University of Pennsylvania
ScholarlyCommons

Departmental Papers (ESE) | Department of Electrical & Systems Engineering

July 2006

Supporting Excess Real-time Traffic with Active Drop Queue

Yaqing Huang
University of Pennsylvania, yaqingh@gmail.com

Roch A. Guérin
University of Pennsylvania, guerin@acm.org

Pranav Gupta
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/ese_papers

Recommended Citation

Yaqing Huang, Roch A. Guérin, and Pranav Gupta, "Supporting Excess Real-time Traffic with Active Drop Queue", . July 2006.

Copyright 2006 IEEE. To appear in *IEEE/ACM Transactions on Networking*, Volume 14, Issue 5, October 2006, pages 965-977.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/ese_papers/196
For more information, please contact libraryrepository@pobox.upenn.edu.

Supporting Excess Real-time Traffic with Active Drop Queue

Abstract
Real-time applications often stand to benefit from service guarantees, and in particular delay guarantees. However, most mechanisms that provide delay guarantees also hard-limit the amount of traffic the application can generate, i.e., to enforce to a traffic contract. This can be a significant constraint and interfere with the operation of many real-time applications. Our purpose in this paper is to propose and investigate solutions that overcome this limitation.We have four major goals: (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 for both guaranteeing delay bounds, while allowing the transmission of excess traffic. We evaluate the "cost" of our scheme by measuring the processing overhead of an actual implementation, and we investigate its performance by means of simulations using video traffic traces.

Keywords
QoS, Queue Management, Real-Time

Comments
Copyright 2006 IEEE. To appear in *IEEE/ACM Transactions on Networking*, Volume 14, Issue 5, October 2006, pages 965-977.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it

This journal article is available at ScholarlyCommons: http://repository.upenn.edu/ese_papers/196

Supporting Excess Real-time Traffic with Active Drop Queue

Yaqing Huang, Roch Guérin, *Fellow, IEEE,* and Pranav Gupta

***Abstract*— Real-time applications often stand to benefit from service guarantees, and in particular delay guarantees. However, most mechanisms that provide delay guarantees also hard-limit the amount of traffic the application can generate, i.e., to enforce to a traffic contract. This can be a significant constraint and interfere with the operation of many real-time applications. Our purpose in this paper is to propose and investigate solutions that overcome this limitation. We have four major goals: (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 for both guaranteeing delay bounds, while allowing the transmission of excess traffic. We evaluate the "cost" of our scheme by measuring the processing overhead of an actual implementation, and we investigate its performance by means of simulations using video traffic traces.**

I. Introduction

With the expansion of the Internet capacity, the demand for real-time multimedia applications such as streaming video, Internet games, VoIP, etc., has been increasing over the past few years. However, despite the ever increasing speed of Internet 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 traditional 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 those guarantees. Schedulers control access to transmission resources and buffer management is concerned with storage resources. Providing rate and delay guarantees typically calls for the use of both mechanisms, unless link speeds are high enough to ensure that delay guarantees are met even with a full buffer. In such cases, rate and delay guarantees can be provided through buffer management only, e.g., using a mechanism as described in [1]. 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 application traffic to which those guarantees apply, i.e., in the form of a traffic contract. In particular, existing mechanisms require that the application limit the amount of traffic it generates according to the traffic contract.

Traffic contracts are often in the form of token buckets, e.g., [2], [3], that specify a fixed transmission rate while allowing for short term rate variations through a "burst tolerance." Conformance is enforced by either dropping, reshaping, or marking as excess traffic, packets that violate the contract. 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 the specified burst tolerance.

Avoiding contract violations is difficult if not impossible for many real-time applications, as their traffic is difficult to predict. For example, video traffic can exhibit significant and prolonged changes in transmission rates as a function of scene characteristics. The range of fluctuations is more pronounced when using variable bit rate encoders, but is also present, albeit over a smaller time scale, when using so called "fixed rate" encoders. Similarly, voice traffic between two VoIP gateways varies based on both the number of simultaneous voice connections in progress, and the rate fluctuations within each connection. Dropping non-conformant traffic can result in substantial quality degradation, and reshaping it to a conforming stream usually introduces additional delays that are also detrimental to quality. As a result, allowing non-conformant real-time packets to enter the network by marking them as excess traffic is desirable. However, in order for such an option to be useful, it is important to ensure that excess real-time packets be transferred across the network within the desired delay bound. On the other hand, providing such a preferential delay treatment should not be done at the expense of other service classes.

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

1) Ensure that the zero loss and delay bounds guaranteed to conformant, real-time traffic are met,
2) Transmit as much excess, real-time traffic as possible and within the requested delay bound,
3) Enforce link-sharing between excess, real-time traffic and other service classes according to given proportions.
4) Support, as an optional feature, the ordering of real-time (conformant and excess) packets.

There have been several works that shared some of the above goals. The ABE proposal [4], [5] is one of the more

This work was supported by NSF grants ANI-9902943 and ITR-0085930, and by a grant from Nortel Networks. A shorter version of this work was presented at ITC'18, September, 2003.
The authors are with the School of Engineering of the University of Pennsylvania (email: {yaqing, guerin}@ee.upenn.edu; guptap@seas.upenn.edu

relevant[1]. ABE 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 and ensures some general link-sharing proportion between delay sensitive and throughput sensitive traffic. There are, however, significant differences between ABE and the mechanisms described in this paper. First, we target explicit service guarantees, i.e., delay bound, and assume the existence of traffic contracts for the conformant real-time traffic. Second, we distinguish between "conformant" and "excess", and focus on buffer management to remove *expired* excess real-time packets. Third, we explicitly control the level of link-sharing between excess real-time traffic and other service classes.

We call the set of mechanisms we propose to achieve the above goals, Active Drop Queue (ADQ). ADQ achieves the above goals through a combination of scheduling and buffer management mechanisms. In the rest of the paper, we denote the real-time packets that conform to the traffic contract as "conformant" traffic, and the real-time packets that exceed the traffic contract as "excess" traffic. We assume that the responsibility of marking packets that do not conform to the traffic contract as excess packets lies with the users, even if additional 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. Therefore, even without network verification, there's a strong incentive for users not to do so.

As mentioned earlier, ADQ relies on both scheduling and buffer management. Scheduling in ADQ is responsible for guaranteeing the delay bound of conformant traffic and enforcing link-sharing between excess traffic and other service classes. We present two versions of ADQ that represent a different trade-off between complexity and efficiency. The first version of ADQ involves two schedulers, SCED+ [7] for guaranteeing delay to the conformant traffic, and SCFQ [8] for enforcing link-sharing between the excess traffic and other service classes. The motivation for using this combination of two schedulers is that SCED+ is capable of delaying transmissions of conformant packets until the last moment, which offers more transmission opportunities for excess packets, and therefore a greater excess throughput. The second version of ADQ involves only the SCFQ scheduler that is responsible for enforcing both delay guarantees and link-sharing. The simpler SCFQ scheduler often transmits conformant packets earlier than their deadline, which can potentially lower the throughput of excess traffic.

Buffer management in ADQ ensures that only excess packets that have not yet missed their deadlines are transmitted. In other words, buffer management is primarily responsible for promptly removing "expired" packets. This is the key to maximizing excess traffic throughput. Failure to remove expired excess packets affects excess throughput in two ways. Expired excess packets unnecessarily occupy storage space, which may result in the unnecessary dropping of arriving excess packets. Expired excess packets also waste transmission opportunities if they are transmitted. The main challenges of the buffer management scheme are, therefore, to remove expired excess packets with the smallest possible overhead, preferably in $O(1)$ time [2], and do so while removing the smallest possible number of non-expired packets.

