Supporting Excess Real-time Traffic with Active Drop Queue*

Yaqing Huang[a], Roch Guérin[a] and Pranav Gupta[b†]

[a] {yaqing, guerin}@ee.upenn.edu
Dept. of ESE, University of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19020, USA.

[b] guptap@seas.upenn.edu
32 Sunset Rd. Demarest, NJ 07627, USA.

The requirements of real-time applications mean that they often stand to benefit from network service guarantees, and in particular delay guarantees. However, most of the mechanisms that provide delay guarantees do so by hard-limiting the amount of traffic the application can generate, i.e., to conform to a traffic contract. This can be a significant constraint that conflicts with the operation of many real-time applications. Our purpose is to propose and investigate solutions that overcome this limitation. Our four major goals are (1) guarantee a delay bound to a contracted amount of real-time traffic; (2) transmit with the same delay bound as many excess real-time packets as possible; (3) enforce a given link sharing ratio between excess real-time traffic and other service classes, e.g., best-effort; (4) preserve the ordering of real-time packets, if required. Our approach is based on a combination of buffer management and scheduling mechanisms. We evaluate its "cost" by measuring the overhead involved in an actual implementation, and we investigate its performance by simulations using video traffic traces.

1. Introduction

The demand for real-time multimedia applications such as streaming video, VoIP, etc., has been increasing in the recent years. However, despite the ever increasing speed of backbone links, access links often remain congested and, therefore, introduce throughput and delay limitations. Those limitations are particularly detrimental to real-time applications that have a more limited ability to adapt to fluctuations in network conditions than data applications. As a result, real-time applications have been both a prime candidate and a strong motivation for introducing service guarantees in the Internet, or at least on access links.

Service guarantees are traditionally in the form of rate and delay guarantees, with scheduling and buffer management the two main underlying mechanisms used to enforce them. Such guarantees can be provided by buffer management alone [1] if the link speeds are high enough. However, on access links that are our focus, both mechanisms need to be considered. In such a setting, the provision of delay guarantees is typically associated with the explicit identification of the traffic to which those guarantees apply, i.e., in the form of a traffic contract. In particular, existing mechanisms require that the application limits its traffic according to its contract.

Traffic contracts are often in the form of token buckets, e.g., [2,3], that specify a fixed rate while allowing for short term rate variations through a "burst tolerance." Conformance is en-

*This work is supported in part by NSF grant ANI-9902943, ANI-9906855 and ITR-0085930 and by a grant from Nortel Networks.

†This work is done during Pranav Gupta's Master study at Penn.

forced by either dropping, reshaping, or marking as excess traffic, packets that violate the contract. Violations occur when the application transmits faster than its contracted rate for an extended period of time, or generates a burst of packets that exceeds its burst tolerance.

Avoiding contract violations is difficult if not impossible for many real-time applications. For example, video traffic, can exhibit significant and prolonged changes in rate as a function of scene characteristics. Similarly, voice traffic between two VoIP gateways varies based on both the number of voice connections in progress, and the rate fluctuations within each connection. Dropping and reshaping can both result in substantial quality degradation. As a result, allowing non-conformant real-time packets to enter the network by marking them as excess traffic is desirable. However, for such an option to be useful, it is important to ensure that the excess packets be transferred across the network within the desired delay bound. In addition, such preferential delay treatment should not be offered at the expense of other service classes.

Our aim is to devise mechanisms that achieve the following goals at the lowest cost:

1. Ensure zero losses and delay bounds guarantees to conformant, real-time traffic,
2. Transmit as much excess, real-time traffic as possible and within the delay bound,
3. Enforce link-sharing ratio between excess, real-time traffic and other service classes.
4. Support, as an optional feature, the ordering of real-time (conformant and excess) packets.

There have been several prior works with related goals. ABE [4,5] [3] allows real-time applications to receive preferential delay treatment, without the requirement of a traffic contract. This is achieved through a scheduling mechanism that trades-off throughput for lower delay. There are, however, significant differences between ABE and the mechanisms described in this paper. First, we target explicit service guarantees and assume the existence of traffic contracts for real-time traffic. Second, we distinguish between "conformant" and "excess" real-time traffic, and focus on buffer management to remove *expired* excess packets. Third, we explicitly control the level of link-sharing between excess real-time traffic and other service classes.

We propose Active Drop Queue (ADQ) that achieves the above goals through a combination of scheduling and buffer management mechanisms. We denote the real-time packets that conform to the traffic contract as "conformant" traffic, and the real-time packets that exceed the contract as "excess" traffic. We assume that the responsibility of marking packets lies with the users, even if contract verification/enforcement is likely to be performed by the network. As we shall see, marking excess traffic as conformant could cause many if not most conformant packets to fail their deadlines. Thus, even without network verification, it is not wise to do so.

