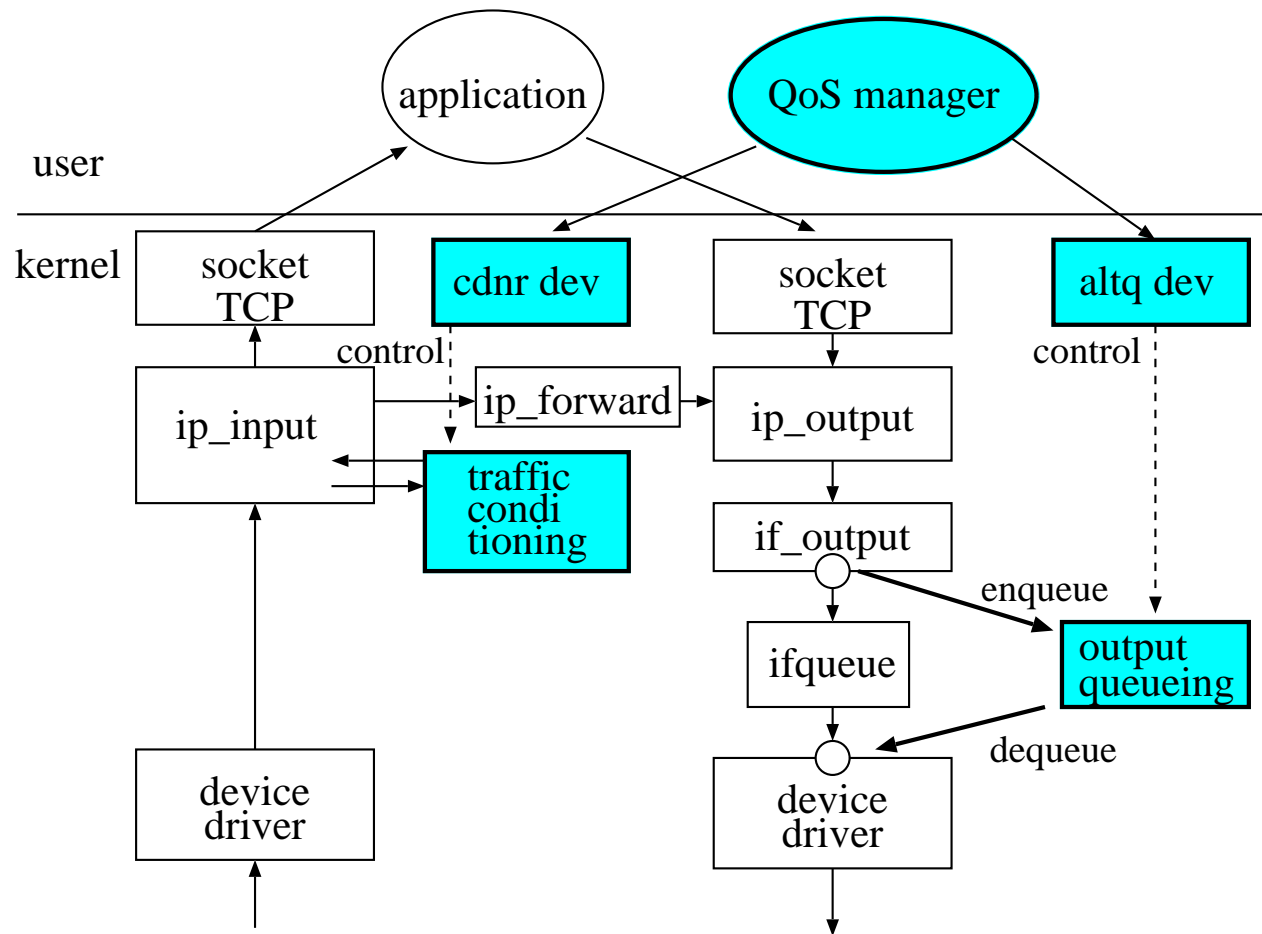
ALTQ system architecture

- traffic control on output queues
 - based on Intserv and Diffserv model
 - traffic conditioning block on input interface
 - output queueing block on output interface
- flexibility to accommodate various QoS components
 - switch to QoS control block
 - 3-step approach
 - system framework
 - forwarding mechanisms
 - management mechanisms