Another goal of ADQ, besides transmitting excess packets within their deadlines, is to optionally preserve the overall ordering of real-time packets. Preserving packet ordering is desirable when all real-time packets are generated by the same application. For example, this would be the case with a VBR video application that may occasionally exceed its contracted rate, and would, therefore, have to send some packets marked as excess. Such an application may be able to tolerate some amount of reordering, e. g. , through a playback buffer, but will operate more smoothly if ordering can be preserved. Conversely, preserving ordering between conformant and excess packets is unnecessary when they are generated by different users. One such example would be a VoIP gateway that has a certain amount of contracted bandwidth for voice sessions, and that instead of blocking new sessions when the bandwidth is fully used, let them proceed albeit as marked as excess. Because packets from excess sessions are independent of those of conformant sessions, ordering is not required. In such a setting, a potentially more important criterion is to ensure some level of fairness across excess flows. As we will explain later, the excess traffic in ADQ is handled by a FIFO queue with packet removal procedures that are unaware of flow identities. Therefore, assuming that VoIP flows are similar in nature, i.e., have similar rate and burstiness, ADQ should provide reasonable fairness across excess flows.

In the paper, we first describe the combinations of scheduling and buffer management mechanisms on which we rely, and then proceed to investigate their performance and complexity. Performance is evaluated by means of simulation using the NS simulator [9] with real traffic traces. MPEG-4 video trace files of the movie "Jurassic Park I" [10] are used as real-time traffic. Our investigation of the complexity of the different schemes considered is through benchmarking of an actual Linux kernel implementations of ADQ. We implemented two versions of ADQ with the two scheduler configurations mentioned before. The same buffer management scheme is used in both versions. The complexity of our schemes is compared to that of several simpler scheduling algorithms with simple FIFO queues. This benchmarking is done using MPEG-4 streaming video traffic generated by the MPEG4IP streaming video package [11] and UDP cross-traffic generated using MGEN [12].

Note that in addition to complexity and scalability, another factor to consider is the overall added cost that the deployment of ADQ imposes. Basically, ADQ calls for the use of a scheduler and a memory queue. The cost for its scheduler is no different from that of many other schemes used to support real-time applications, e.g. the ones we discuss in Section VI. In other words, introducing the kind of capabilities that ADQ offers, implies the addition of the necessary logic (s/w or h/w) for implementing scheduling decisions. ADQ also requires a

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

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

functionally more complex memory, i.e., it needs to be capable of removing blocks of consecutive packets from the head of the queue, which again represents an additional up-front cost compared to systems that do not offer such capabilities.

The rest of the paper is organized as follows. We first review the general structure of the ADQ algorithm in Section II. We then present the two scheduling schemes we investigated and the trade-off they represent in Section III. Details on the buffer management scheme are provided in Section IV. Section V introduces our Linux kernel implementation of ADQ, which was aimed at validating its feasibility. In section VI, we first evaluate the performance of ADQ through NS simulations. We then asses the complexity of ADQ by benchmarking the implementation described in Section V. Section VII concludes with a brief summary.

II. ADQ OVERVIEW

The ADQ algorithm, shown in Fig. 1, combines scheduling and buffer management mechanisms. ADQ relies on its scheduling algorithm to enforce delay guarantee for conformant traffic, and link-sharing between excess traffic and best-effort traffic. Ensuring that transmitted excess packets meet their delay bounds is primarily the responsibility of buffer management, which promptly removes expired excess packets from the excess queue. Preserving the overall ordering of real-time packets, when required, is supported by coordination between the scheduler and the buffer management mechanisms.

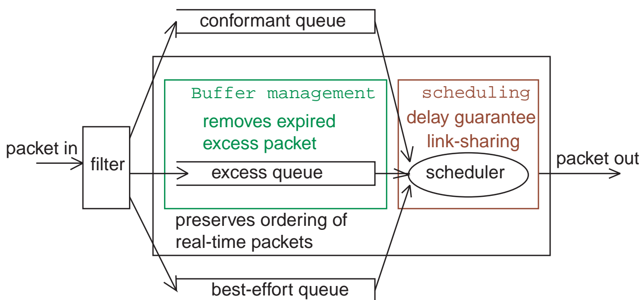

Fig. 1. ADQ general structure

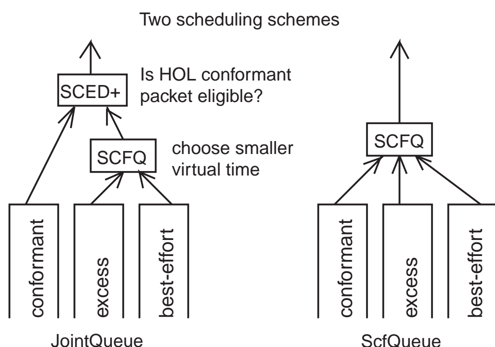

Fig. 2. Two scheduler configurations in ADQ

We identify three types of traffic: conformant, excess and best-effort and for each ADQ keeps a separate, logical, FIFO queue. Upon arrivals, packets are directed to the corresponding queues. A fixed amount of buffer is dedicated to all real-time traffic. Part of this buffer is assigned to the conformant queue for which we can compute the amount of buffer space needed to avoid packet losses. The remaining real-time buffer space is then allocated 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 later be removed from the buffer, either to make room for a newer arriving packet, or because they expire while in the queue. The best-effort is assigned a fixed amount of buffer as well. Its uses a simple FIFO queue, and arriving packets are dropped if the queue is full.

The general structure of our two scheduler configurations are shown in Fig. 2. 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, a deadline $d_k = t + \Delta$ is assigned to the packet. 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 they expire, so that we have as many opportunities as possible to transmit excess packets[3]. Our first scheduler scheme, a combination of SCED+ and SCFQ similar to [13], satisfies this requirement. In this scheme, the SCED+ scheduler is capable of delaying until the last moment the transmission of a conformant packet, and the remaining transmission opportunities are offered to either excess or best-effort packets, based on the SCFQ scheduler that controls the link-sharing policy. Our second scheme relies on a single SCFQ scheduler that may often transmits conformant packets earlier than necessary, and as a result decrease the number of transmission opportunities available to excess packets before they expire. Nevertheless, we explore such a scheme, because of its lower complexity, and evaluate its impact on the excess traffic throughput. We named the ADQ version that uses the two-scheduler scheme, "JointQueue," and the ADQ version using only the SCFQ scheduler, "ScfQueue."

Because the volume of excess traffic is unknown, scheduling alone is not sufficient to ensure that excess packets are transmitted within the desired delay bound. The key to supporting excess real-time traffic is to be able to identify and remove *expired* excess packets, instead of wasting buffer space and transmission opportunities on them. ADQ's buffer management addresses this issue through two main procedures: "synchronization" and "clean-up." The "synchronization" procedure is performed when transmitting a conformant packet. It locates and removes expired excess packets based on the ordering of arrival times of conformant and excess packets, and the fact that all real-time packets have the same delay bound. In most cases, "synchronization" alone is sufficient. However, a "clean-up" operation is occasionally needed when "synchronization" is not performed frequently enough. Both procedures are detailed in Section IV.

The example of Fig. 3 illustrates the typical behavior of

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

"JointQueue." For simplicity, the example assumes that all packets are 125 bytes, the delay bound for real-time packets is 0.5 ms, and it takes 0.1 ms to transmit each packet. We also assume that the excess and the best-effort traffic share the residual bandwidth equally.