Scheduling in ADQ guarantees the delay bound for conformant traffic and enforces link-sharing between excess traffic and other classes. We present two versions of ADQ, different in the choice of schedulers, which represent trade-offs between complexity and efficiency.

Buffer management in ADQ ensures that only non-expired excess packets are transmitted, by promptly removing "expired" packets. Expired excess packets unnecessarily occupy storage space and waste bandwidth if transmitted, both of which can affect excess traffic throughput. Thus, the main challenges are to remove expired excess packets with the smallest overhead, preferably in $O(1)$ time [4], and do so while removing as few non-expired packets as possible.

Another goal of ADQ is to optionally preserve the overall ordering of real-time packets. This is desirable when all packets are generated by the same application, e. g., a VBR video application that occasionally exceeds its contracted rate and marks the corresponding packets as

[3] The BEDS proposal [6] is another work that shared some of the same goals.

[4] The implicit time unit is a packet transmission time.

excess. Conversely, preserving ordering between conformant and excess packets is unnecessary if they are generated by different users. For example, a VoIP gateway with a certain amount of contracted bandwidth for voice sessions can instead of blocking new sessions when the bandwidth is fully used, let them proceed albeit marked as excess. Because packets from excess and conformant sessions are independent of each other, ordering is not required.

In the rest of the paper, we review the general structure of ADQ and present the two scheduling schemes we investigate in Sections 2 and 3, while buffer management is covered in Section 4. Section 5 introduces a Linux implementation of ADQ aimed at validating its feasibility. In section 6, we evaluate ADQ's performance through NS [9] simulations, and quantify its complexity by benchmarking our implementation. Section 7 concludes the paper.

2. ADQ overview

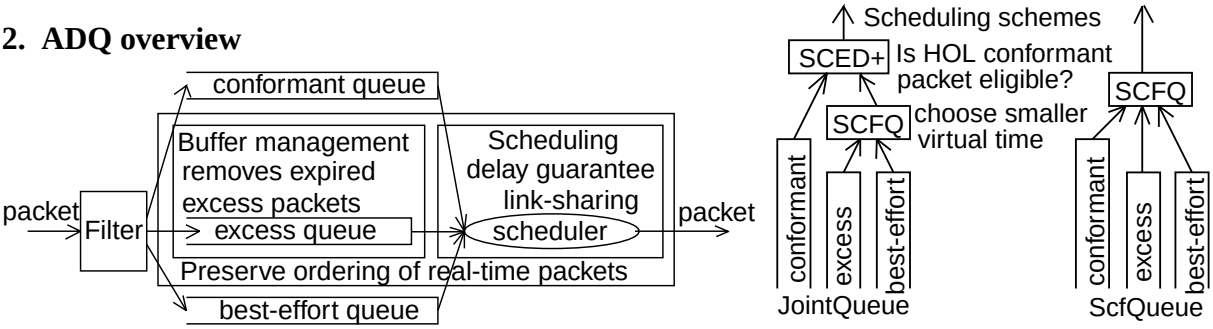

Figure 1. ADQ general structure

The ADQ algorithm, shown in Fig. 1, combines both scheduling and buffer management mechanisms. It relies on the scheduler to provide delay guarantees to conformant traffic and enforce link-sharing between excess and best-effort traffic. Buffer management also ensures that transmitted excess packets meet their delay bounds. ADQ can also be configured to preserve the ordering of all real-time packets.

We identify three types of traffic: conformant, excess and best-effort, and each keeps a separate, logical, FIFO queue. Upon arrivals, packets are directed to the corresponding queue. A fixed amount of buffer is dedicated to the real-time traffic. Part of this buffer is assigned to the conformant traffic for which we compute the amount of buffer needed to avoid losses. The remaining real-time buffer is assigned to the excess queue. When the excess queue cannot accommodate an arriving packet, older packets are removed from the head of the excess queue. Hence, excess packets are never dropped on arrival. However, they may be removed later, either to make room for a newer arriving packet, or because they become expired. The best-effort queue is a fixed size FIFO queue, and arriving packets are dropped if the queue is full.

We assume that both conformant and excess packets require the same constant delay bound Δ. In other words, when a real-time packet k arrives at time t, it is assigned a deadline $d_k = t + \Delta$. Packet k must either be transmitted before d_k or removed from buffer after it has expired.