ALTQ traffic control model



ALTQ system implementation model



gaps between theoretical models and real systems

- a real system has various limitations and complexities
 - imposed by software and hardware
- error handling and exceptional processing
 - are significant portion of a well-engineered program
- BSD systems support a wide range of hardware and devices
 - various CPU types
 - network cards from 32Kbps modem to GbE
 - legacy subsystems
 - for backward compatibility or for legacy hardware
 - historical remains from long incremental evolution
- it is an engineering challenge to redesign a theoretical work
 - to fit into the existing OS
 - to make it work for diverse hardware and software
- issues are often overlooked by research people

mismatches in output queue models

- 2 examples found in the ALTQ development
 - queue operation model in device drivers
 - output buffer model in PC-based hardware

mismatch in queue operation model

- in a theoretical queue model
 - enqueue and dequeue are enough
- in a real system
 - poll or purge queue
 - to cope with errors and resource exhaustion
- ALTQ defines a new set of queue operations
 - to support different types of queueing disciplines
- need to modify the existing drivers
 - since the current queue abstraction isn't flexible enough
- design trade-off between clean abstraction and compatibility
 - ALTQ supports poll and purge, but not prepend

current queue model in BSD UNIX

- FIFO queue assumption
 - drop-tail assumption
 - lack of poll operation
 - lack of purge operation
 - use of prepend operation
- drop-tail example

```
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);
```

output queue abstraction in ALTQ

- simple blackbox queue model
 - hides packet scheduler and buffer management
- 4 queue operations defined in ALTQ
 - ENQUEUE
 - enqueues a packet
 - could discard a packet
 - DEQUEUE
 - removes a packet from the queue
 - packet scheduling
 - POLL
 - returns a packet without removing from the queue
 - PURGE
 - discard all packets in the queue
- allows incremental transition

modifications to the existing drivers

- straightforward for most of the drivers
 - simple macro replacements
- several drivers are changed
 - to use poll-and-dequeue
 - not to use prepend
 - for queueing disciplines with multiple queues
- a few drivers had to be simplified
 - since their error recoveries are too complex

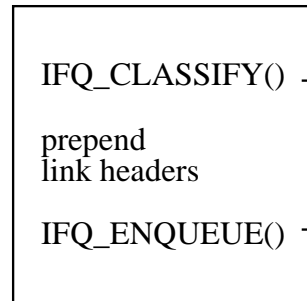
new output queue macros

Table 1: new output queue macros

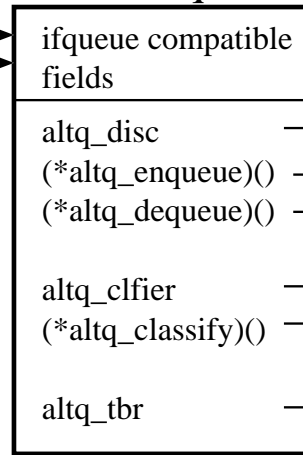
Macro	Description
IFQ_ENQUEUE(ifq, m, pktattr, err)	enqueue a packet to the queue
IFQ_DEQUEUE(ifq, m)	dequeue a packet from the queue
IFQ_POLL(ifq, m)	poll the next packet to be dequeued
IFQ_PURGE(ifq)	discard all packets in the queue
IFQ_IS_EMPTY(ifq)	TRUE if the queue is empty
IFQ_CLASSIFY(ifq, m, 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

output queue implementation in ALTQ

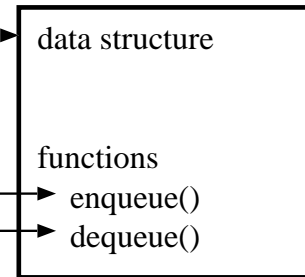
if_output



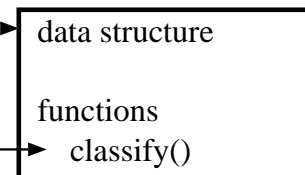
struct ifaltq



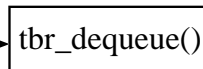
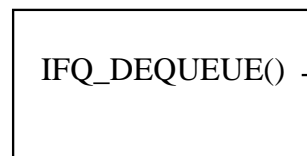
queueing discipline