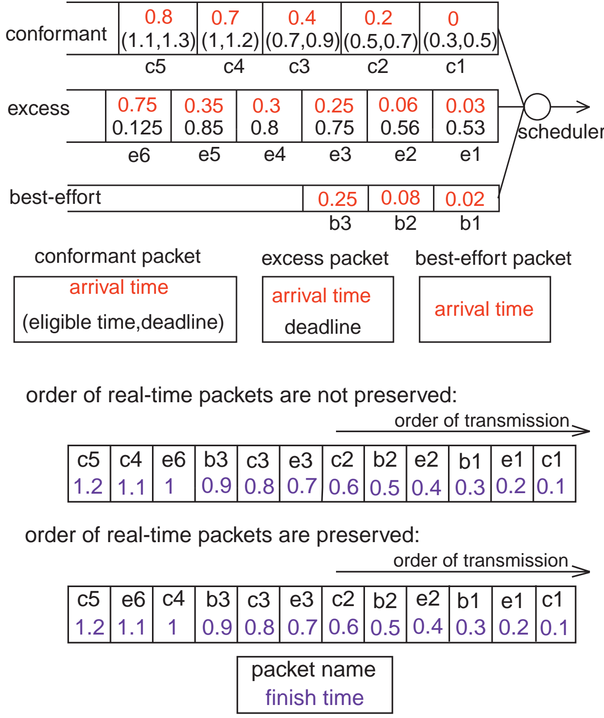

Fig. 3. ADQ example

III. Scheduling Algorithms

In this section, we discuss the two possible choices of scheduling schemes for ADQ.

A. JointQueue

In this scheme, SCED+ is used to provide delay bound and lossless performance to conformant traffic, and SCFQ is used to enforce link-sharing between excess and best-effort traffic. By assigning the proper service curve, SCED+ can schedule conformant packets as late as possible without violating their delay and loss requirements. Assuming a constant delay bound Δ and a token-bucket-modeled arrival curve f_c, a service curve S as shown in Fig. 4 satisfies our goal. We can then compute the buffer requirement of the conformant queue to avoid packet loss by Eq. (1) [7] and the eligibility time of a conformant packet c_k arriving at time t from Eq. (2) [7], so that packet c_k is transmitted as late as possible.

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

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

When the scheduler selects the next packet for transmission, it considers the head-of-line (HOL) conformant packet c_1. If c_1 is eligible, i.e., $eligible_1^c \leq current_time$, the scheduler pick c_1 for the next transmission. If the HOL conformant packet is not eligible, the scheduler chooses between the HOL excess or the HOL best-effort packets, whichever has a 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 so that the two queues share the residual transmission opportunities at a ratio of $e_ratio : b_ratio$. Eqs. (3) and (4) give $v(a_k^e)$ and $v(a_k^b)$, 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 respectively.

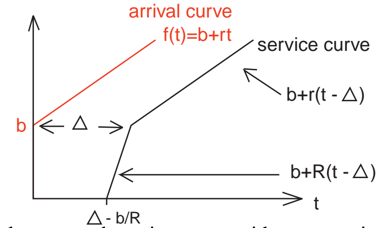

Fig. 4. Arrival curve and service curve with max service rate R.

$$v_k^e = \frac{L_k^e}{e_ratio} + \max\left(v(a_k^e), v_j^e\right). \quad (3)$$

$$v_k^b = \frac{L_k^b}{b_ratio} + \max\left(v(a_k^b), v_{k-1}^b\right). \quad (4)$$

B. ScfQueue

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

When using SCFQ, we first compute the rate r_c allocated to the conformant queue so as to guarantee the delay bound of conformant packets. Given r_c, we can then compute the amount of buffer needed by the conformant queue to avoid packet losses. Finally, we compute the rates r_e and r_b allocated to the excess and best-effort queues, respectively, to achieve the desired link-sharing level. Virtual times are then computed based on r_c, r_e and r_b, as in Eq. (3) and (4). SCFQ schedules all packets in order of their virtual times, and because the rate used to compute virtual times is based on a worst case arrival pattern that need not be applicable to all packets, this can result in early transmissions.

IV. Buffer Management

A key issue in supporting excess real-time packets is to be able to identify and remove from the buffer excess packets that have "expired". A straightforward solution is to check all enqueued packets after each transmission. However, this has a prohibitive $O(n)$ worst-case time complexity, where n is the number of packets in the excess queue. In this section,

```
Synchronization
{ IF syncIndex of HOL excess packet = index of
        HOL conformat packet;
  THEN tmpidx := segIndex of HOL excess packet;
        remove all excess packets before tmpidx;
  ENDIF
}
```

Fig. 5. Synchronization.

we describe a buffer management scheme, which can remove expired excess packets in $O(1)$ time complexity, which is desirable from a complexity standpoint. ADQ achieves this by relying on two procedures: “synchronization” and “clean-up”.

A. Synchronization

Synchronization removes expired excess packets based on the fact that real-time packets share a common delay bound, so that when a real-time packet p expires, all real-time packets arrived before p have also expired. If we assume that conformant packets are transmitted at or close to their deadlines, the transmission of a conformant packet c_k can then be used to “synchronize” the excess queue by removing from the excess queue all packets that arrived before c_k. This is the basic idea behind the “synchronization” procedure. The efficiency of the procedure depends on 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 conformant traffic, the less likely it is that conformant packets are transmitted much earlier than their deadlines. Similarly, a scheduler such as SCED+ that delays the transmission of conformant packets as long as possible, should perform better than SCFQ because it minimizes the number of early synchronizations.

The ability to associate a conformant packet with the excess packets that arrived before it can be accomplished relatively easily because we only need to associate it with those excess packets that arrived before it but after the preceding conformant packet. This effectively divides the excess queue into “segments”, synchronized by conformant packets. As shown in Fig. 6, segments are defined by having each excess packet contain a pointer, *syncIndex,* to the conformant packet synchronizing it, and a pointer, *segIndex*, 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. See Fig. 6 for an example of how the segment pointers are arranged and Fig. 5 for a summary of major the operations involved in a synchronization procedure.

Although the number of packets in the first segment is not $O(1)$ in general, the buffer space they use is continuous and lies between the head of the queue and the address specified by the *segIndex* of the HOL excess packet. Releasing such continuous buffer space can be done in $O(1)$ time.

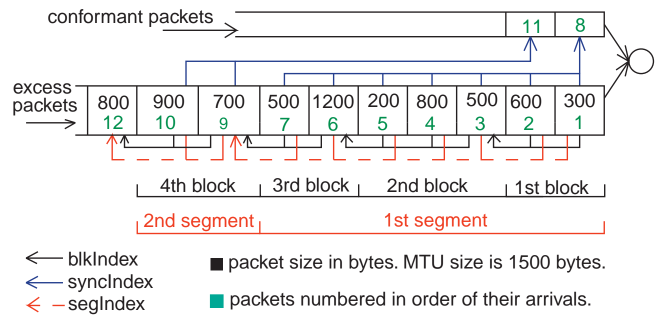

Fig. 6. Example of segments and blocks.

B. Clean-up

Under normal circumstances, namely when there is a regular stream of conformant packets, transmissions of conformant packets will trigger the synchronization procedure frequently enough to remove most expired excess packets in a timely manner. However, a large burst of excess packets or a lack of conformant packets for an extended 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. Another procedure, “clean-up,” is used to handle such cases.