Ideally, conformant packets should be transmitted just before expiration, so that we have as many opportunities as possible to transmit excess packets[5]. Our first scheme, a combination of SCED+ and SCFQ similar to [10], satisfies this requirement. SCED+ delays the transmissions of conformant packets until the last moment, and the remaining transmission opportunities are shared between excess and best-effort packets, based on the SCFQ scheduler.

Due to its lower complexity, we also explore a scheme that uses a single SCFQ scheduler. The disadvantage is that it often transmits conformant packets earlier than necessary, thus decreasing

[5] As we discuss later, this needs to be tempered when ordering of conformant and excess packets is required.

the transmission opportunities available to excess packets before expiration. The ADQ version that relies on two schedulers is named "JointQueue", and the SCFQ only version "ScfQueue".

Because the volume of excess traffic is unknown, scheduling alone is not sufficient to ensure a delay bound to excess packets. The key is to be able to identify and remove *expired* excess packets. ADQ's buffer management addresses this issue using two procedures: "synchronization" and "clean-up". "Synchronization" locates and removes expired excess packets based on the ordering of arrivals between conformant and excess packets, and the fact that all real-time packets have the same delay bound. "Clean-up" is triggered only occasionally when "synchronization" is not frequent enough. Both are detailed in Section 4.

Fig. 2 illustrates the typical behavior of "JointQueue". For simplicity, the example assumes that all packets are 125 bytes, the delay bound is 0.5 ms, and it takes 0.1 ms to transmit each packet. We also assume that the excess and the best-effort share the residual bandwidth equally.

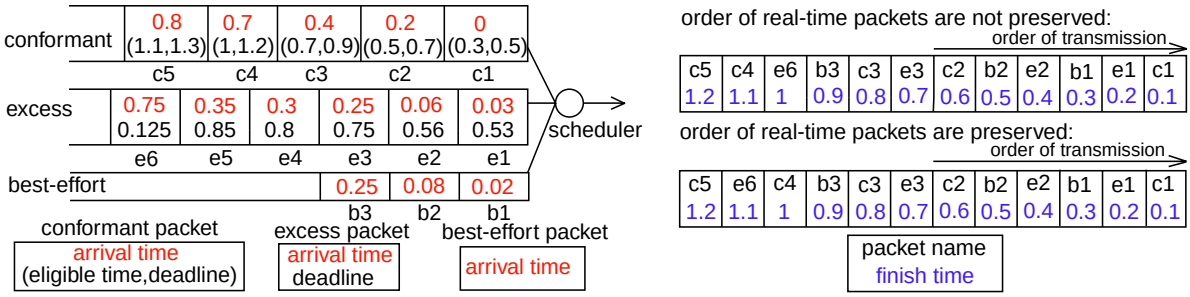

Figure 2. ADQ example

3. Scheduling algorithms

3.1. JointQueue

In the JointQueue scheme, SCED+ provides delay guarantee and lossless performance to the conformant traffic, and SCFQ is used to enforce link-sharing between the excess and the best-effort traffic. By assigning a proper service curve, SCED+ can schedule conformant packets as late as possible without violating their delay requirement. Assuming a constant delay bound Δ and a token-bucket-modelled arrival curve f_c, a service curve S as shown in Fig. 3 satisfies our goal. We can compute the buffer requirement of the conformant queue to avoid packet loss by (1) [7] and the eligibility time of a conformant packet c_k arriving at time t by (2) [7].

When making scheduling decision, the scheduler first considers the head-of-line (HOL) conformant packet c_1. If c_1 is eligible, i.e., $eligible_1^c \leq current\_time$, c_1 is picked for transmission. Otherwise, the scheduler chooses between HOL excess or HOL best-effort packets, whichever has the smaller virtual time. The virtual times of excess and best-effort packets, v_k^e and v_k^b respectively, are computed according to SCFQ to enforce a link-sharing ratio of $e\_ratio : b\_ratio$. In (3), $v(a_k^e)$ and $v(a_k^b)$ are the system virtual times upon arrival of excess packet e_k and best-effort packet b_k, respectively. v_j^e is the virtual time of the last transmitted excess packet before e_k. (Note that not all enqueued excess packets may end up being transmitted.) v_{k-1}^b is the virtual time of the last transmitted best-effort packet before b_k. L_k^e and L_k^b are the lengths of e_k and b_k.