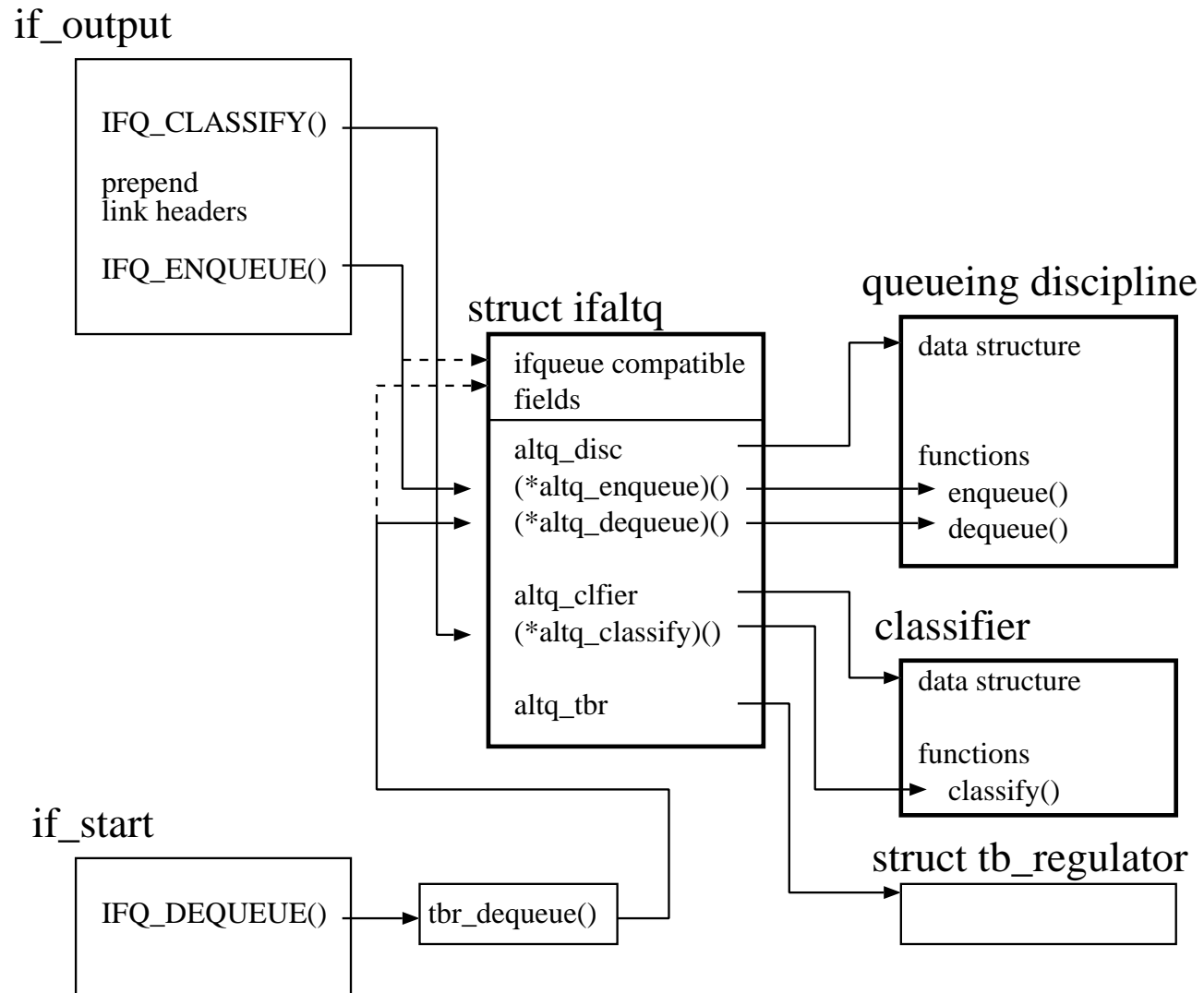
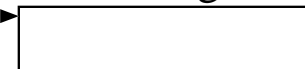
classifier



if_start



struct tb_regulator



transition to the new model

- IF macros are refined

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

- modifications to existing code
 - enqueue
 - dequeue
 - empty check
 - poll-and-dequeue
 - eliminating IF_PREPEND

enqueue operation

##old-style##

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

    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);

    ifp->if_obytes += m->m_pkthdr.len;
    if (m->m_flags & M_MCAST)
        ifp->if_omcasts++;
    if ((ifp->if_flags
        & IFF_OACTIVE) == 0)
        (*ifp->if_start)(ifp);
    splx(s);
    return (error);
}
```

##new-style##

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

    mflags = m->m_flags;
    len = m->m_pkthdr.len;
    s = splimp();
    IFQ_ENQUEUE(&ifp->if_snd, m,
                &pktattr, error);
    if (error != 0) {
        splx(s);
        return (error);
    }
    ifp->if_obytes += len;
    if (mflags & M_MCAST)
        ifp->if_omcasts++;
    if ((ifp->if_flags
        & IFF_OACTIVE) == 0)
        (*ifp->if_start)(ifp);
    splx(s);
    return (error);
}
```