Clean-up is called in either 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 excess 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 scheduler finding another expired HOL excess packet. In this second situation, the goal is to quickly find a non-expired excess packet to take advantage of the available transmission opportunity. This can always be accomplished by removing the first segment in the excess queue, 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 in the first segment. 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 present 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. Fig. 6 gives an example of an excess queue with block pointers. We assume that packets 1 to 4 are expired at the time the clean-up procedure is called. Note that the first block in Fig. 6 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. If this is the case, we proceed to remove the second block that is always complete. This always

```
Cleanup when enqueue
{ psize := size of coming excess packet;
  IF psize <= first block length + remaining
              excess queue space;
  THEN tmpidx := blkIndex of HOL excess packet;
       remove all excess packets before tmpidx;
  ELSE tmpidx := blkIndex of HOL excess packet;
       tmpidx := blkIndex of the excess packet
                 at tmpidx;
       remove all excess packets before tmpidx;
  ENDIF
}
```

Fig. 7. Clean-up triggered in an enqueue process

```
Cleanup when dequeue
{ blkidx := blkIndex of HOL excess packet;
  segidx := segIndex of HOL excess packet;
  IF blkidx - index of HOL excess packet >=
     segidx - index of HOL excess packet;
  THEN remove all excess packets before segidx;
  ELSE remove all excess packets before blkidx;
    IF the (new) HOL excess packet is expired;
    THEN tmpidx := index of HOL excess packet;
      IF tmpidx - index of HOL excess packet >=
         segidx - index of HOL excess packet;
      THEN remove all excess packets before
           segidx;
      ELSE remove all excess packets before
           tmpidx;
        IF the (new) HOL excess packet is
           expired;
        THEN remove all excess packets before
             segidx;
        ENDIF
      ENDIF
    ENDIF
  ENDIF
}
```

Fig. 8. Clean-up triggered in a dequeue process

guarantees enough space for the new packet. In this case, the clean-up procedure is called during an enqueue process. See Fig. 7 for a summary of the major operations involved. Assuming that the incoming excess packet is 1200 bytes in the example of Fig. 6. 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. These are packets 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 then default to removing the entire first segment, which guarantees that the new HOL packet is not expired. In this case, the clean-up procedure is called during a dequeue process, and the major operations involved are shown in Fig. 8. In the example of Fig. 6, the clean-up is completed when the first two blocks, namely, packets 1 to 5, have been removed. Packet 6 becomes the new HOL excess packet and is transmitted.

The clean-up procedure 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 operation involves at most one clean-up procedure; and each dequeue operation involves at most one synchronization or one clean-up procedure. Therefore, our buffer management scheme has a worst case time complexity of $O(1)$ per enqueue and dequeue operation. In Section VI, we investigate two different "clean-up" procedures. The first alternative always removes an entire segment without checking first if a less drastic measure would be sufficient. This clearly represents a simpler solution but at the cost of possibly lowering the excess throughput. The second alternative conducts a full search within the first segment to find the first non-expired excess packet. This, however comes at the cost of a higher, $O(\log n)$, complexity, where n is the number of packets in the first segment. We explore the trade-off between performance and complexity across clean-up procedures through simulations.

C. *Preserving the ordering of real-time packets*

ADQ can also be configured to preserve packet ordering between the conformant and excess queues when such a feature is desirable. Enabling this feature can 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 transmitted, is also transmitted earlier. Preserving packet ordering, therefore, only requires a mechanism to ensure that excess packets are never transmitted before conformant packets that arrived before them. This is achieved through a minor modification of 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 arrived earlier, we transmit c_1 instead of e_1 to enforce the ordering. This typically defers the transmission of e_1 by one transmission opportunity. Let us revisit the example of Fig. 3, but now requiring that packet ordering be preserved. The behavior of "JointQueue" is almost the same as when ordering was not preserved, except that the transmission of packet e_6 and c_4 are exchanged. At time $t = 0.9$, the scheduler has an opportunity to transmit e_6. However, to enforce ordering, it transmits c_4 before e_6, since c_4 arrived earlier. e_6 is transmitted at $t = 1$.

V. LINUX KERNEL IMPLEMENTATION OF ADQ

Both versions of ADQ: JointQueue and ScfQueue, have been implemented as Linux kernel queueing modules [14] under Redhat 7.2, kernel version 2.4.2, and are available from ADQ's homepage [15]. The two modules use the same buffer management scheme and differ only in the choice of scheduling algorithms.

The ADQ queuing discipline controls the queue(s) associated with the output network interface card (NIC) through the *enqueue* and *dequeue* functions. As shown in Fig. 9, the *enqueue* function of ADQ first filters the incoming packet based on the *TOS* value. Then the corresponding lower layer enqueue function is called to insert the packet and update queue structures accordingly. Scheduling algorithms are implemented in the *dequeue* function of ADQ. When it is time to transmit

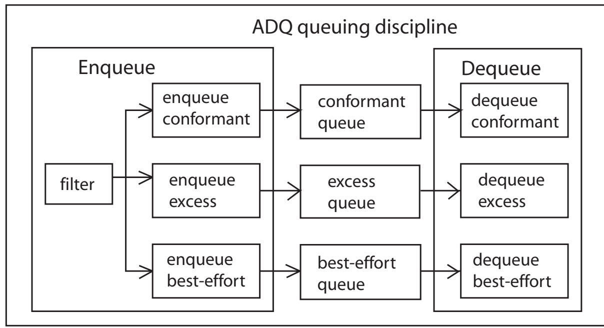

Fig. 9. General structure of ADQ queuing discipline.

a packet, the *dequeue* function of ADQ first decides which packet to transmit based on the scheduling algorithm(s) in use. The appropriate lower layer dequeue function is then called to remove the selected packet and update queue structures. The lower layer enqueue and dequeue functions may call the "synchronization" procedure or the "clean-up" procedure if needed as explained in the previous sections. We now briefly discuss some of the implementation choices made to ensure an $O(1)$ implementation of the buffer management scheme.

1) The data structure of the excess queues is a cyclic array. Thus removing multiple excess packets requires only an update of the head pointer and release of the buffer from the original head pointer to the new head pointer.
2) The *segIndex* and *syncIndex* pointers are stored with the excess packets. It might seem more natural to store them with the conformant packets, but this would require updating the pointers of all the conformant packets associated with the possibly many excess packets that are removed in one enqueue or dequeue operation, which is difficult to accomplish in $O(1)$ time. In addition, with all the pointers stored with the excess packets, the ADQ algorithm degrades to a normal SCFQ or a combined SCED+ and SCFQ scheduling scheme if there's a lack of excess packets for an extended period of time.
3) When a conformant packet c_k arrives, if there are one or more excess packets synchronized by c_k, we need to associate the *syncIndex* of these packets to c_k. To avoid updating all of these pointers upon the arrival of c_k, we use an integer to store the position of c_k, and have these excess packets point their *syncIndex* to that integer upon their arrivals. Thus we only need to store in the integer the index of c_k when it arrives. These integers are stored in a simple array. Similar techniques are used for *segIndex* and *blkIndex* as well.

VI. Performance and Complexity

A. 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, which should have all its packets transmitted within their delay bounds. The purpose of this criterion 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, both conformant and excess, 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 assigned to the excess traffic by the link-sharing policy. In our simulations, this fraction is set to 20%. The closer the effective throughput of real-time packets 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, in which the real-time traffic queue has priority over the best-effort traffic queue. (2) SCFQ with two FIFO queues. In this setting, all real-time traffic is assigned to the same queue. The real-time queue is allocated a rate that will provide satisfactory delay performance to the real-time traffic if its input traffic conforms to its traffic contract. The rest of bandwidth is allocated to the best-effort traffic queue. (3) SCFQ with three FIFO queues; two for conformant and excess traffic and the third for excess traffic. The conformant queue is allocated bandwidth based on its traffic contract and delay bound, and the remaining bandwidth is shared by the excess and best-effort queues according to the desired link-sharing ratio.