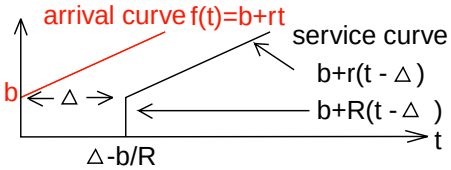

Figure 3. Arrival and services curves with service rate R.

$$c_{limit} = b + r\left(\Delta - \frac{b}{R}\right) \quad (1)$$

$$eligible_k^c = t + \Delta - \frac{b}{R} \quad (2)$$

$$v_k^e = \frac{L_k^e}{e\_ratio} + \max\left(v(a_k^e), v_j^e\right), \quad \text{and} \quad v_k^b = \frac{L_k^b}{b\_ratio} + \max\left(v(a_k^b), v_{k-1}^b\right). \tag{3}$$

3.2. ScfQueue

It is possible to use SCFQ alone to provide delay and lossless performance guarantees to conformant packets, as well as to enforce link-sharing. However, SCFQ will often transmit conformant packets earlier than necessary, and this may negatively affect the excess throughput.

We first compute the rate r_c for the conformant traffic to satisfy its delay requirement. Given r_c, we compute the buffer requirement of the conformant queue to avoid packet losses. Finally, we compute the rates r_e and r_b for the excess and best-effort traffic based on the link-sharing policy. Packets are scheduled in order of their virtual times computed based on r_c, r_e and r_b.

4. Buffer management

The key issue in supporting excess real-time traffic is to be able to identify and remove *expired* excess packets. We propose a buffer management scheme that can remove expired excess packets in $O(1)$ time through two procedures: "synchronization" and "clean-up".

4.1. Synchronization

Synchronization relies on the fact that when a real-time packet p expires, all real-time packets arrived before p are expired, since they all have the same delay bound. Assuming that conformant packets are transmitted at or close to their deadlines, the transmission of a conformant packet c_k can be used to "synchronize" the excess queue by removing all excess packets that arrived before c_k. This is the basic idea behind "synchronization". The efficiency of the procedure depends on both our ability to identify excess packets that arrived before a conformant packet, and on the validity of our assumption that a conformant packet is transmitted at or slightly before its deadline. This latter aspect depends on both the traffic envelope of the conformant traffic and the scheduler used. The smoother the traffic, the less likely it is that conformant packets are transmitted much before their deadlines. Similarly, a scheduler such as SCED+ that schedules conformant packets as late as possible, may result in better results than SCFQ.

The ability to synchronize excess packets with a conformant packet can be accomplished easily because we only need to identify those excess packets that arrived before it and after the preceding conformant packet. This effectively divides the excess queue into "segments", synchronized by conformant packets. Segment are shown in Fig. 4 and defined by having each excess packet contain a pointer, *syncIndex*, pointing to the conformant packet synchronizing it, and a pointer, *segIndex*, pointing to the first excess packet in the next segment. When transmitting a conformant packet, synchronization is performed if the HOL excess packet e_1's *syncIndex* points to the conformant packet being transmitted. If this is the case, then e_1's *segIndex* locates all the excess packets that need to be removed. Although the number of excess packets in the first segment is not $O(1)$ in general, the total buffer used by these packets are continuous and lies between e_1 and the address specified by e_1's *segIndex*. Releasing such continuous buffer requires only $O(1)$ time, and so does the synchronization procedure.

4.2. Clean-up

Under normal circumstances, namely when there is a regular stream of conformant packets, synchronization is triggered frequently enough to remove expired excess packets in a timely manner. However, a burst of excess packets or lack of conformant packets for an extended

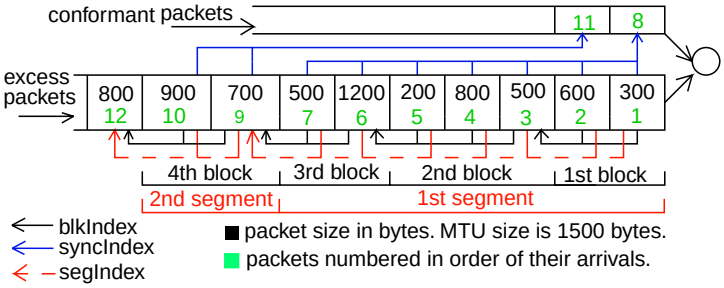

Figure 4. Example of segments and blocks.

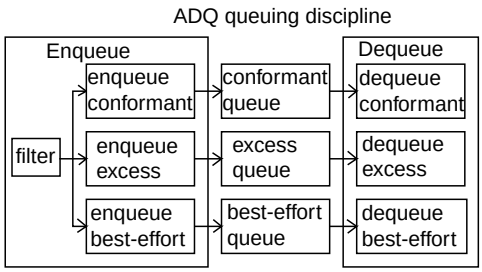

Figure 5. ADQ queuing discipline.

period of time, can result in the build-up of a long segment of excess packets that can remain in the excess queue long after they have expired. "Clean-up" is used to handle such rare cases.