dequeue operation

##old-style##

```
IF_DEQUEUE(&ifp->if_snd, m);
```

##new-style##

```
|  
| IFQ_DEQUEUE(&ifp->if_snd, m);  
| if (m == NULL)  
|     return;  
|
```

empty check

##old-style##

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

##new-style##

```
| if (!IFQ_IS_EMPTY(&ifp->if_snd))
```

poll-and-dequeue

##old-style##

```
m = ifp->if_snd.ifq_head;
if (m != NULL) {

    /* use m to get resources */
    if (something goes wrong)
        return;

    IF_DEQUEUE(&ifp->if_snd, m);

    /* kick the hardware */
}
```

##new-style##

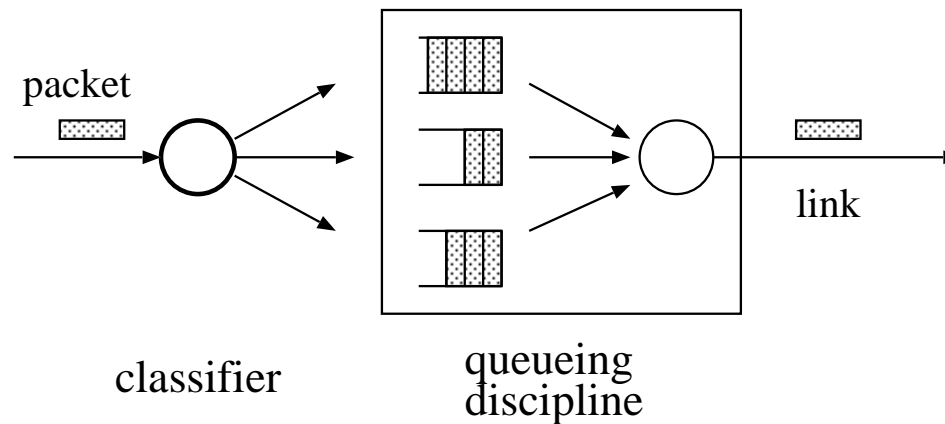
```
| IFQ_POLL(&ifp->if_snd, m);
| if (m != NULL) {
|
|     /* use m to get resources */
|     if (something goes wrong)
|         return;
|
|     IFQ_DEQUEUE(&ifp->if_snd, m);
|
|     /* kick the hardware */
| }
|
```