The performance of the JointQueue and ScfQueue variations of ADQ is compared to that of these three schemes. For that purpose, we consider several scenarios which correspond to different configurations for the real-time traffic sources. Scenario 1 considers a case where multiple video sources are multiplexed and controlled through a common token bucket. Because conformant and excess packets can originate from the same video source, this configuration requires that packet ordering be preserved. In scenario 2 and 4, conformant and excess video traffic is generated by two different sets of video sources. This removes the need to maintain packet ordering and allows the generation of different traffic patterns, in particular different bursts of conformant and excess traffic. Scenario 3 is similar to scenario 1 with the one difference that each video source is now controlled by its own token bucket. This configuration again calls for maintaining packet ordering, but generates very different traffic patterns in terms of mixture of conformant and excess traffic.

In scenario 1, we configure the network structure as shown in Fig. 10. Nodes n_0 and n_1 are the sources of the real-time traffic and the best-effort traffic, respectively. The real-time traffic is generated from MPEG-4 video trace files of the movie "Jurassic Park I," [10] while the best-effort traffic is generated from 10 long-lived TCP (FTP) flows. Each traffic node has an aggregate input rate of about 10 Mbps, and both

are destined to node n_3. Therefore, if left uncontrolled, each node can saturate the link connecting n_2 and n_3. At node n_2,we simulate JointQueue and ScfQueue, as well as the three other schemes that we use for comparison purposes.

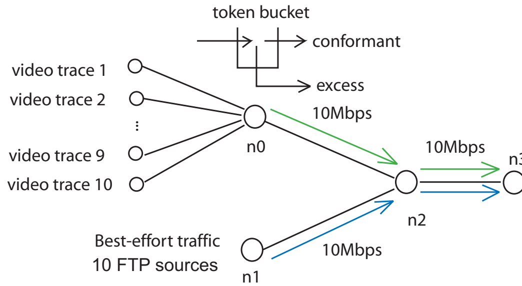

Fig. 10. NS simulation network structure for scenario 1.

In scenario 1, the token bucket marker used to enforce the traffic contract of the aggregated video traffic on node n_0 has a token bucket depth of 3000 bytes, and its token rate is varied from 0.5 Mbps to 4 Mbps. Non-conformant packets are marked as excess packets. The delay bound at node n_2 is set to be $20ms$. Given that conformant and excess packets are generated by the same source, JointQueue and ScfQueue are configured to preserve the ordering of real-time packets.

The simulations of scenario 1 showed that a priority queue scheme allows the real-time traffic to reach an effective throughput of about 9.74 Mbps out of a total bandwidth of 10 Mbps. However, this was achieved at the cost of starving the best-effort traffic, and clearly indicates that a priority-based scheme is not suitable. As shown in Fig. 11, when using SCFQ with only two queues, the presence of excess packets in the real-time queue results in significant delay violations and packet losses for the conformant traffic. This problem is eliminated when we introduce a third queue separating excess and conformant packets. All conformant packets are now transmitted within their delay bounds without loss. However, the effective throughput of real-time traffic is nearly the same as the throughput of conformant traffic only. The 20% of the remaining bandwidth allocated to the excess traffic is essentially 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. 12. For both versions of ADQ, all conformant packets are transmitted within delay bounds and the excess traffic is able to achieve a meaningful throughput. In addition, across all simulated scenarios ADQ consistently ensures that the conformant traffic is always transmitted within its delay bound, i.e., achieves its target throughput. As a result and in order to simplify the presentation, those curves will not be presented in subsequent figures. 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. This reduction in excess traffic throughput was expected, and is caused by the early synchronizations that SCFQ introduces when compared to SCED+.

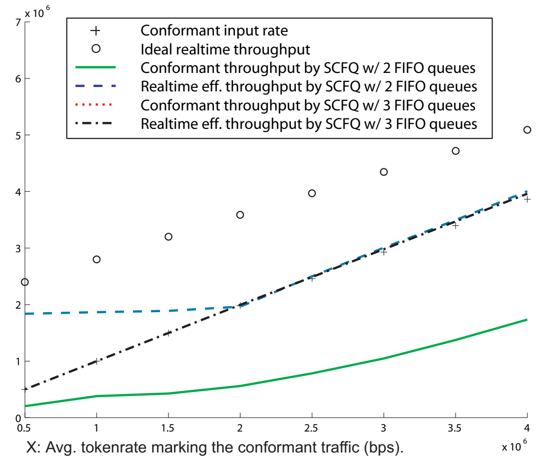

Fig. 11. SCFQ with FIFO queues in scenario 1.

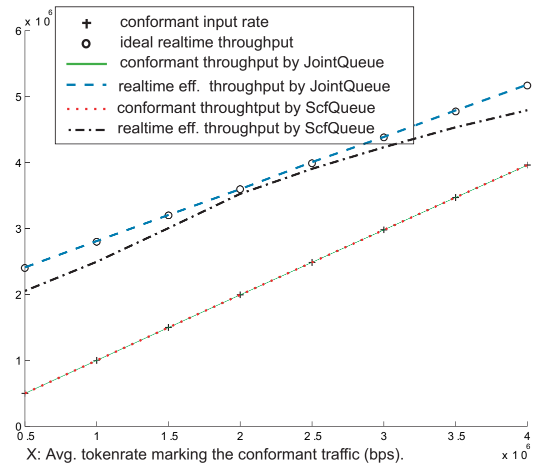

Fig. 12. JointQueue and ScfQueue in scenario 1.

Next, we investigate the impact of ADQ on the loss patterns experienced by the best-effort traffic, and in particular whether it affects TCP traffic to the point that it is not able to grab its fair share of bandwidth. Two different types of TCP traffic are used for that investigation. First, we use 10 long-lived FTP flows; and second we use 40 short-lived on-off TCP sources. The on-off sources all have the same exponential on-off pattern, with an average on period of 15 seconds and an average off period of 1 second to emulate average Internet flows [16]. The simulation results, shown in Fig. 13, demonstrate that ADQ doesn't negatively affect the ability of either type of TCP flows to achieve their fair bandwidth share. Furthermore, when ScfQueue is used, which achieves a lower excess traffic throughput than its ideal target, TCP traffic is actually able to grab the additional available bandwidth.

In scenario 2, we test the sensitivity of ADQ to different patterns of conformant traffic. The configuration, as shown in Fig. 14, is similar to that of scenario 1, except that the video sources are now split into two groups. The first group generates only conformant traffic through a token bucket configured to *drop* non-conformant packets. Two different token bucket

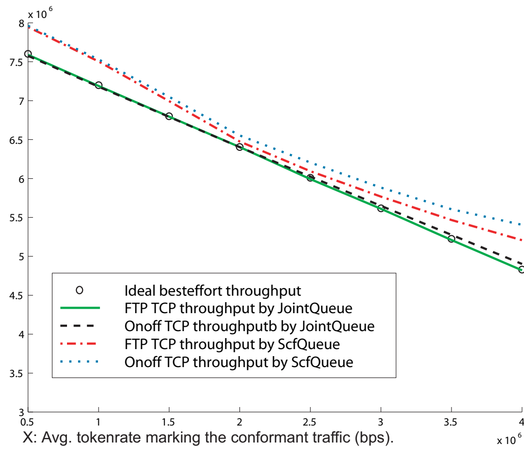

Fig. 13. Best-effort traffic throughput by JointQueue and ScfQueue in scenario 1.