Clean-up is called in either one of two situations. First, when an incoming excess packet finds the excess queue full; and second, when the scheduler finds the HOL excess packet expired. In the first situation, the goal is to remove enough packets from (the head of) the excess queue to make room for the incoming packet. Note that this may involve the removal of some non-expired excess packets. However, the replacement of "older" packets with a "newer" one should help reduce the likelihood of a subsequent clean-up triggered by the second situation, which goal is to find a non-expired excess packet to take advantage of an available transmission opportunity. This can always be accomplished by removing the first segment, because all excess packets beyond the first segment arrived *after* the current HOL conformant packet, which is not yet expired. However, this may cause the unnecessary removal of non-expired packets. Thus, we introduce two intermediate steps before resorting to removing the first segment.

A clean-up procedure involves two type of pointers: *segIndex*, as used in synchronizations; and *blkIndex*, a pointer contained in each excess packet, and pointing to the first excess packet in the next "block". A block consists of one or more contiguous excess packets, so that their total length is at least equal to the link MTU. See Fig. 4 for an example. We assume that packets 1 to 4 are expired when the clean-up procedure is called. Note that the first block in Fig. 4 is not complete because some of its packets have already been transmitted.

- If clean-up is called to make room for an arriving packet, it removes the first block. This may not be enough when the first block is not complete, as will be the case in our example. Then it proceeds to remove the second block, which is always complete. This always guarantees enough space. Assuming that the incoming packet is 1200 bytes in the example of Fig. 4. Packets in the first two blocks, i. e. , packet 1 to 5, will be removed.
- If clean-up is called to identify a new non-expired HOL excess packet, ideally, it should remove only the expired excess packets. This is packet 1 to 4 in our example. However, searching an ordered list typically requires $O(\log n)$ time. Thus, we trade accuracy for simplicity. We first attempt to remove the first block. If the new HOL excess packet is still expired, we proceed to remove the second block. If this fails again, we default to removing the whole first segment, which guarantees a new non-expired HOL packet. In Fig. 4, the clean-up is completed when the first two blocks of packets, namely, packet 1 to 5, have been removed. Packet 6 becomes the new HOL excess packet and is transmitted.

Clean-up removes either the first segment or the first one(two) block(s) of packets. As the synchronization procedure, it only requires $O(1)$ time. Each enqueue involves at most one clean-up; and each dequeue involves at most one synchronization and one clean-up. Therefore, both the enqueue and dequeue procedures have a worst case time complexity of $O(1)$.

4.3. Preserving the ordering of real-time packets

ADQ can also be configured to preserve ordering between conformant and excess packets. However, enabling this feature may impact the excess throughput, as the ordering constraint will occasionally force the excess queue to "pass" on some transmission opportunities.

Synchronization guarantees that if an excess packet arrives earlier than a conformant packet, then that excess packet, if ever transmitted, will also be transmitted earlier. Therefore, preserving packet ordering only requires an additional mechanism to ensure that excess packets are never transmitted before conformant packets that preceded them. This requires a minor modification to the scheduler. Specifically, when an HOL excess packet e_1 is chosen for transmission, we check whether the HOL conformant packet c_1 arrived earlier than e_1. If c_1 did arrive earlier, we transmit c_1 instead. e_1 is typically deferred by one transmission opportunity. Revisiting the example of Fig. 2, only the transmission of packet e_6 and c_4 are switched if ordering needs to be preserved. At time $t = 0.9$, the scheduler has an opportunity to transmit e_6, but it transmits c_4 to enforce ordering. e_6 is transmitted later at $t = 1$.

5. Linux kernel Implementation of ADQ

Both versions of ADQ: JointQueue and ScfQueue, are implemented as Linux kernel queueing discipline modules [11]. The two modules use the same buffer management scheme and differ only in the choice of scheduling algorithms. The modules have been developed under Redhat 7.2, kernel version $2.4.2$, and are available from ADQ's homepage [12].

The ADQ queuing discipline controls the queue(s) associated with the output NIC through the *enqueue* and *dequeue* functions. As shown in Fig. 5, the *enqueue* function of ADQ first filters incoming packets based on *TOS* value. Then the corresponding lower layer enqueue function is called to insert the packet and update the queue structure. Scheduling is implemented in the *dequeue* function of ADQ. First, the *dequeue* function decides which packet to transmit based on the scheduling algorithms. Then the appropriate lower layer dequeue function is called to retrieve the selected packet from the queue. The lower layer enqueue and dequeue functions may call the "synchronization" procedure or the "clean-up" procedure if needed. [13] discusses details of the implementation to ensure an $O(1)$ complexity of the buffer management scheme.