eliminating IF_PREPEND

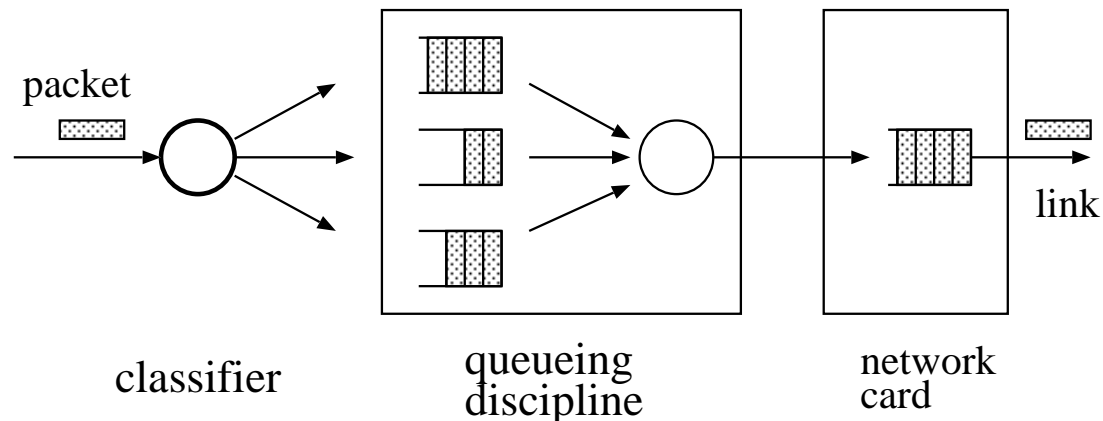
```
##old-style##                                ##new-style##
IF_DEQUEUE(&ifp->if_snd, m);                  | IFQ_POLL(&ifp->if_snd, m);
if (m != NULL) {                             | if (m != NULL) {
    if (something_goes_wrong) {              |     if (something_goes_wrong) {
        IF_PREPEND(&ifp->if_snd,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

mismatch in output buffer model

- output queue in theory has a single buffer
- a real system has 2 buffers; one by software in OS, the other by hardware in network card



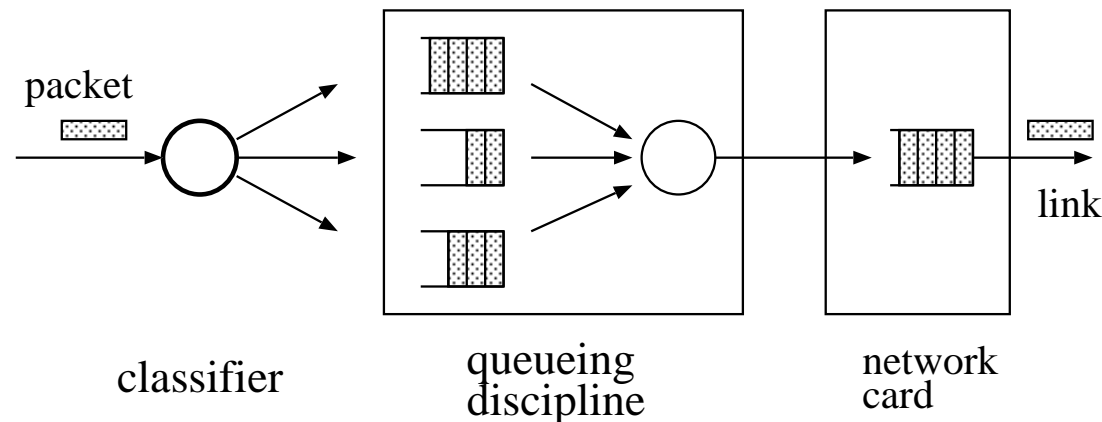
(a) output queue model in theory



(b) real system with network card

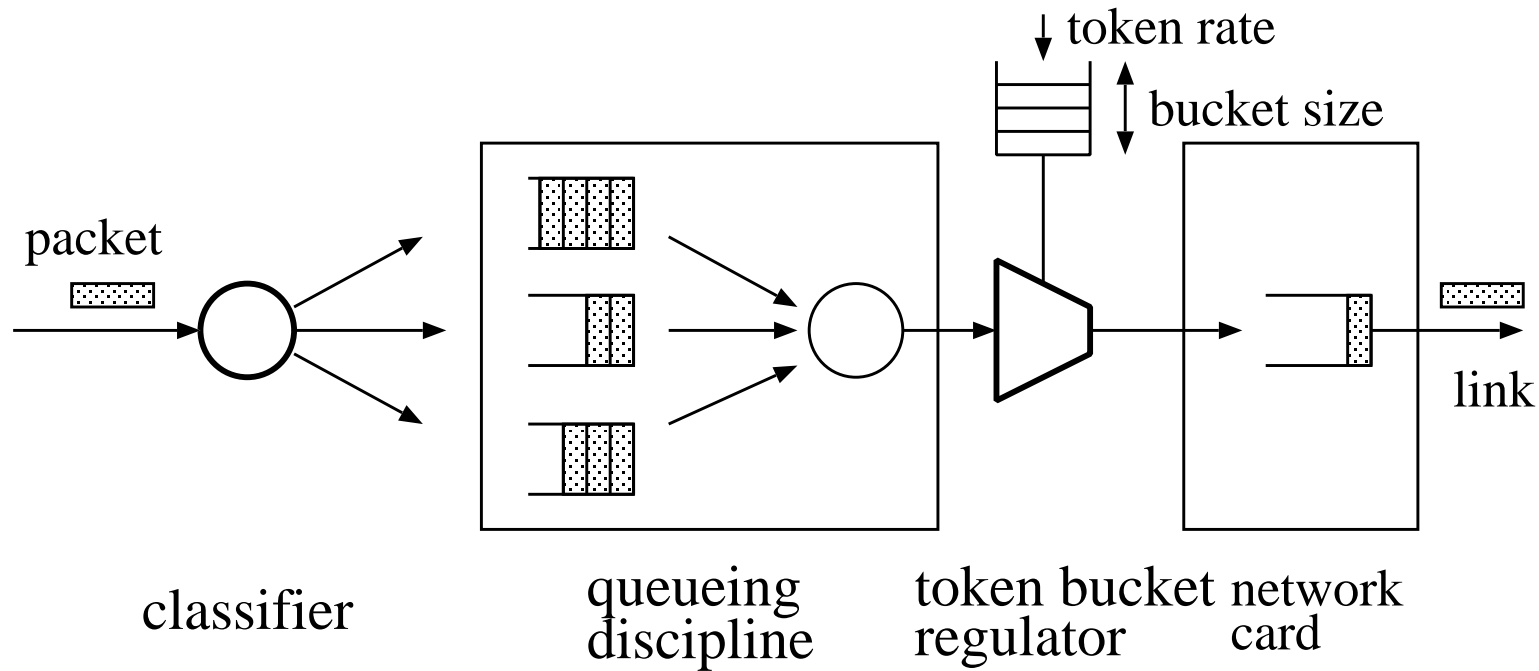
effects of large buffer in network card

- transmission buffer (DMA chains) in the device
 - to avoid under-run, and to reduce interrupts
 - (didn't exist in ISA-based cards)
- negative effects to QoS when buffer is too large
 - increased delay: another FIFO below scheduler
 - dequeues become bursty: scheduler loses control
 - reduced interrupts: CPU load vs. CPU control
- these problems are invisible under FIFO!
 - drivers try to make DMA as long as possible!
 - for 100Mbps, 10KB is enough but it goes larger than 200KB!



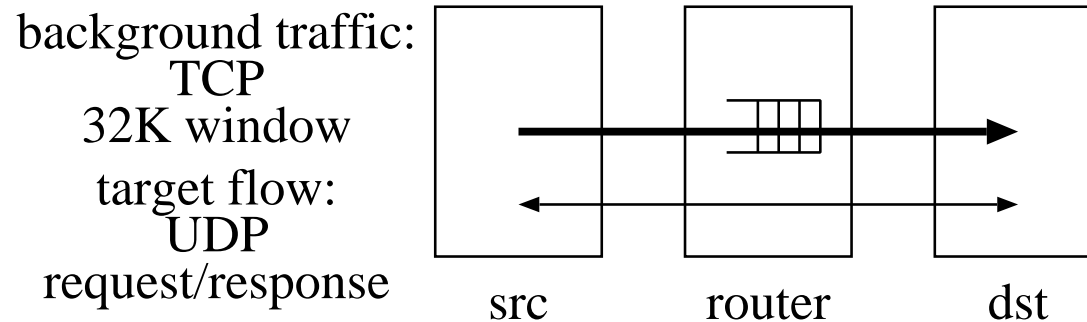
token bucket regulator in ALTQ

- device-independent mechanism to control buffering in drivers
 - works with existing drivers without modification
 - 2 tunable parameters