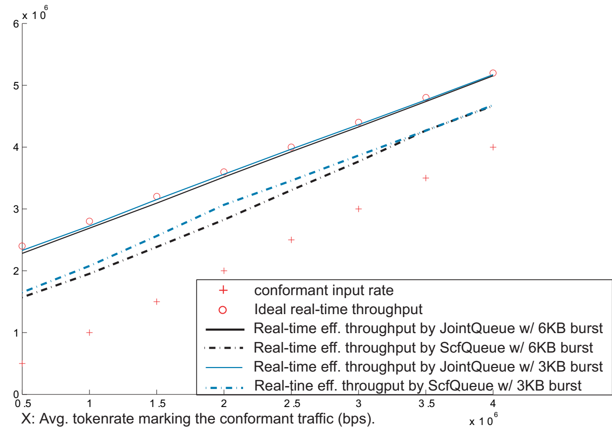

Fig. 15. JointQueue and ScfQueue in scenario 2 with burst tolerances of 3000 bytes and 6000 bytes.

depths of 3000 bytes and 6000 bytes were used together with token rates ranging from 0.5 Mbps to 4 Mbps to vary the intensity and burstiness of the conformant traffic. The second group of video sources is not regulated by token buckets, all of its traffic is marked as excess. Because conformant and excess packets are generated by different sources, packet ordering need not be preserved.

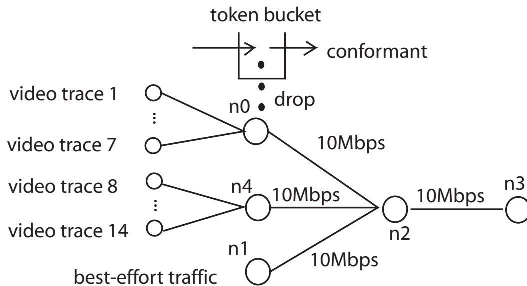

Fig. 14. NS simulation network structure for scenario 2.

The simulation results, shown in Fig. 15, show that JointQueue performs almost universally well under different traffic intensities and burstiness; while the performance of ScfQueue degrades slightly with a more bursty and intense conformant traffic. The degradation is again expected, and due to the increased amount of prematurely removed excess packets caused by early synchronizations.

In scenario 3, we test the performance of ADQ in a configuration similar to that of scenario 1, except that each video source is marked by its own token bucket with a burst tolerance of 1500 bytes. Note that, because the video traces used in the simulation are relatively bursty, using individual token buckets to mark each source results in a different total conformant input rate when compared to that of scenario 1, where the aggregated video traffic was marked using a single token bucket.

The simulation results of scenario 3 are shown in Fig. 17, and again demonstrate that ADQ is capable of providing service guarantees to the conformant traffic and of achieving

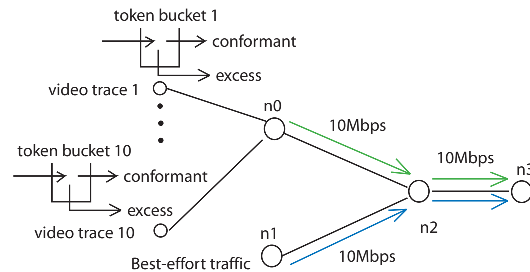

Fig. 16. NS simulation network structure for scenario 3.

meaningful excess traffic throughput. The performance of ADQ is only slightly worse than in scenario 1, where all the video sources shared a common token bucket marker. This is because in this scenario the maximum burst size that the scheduler is configured to handle in order to guarantee the delay bound of conformant traffic is 5 times that of scenario 1 (the burst size of each one of the 10 individual sources is simply added-up for a total of 15000 bytes). This larger worst-case burst size also makes it less likely that the scheduler actually sees such bursts. This in turn results in a larger number of early transmissions of conformant packets, and therefore early synchronization operations. Hence, a slight degradation in performance of ADQ was to be expected.

Scenario 4 allows us to investigate the impact of the clean-up procedure used in the dequeue processes on ADQ's performance. Its impact on the complexity of ADQ is investigated in the next section.

Using the configuration of scenario 2, we compare the performance of ADQ using the different clean-up procedures mentioned earlier, namely, our original clean-up procedure that attempts to strike a balance between efficiency and avoiding to remove non-expired excess packets, and both a more aggressive and a more conservative procedure. The more aggressive clean-up procedure directly removes the entire first segment whenever clean-up is triggered during dequeue. The more conservative (and more complex) clean-up procedure performs a full search of the first segment in order to precisely identify

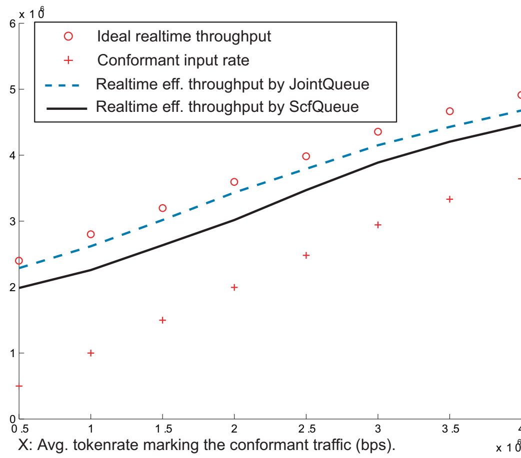

Fig. 17. JointQueue and ScfQueue in scenario 3.

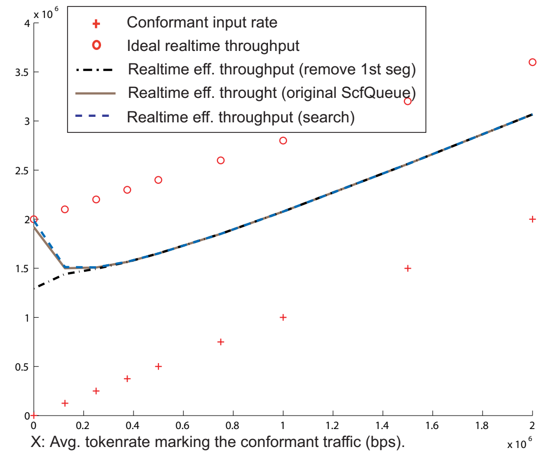

Fig. 19. Performance of ScfQueue with different clean-ups.

the first non-expired excess packet.

Simulations results are shown in Fig. 18 and 19 and indicate that for both JointQueue and ScfQueue the performance difference between the coarser clean-up and the full-search clean-up is only significant when the volume of conformant traffic is so small that the lack of synchronization causes a large number of clean-up procedures to be performed. It is worth noting that the original clean-up procedure of ADQ achieves similar performance as the full-search clean-up. Note also that in the case of ScfQueue, close to optimal performance can be achieved when the volume of conformant traffic is very low by using either of the two conservative clean-up procedures. This is in contrast to scenarios with a higher volume of conformant traffic, where irrespective of the clean-up procedure used, ScfQueue is not able to achieve a throughput equal to the maximum possible real-time effective throughput. This difference is primarily due to the fact that a small number of conformant packets also means few synchronizations, which are the main cause of the lower performance of ScfQueue since they are often performed earlier than necessary.