6. Performance and Complexity

6.1. NS-2 simulation

To investigate how well ADQ performs against our initial design goals, we evaluate its performance against the following two criteria:

1. The throughput of conformant traffic. Ideally, this should be identical to the input rate of the conformant traffic, with all packets transmitted within their delay bound. The purpose is to check whether the presence of excess traffic affects the conformant traffic.
2. The total effective throughput of real-time traffic. This consists of all real-time packets, that were transmitted *prior to* their deadlines. We compare this value to the ideal target consisting of the sum of the conformant traffic input rate, and the fraction of the residual bandwidth the excess traffic is entitled to by the link-sharing policy. In our simulations, this fraction is 20%. The closer the effective throughput comes to the ideal throughput, the more excess packets are transmitted within the desired delay bound. This also verifies whether link-sharing is properly enforced between the excess and best-effort traffic (the best-effort traffic intensity is high enough to occupy any unused bandwidth).

We compare the performance of the two versions of ADQ with three schemes that have been commonly considered for supporting real-time traffic. (1) Priority queue with two FIFO queues.

The real-time traffic has priority over the best-effort traffic. (2) SCFQ with two FIFO queues. All real-time traffic is assigned to the same queue. The real-time queue is allocated sufficient bandwidth that meet the delay bound of the conformant traffic and the remainder is allocated to the best-effort queue. (3) SCFQ with three FIFO queues. Conformant and excess traffic are assigned to separate queues. The conformant traffic is allocated the required bandwidth, and the excess and the best-effort traffic share the remaining bandwidth based on a given ratio.

The simulation configuration is shown in Fig. 6. n_0 and n_1 are the sources of real-time and best-effort traffic, respectively. The real-time traffic is generated from MPEG-4 traces of the movie "Jurassic Park I"[14], while the best-effort traffic is generated from TCP traffic traces. Each traffic sources has an aggregate input rate of about 10 Mbps, and both are destined to node n_3. At node n_2, we simulate JointQueue and ScfQueue, as well as the three schemes mentioned above. The token bucket used to enforce the traffic contract of video traffic has a depth of 3000 bytes, and its rate is varied from 1 Mbps to 5 Mbps. Non-conformant packets are marked as excess. The delay bound is $10ms$. Given that conformant and excess packets are generated by the same source, JointQueue and ScfQueue are configured to preserve their ordering.

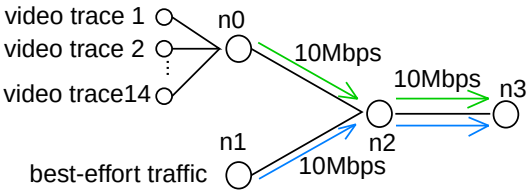

Figure 6. NS simulation network structure.

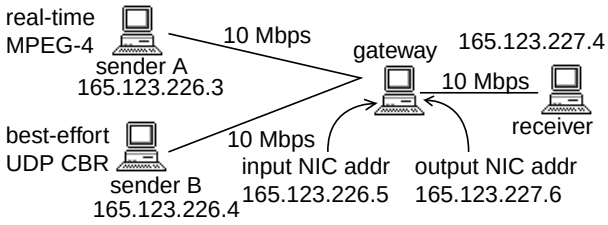

Figure 7. Experiment test bed setup.

The simulations showed that a priority queue scheme allows the real-time traffic to grab all the available bandwidth on link $(n2, n3)$ and starves the best-effort traffic. This clearly indicates that a priority-based scheme is not suitable. As shown in Fig. 8, when using SCFQ with two queues, the presence of excess traffic significantly affects the conformant traffic. This problem is eliminated when the excess and conformant traffic use separate queues. All conformant packets are now transmitted within their delay bounds without loss. However, the real-time effective throughput is nearly the same as the conformant traffic throughput. The 20% of the remaining bandwidth allocated to the excess traffic is wasted in transmitting expired packets. This highlights the need for buffer management if excess traffic is to be adequately supported.

The effectiveness of the buffer management of ADQ is clearly shown in Fig. 9. For both versions of ADQ, all conformant packets are transmitted within delay bounds and the excess traffic is able to achieve a meaningful throughput. For JointQueue, the total real-time throughput is very close to the ideal throughput. However, for ScfQueue, the excess traffic doesn't fully utilize the 20% remaining bandwidth it is entitled to. The reduction in the excess traffic throughput was expected, and is caused by the early synchronizations that SCFQ introduces when compared to SCED+. We have also tested the sensitivity of ADQ to different patterns of the conformant traffic [13]. JointQueue performs universally well; while the performance of ScfQueue degrades with a more bursty and intense conformant traffic.