 - token rate controls long-term transmission rate
 - bucket size limits the burst size



test:latency differentiation with competing traffic

- create backlog by background TCP
- measure latency of target flow



- controlled latency < 3-packet transmission time
 - 1 packet takes 1.2msec over 10baseT
 - 32KB window creates 26msec queueing delay

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	346.76	2.88

ALTQ status

- development on KAME
- integrated into NetBSD and OpenBSD releases
 - NetBSD: based on altq-3.1
 - OpenBSD: integrated into pf
 - (FreeBSD: efforts to port pf/ALTQ)
- 2 versions of ALTQ
 - altq-3.1: focuses on flexibility as a research platform
 - includes various experimental codes
 - pf/altq: focuses on user needs
 - integration with pf: single config for pf and altq
 - limited disciplines (CBQ, HFSC, PRIQ) and knobs
 - future direction
 - the smaller set (pf/altq) for BSDs
 - research support will remain in KAME

summary

- ALTQ has implemented theoretical QoS mechanisms onto BSD systems
 - started as a platform for QoS research
 - but evolved into traffic management subsystem for daily use
- sometimes, the existing system doesn't agree with theoretical models
 - this talk reviews 2 examples in ALTQ
 - (1) queue operation models
 - (2) output buffer models
- importance of learning from experiences
 - if there is a gap between a theoretical model and the real system
 - we may need to reconsider the theoretical model
 - or, we might be able to find new research topics