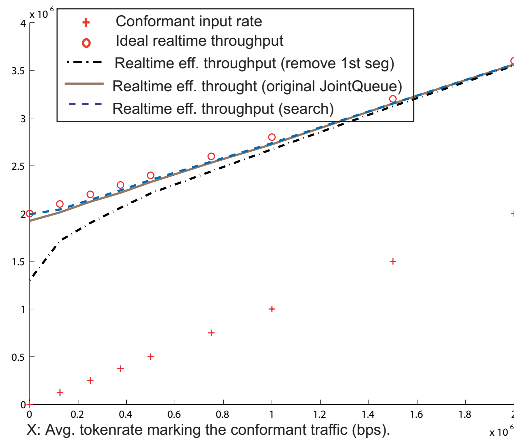

Fig. 18. Performance of JointQueue with different clean-ups.

B. 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 providing such improved services through benchmarking a Linux kernel implementation of JointQueue and ScfQueue. The complexity of ADQ is measured in terms of both the number of operations and buffer accesses needed per transmitted packet, and the actual time spent in the enqueue and dequeue processes (not including the packet transmission time). The complexity of JointQueue and ScfQueue is compared to that of the three schemes mentioned before, namely, the priority queue scheme and the two SCFQ schemes.

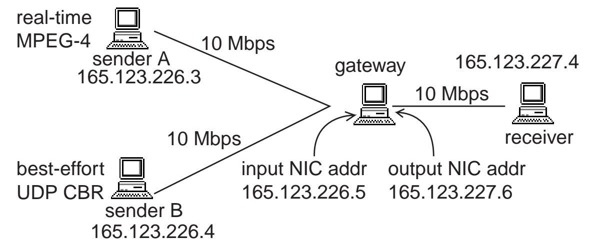

Fig. 20. Experiment test bed setup.

The testbed used in the experiments is shown in Fig. 20. 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 sender machines generate real-time and best-effort traffic destined to the receiver and traversing the gateway machine. Sender A is a MPEG4IP streaming video server [11] that generates MPEG-4 video traffic requested by the MPEG4IP client on the receiver machine. Sender B uses MGEN [12] to generate CBR UDP traffic. The experiment configurations are otherwise the same as those of the NS simulations of scenario 1, except for a lower real-time traffic volume. Note that the CBR traffic intensity of 10 Mbps ensures that there will be congestion on the shared link to the receiver.

We first evaluate the "raw" complexity involved in receiving

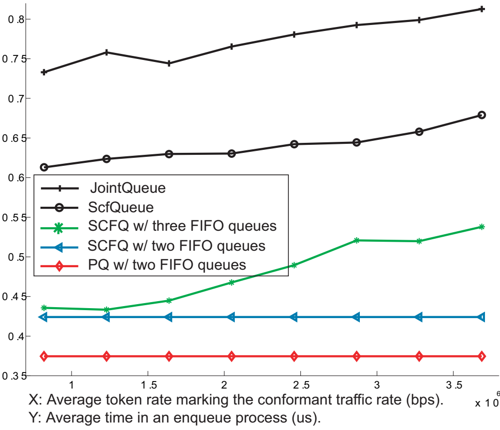

Fig. 21. Average time of an enqueue process.

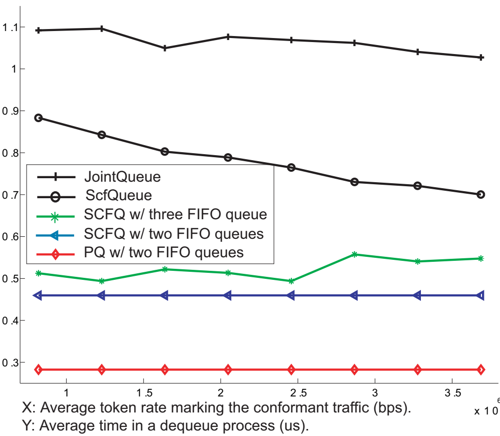

Fig. 22. Average time of a dequeue process.

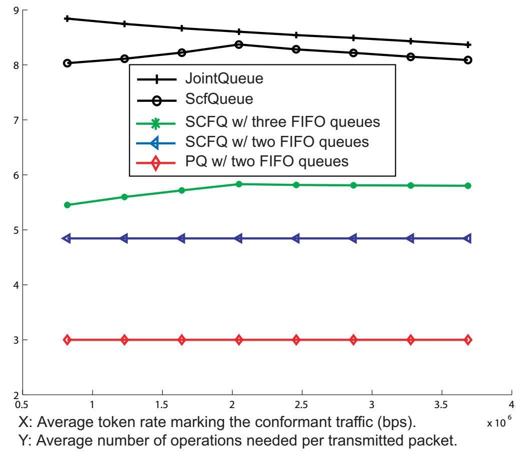

Fig. 23. Average number of operations per transmitted packet.

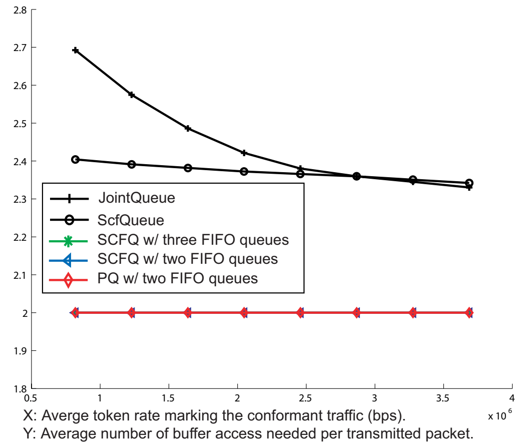

Fig. 24. Average number of buffer accesses per transmitted packet.

and transmitting packets using ADQ. For that purpose, we record the total time, in microseconds, taken by both an enqueue and a dequeue operation. Fig. 21 and Fig. 22 show that JointQueue and ScfQueue require larger enqueue and dequeue times than the other schemes because of the additional complexity involved. More clean-up procedures, thus a higher time, are usually needed when JointQueue is used.

Next, we extend our investigation to take into account the efficiency of the different schemes. In particular, some of the schemes involve doing work on packets that are either not transmitted or transmitted after their deadline has passed. Such work is obviously of little benefit. Therefore, in order to provide for a “fairer” comparison, we compute the complexity of each scheme averaged over packets whose transmission we deem successful. First, we measure complexity in terms of both the number of operations and memory accesses performed per transmitted packet (i.e., averaged over *all* transmitted packets). The results of Fig. 23 and Fig. 24 show that both versions of ADQ schemes have indeed greater complexity than the three simpler schemes, and JointQueue is indeed more complex than ScfQueue. But the differences are not huge.

Next, in order to get a better understanding of the costs associated with different types of packets, we measure the complexity required per transmitted conformant packet, excess packet, and best-effort packet separately. We compare this itemized complexity of JointQueue, which is the more complex version of ADQ, to that of the three-queue version of SCFQ. Fig. 25 and Fig. 26 show that, as expected, the higher complexity of ADQ is a result of the added processing steps required when handling conformant and excess packets. Specifically, the complexity associated with best-effort packets in ADQ is similar to that of the three-queue SCFQ version, as the handling of best-effort packet requires only scheduling of packets, which requires similar operations in the two systems. The handling of real-time (conformant and excess) packets is more costly because of the associated buffer management operations involved, with the handling of excess packets being the more costly of the two, as it often triggers clean-up procedures which are more complex than the synchronization procedure used when handling conformant packets.