6.2. Kernel Implementation Experiments

As shown by the simulation results, ADQ, and in particular JointQueue, fulfills our goal of effectively supporting excess real-time traffic. We investigate the cost of such improved service through benchmarking the Linux implementations of ADQ. The complexity is measured in the actual time spent in the enqueue and dequeue processes (excluding packet transmission time).

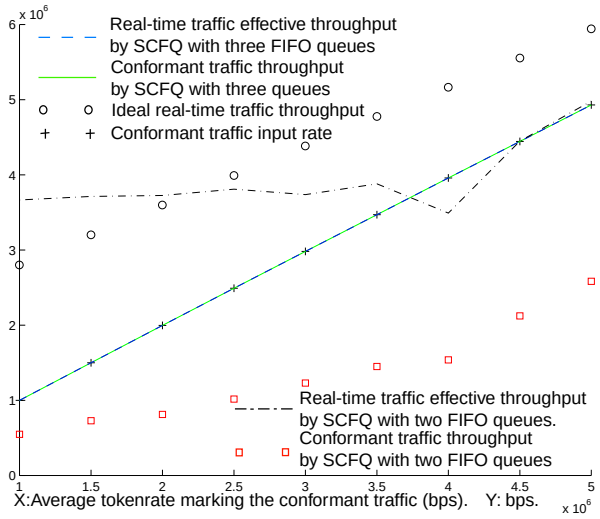

Figure 8. SCFQ

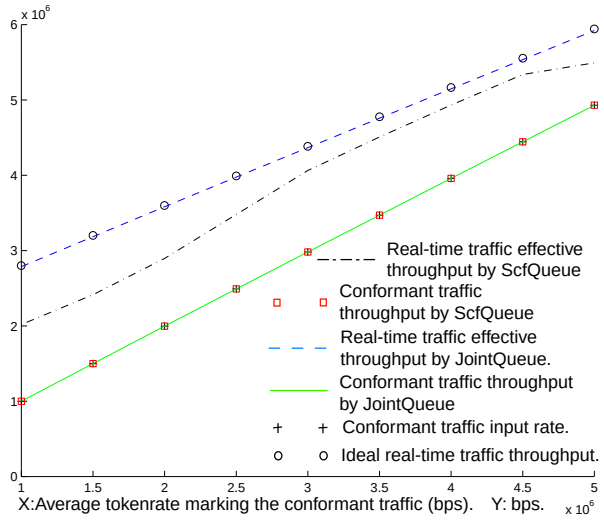

Figure 9. JointQueue and ScfQueue

The complexity of ADQ is then compared to that of the three schemes we mentioned before. In [13], ADQ's complexity is also investigated in terms of the number of operations and buffer accesses needed per transmitted packet, and it shows similar results.

As shown in Fig. 7, the ADQ modules and the other schemes are implemented on the gateway PC with a PIII 1 GHz Intel CPU, 256 MB RAM and Intel 10/100 express NICs. The two senders generate traffic destined to the receiver and traversing the gateway. Sender A is a MPEG4IP [15] streaming video server that generates MPEG-4 video traffic requested by the client on the receiver. Sender B uses MGEN [16] to generate CBR UDP traffic. The configurations are otherwise the same as those of the NS simulations, except for a lower real-time traffic volume.

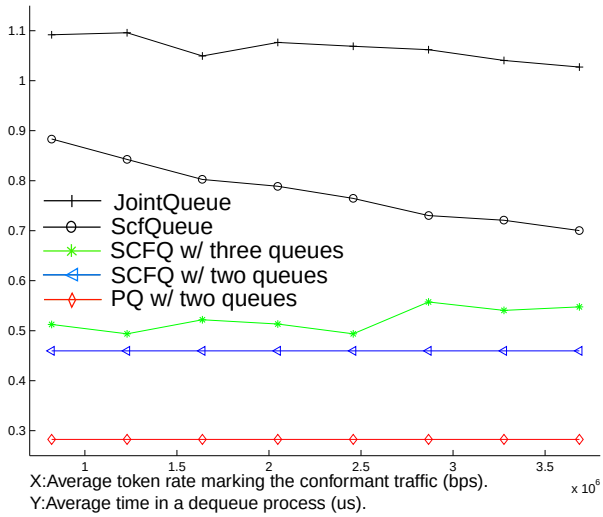

Figure 10. Average dequeue process time.

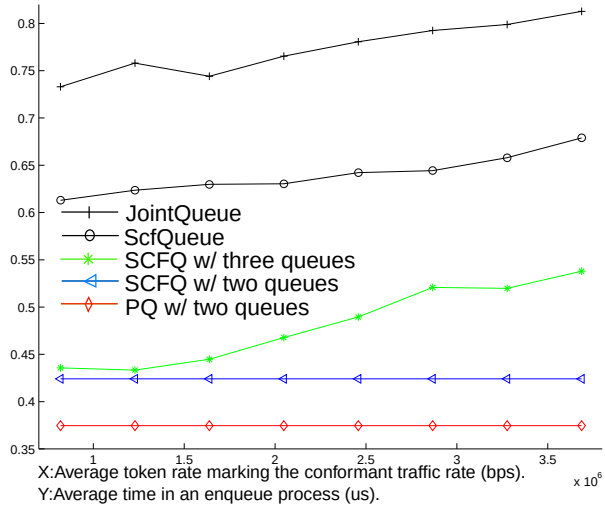

Figure 11. Average enqueue process time.

Fig. 11 and Fig. 10 show that JointQueue and ScfQueue require more enqueue and dequeue process time than the other schemes because of the additional complexity involved. When SCFQ is used, its early synchronizations translates into less clean-up procedures, thus less time, is usually needed in both enqueue and dequeue processes when JointQueue is used. Overall, although ADQ, JointQueue in particular, is expectedly more complex than the three simpler schemes we compare it to, the delta in complexity remains relatively small, i.e., from about

twice the dequeue process time and up to a 60% increase in the enqueue process time when compared to the three-queue version of SCFQ, the only possible contender.

7. Conclusion

In this paper, we proposed a new scheme, ADQ, that combines buffer management and scheduling to support both conformant and excess real-time traffic, while enforcing link-sharing between excess and best-effort traffic. The scheduling algorithm enforces delay guarantees and link-sharing, while the buffer management removes expired excess packets promptly from the queue. This ensures that no bandwidth is wasted on transmitting expired packets. ADQ can also preserve the ordering of real-time packets without significantly sacrificing overall performance.

We evaluated the performance and complexity of ADQ, by means of simulations and a Linux-based implementation. The results were compared to those of three simpler schemes. In all cases ADQ, in particular JointQueue, achieved our design goals, while the other schemes either penalized the best-effort traffic or wasted bandwidth by transmitting expired packets. We believe that our implementation demonstrates the feasibility of more flexible support for real-time traffic, which could facilitate the deployment of real-time applications.

REFERENCES

1. R. Guerin, S. Kamat, V. Peris, and R. Rajan, "Scalable QoS provision through buffer management," in *Proc. ACM SIGCOMM'98*, 1998.
2. J. Heinanen and R. Guerin, "A single rate three color marker," Request For Comments (Informational) RFC 2697, IETF, Sept 1999.
3. J. Heinanen and R. Guerin, "A two rate three color marker," Request For Comments (Informational) RFC 2698, IETF, Sept 1999.
4. P. Hurley, J.-Y. Le Boudec, and P. Thiran, "The asymmetric best-effort service," in *Proc. IEEE GLOBECOM'99*, 1999.
5. P. Hurley, M. Kara, J.-Y. Le Boudec, and P. Thiran, "ABE: Providing a low-delay service within best effort," *IEEE Network Magazine*, vol. 15, no. 3, May 2001.
6. V. Firoiu, X. Zhang, and Y. Guo, "Best effort differentiated services: Trade-off service differentiation for elastic applications," in *Proc. IEEE ICT 2001*, 2001.
7. R. L. Cruz, "SCED+: Efficient management of quality of service guarantees," in *Proc. IEEE INFOCOM'98*, 1998.
8. S. Golestani, "A self-clocked fair queuing scheme for broadband applications," in *Proc. IEEE INFOCOM'94*, 1994.
9. VINT Project, *The ns Manual*, UC Berkeley, LBL, USC/ISI, Xerox Parc, 2001.
10. I. Stoica, H. Zhang, and T. S. Eugene Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time and priority services," in *Proc. ACM SIGCOMM'97*, 1997.
11. iproute2, http://defiant.coinet.com/iproute2.
12. Y. Huang and P. Gupta, "ADQ," http://einstein.seas.upenn.edu/mnlab/softwares/ADQ.html.
13. Y. Huang, R. Guerin, and P. Gupta, "Supporting Excess Real-time Traffic with Active Drop Queue," Univ. of Pennsylvania, Tech. Rep., 2002
14. MPEG-4 and H. 263 video traces for network performance evaluation, http://www-tkn.ee.tu-berlin,de/research/trace/trace.html.
15. MPEG4IP, http://mpeg4ip.sourceforge.net.
16. MGEN, http://manimac.itd.nrl.navy.mil/MGEN/.