We also investigate the difference in complexity of the different clean-up procedures mentioned earlier. As shown in Fig. 27 and Fig. 28, for both JointQueue and ScfQueue, the coarser clean-up procedure results in a smaller number of operations and buffer accesses, and a full-search clean-up procedure results in higher complexity. Again, the difference

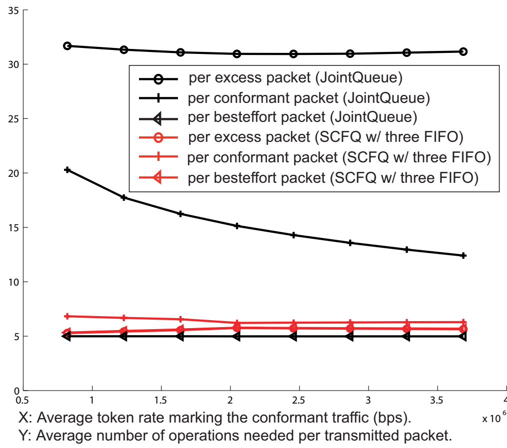

Fig. 25. Breakdown of avg number of operations per transmitted packet.

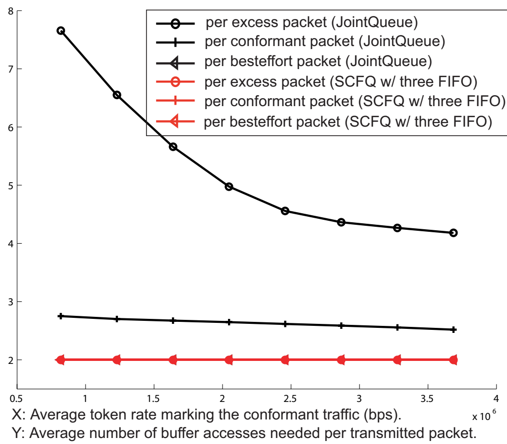

Fig. 26. Breakdown of avg. number of buffer accesses per transmitted packet.

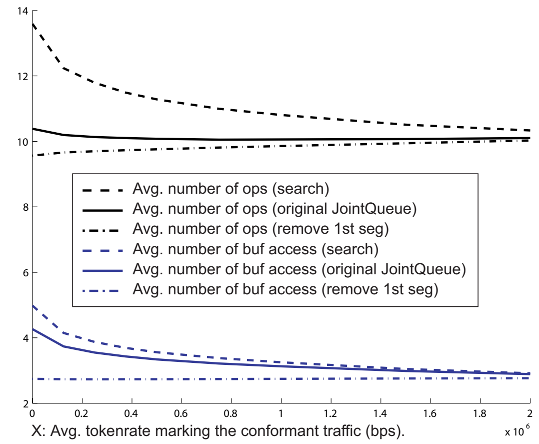

Fig. 27. Complexity difference of JointQueue caused by different clean-up procedures.

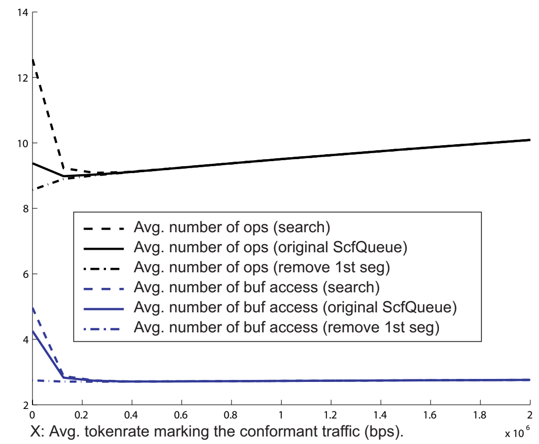

Fig. 28. Complexity difference of ScfQueue caused by different clean-up procedures.

in complexity diminishes as the volume of conformant traffic increases. In the case of ScfQueue, the difference diminishes even faster, as frequent early synchronizations further reduce the need for clean-up procedures. In other words, the choice of a clean-up procedure affects complexity only when the volume of conformant traffic is small. In this case, finer grain clean-up procedures are indeed more expensive, but as seen in Fig. 18 and Fig. 19, also have better performance. From that perspective, it appears that the original clean-up procedure of ADQ represents a reasonable trade-off between performance and complexity.

Overall, we note that although ADQ is obviously more complex than the three simpler schemes we compare it to, the delta in complexity remains relatively small, i.e., from about twice the number of packet operations to a 30% increase in the number of memory accesses when compared to the three-queue version of SCFQ, the only other possible contender.

VII. Conclusion

In this paper, we have proposed a new scheme, ADQ, that combines buffer management and scheduling mechanisms 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 is responsible for the timely removal of expired excess packets from the queue. This ensures that bandwidth is not wasted transmitting expired packets. ADQ can be configured to 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, and by comparing it to three simpler schemes. In all cases ADQ, particularly JointQueue, achieved our design goals, while the other schemes either penalized the best-effort traffic or wasted bandwidth by transmitting expired packets.

Additional simulations not reported in this paper have also shown that ADQ performs well even with low link bandwidth and a small number of flows. As a matter of fact, although increasing link bandwidth and the number of flows carried on a link does improve ADQ's performance, the magnitude of that improvement is small as even a few flow on a low bandwidth are able to achieve close to the maximum effective throughput. However, note that there is an intrinsic low bandwidth limit imposed by the need to guarantee hard delay bounds, namely,

the link bandwidth needs to large enough that the transmission time of a maximum-size packet is less than the target delay bound.

Overall, we believe that ADQ's design and its implementation demonstrate the feasibility of more flexible support for real-time traffic, which could benefit the deployment of real-time applications.

ACKNOWLEDGMENTS

We'd like to thank Mr. Zhi X. Yang for his help with the NS simulations of ADQ.

REFERENCES

[1] R. Guerin, S. Kamat, V. Peris, and R. Rajan, "Scalable QoS provision through buffer management," in *Proc. ACM SIGCOMM'98*, Vancouver, Canada, September 1998, pp. 29–40.

[2] J. Heinanen and R. Guerin, "A single rate three color marker," IETF, Request For Comments (Informational) RFC 2697, Sept 1999.

[3] J. Heinanen and R. Guerin, "A two rate three color marker," IETF, Request For Comments (Informational) RFC 2698, Sept 1999.

[4] P. Hurley, J.-Y. L. Boudec, and P. Thiran, "The asymmetric best-effort service," in *Proc. IEEE GLOBECOM'99*, Rio de Janeiro, Brazil, December 1999.

[5] P. Hurley, M. Kara, J.-Y. L. 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 differentiatioin for elastic applications," in *Proc. IEEE ICT 2001*, Bucharest, Romania, June 2001.

[7] R. L. Cruz, "SCED+: Efficient management of quality of service guarantees," in *Proc. IEEE INFOCOM'98*, vol. 2, San Francisco, USA, March 1998, pp. 625–634.

[8] S. Golestani, "A self-clocked fair queuing scheme for broadband applications," in *Proc. IEEE INFOCOM'94*, vol. 2, Toronto, Canada, June 1994, pp. 636–646.

[9] VINT Project. The network simulator - ns-2. [Online]. Available: http://www.isi.edu/nsnam/ns/

[10] F. H. P. Fitzek and M. Reisslein. MPEG-4 and H. 263 video traces for network performance evaluation. [Online]. Available: http://www-tkn.ee.tu-berlin.de/research/trace/trace.html

[11] D. Mackie, B. May, and A. M. Franquet. MPEG4IP. [Online]. Available: http://mpeg4ip.sourceforge.net

[12] NRL PROTEAN Research Group. MPEG4IP. [Online]. Available: http://manimac.itd.nrl.navy.mil/MGEN/

[13] I. Stoica, H. Zhang, and T. S. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time and priority services," in *Proc. ACM SIGCOMM'97*, Cannes, France, September 1997, pp. 249–262.

[14] (2001) Differentiated service on linux. [Online]. Available: http://diffserv.sourceforge.net/

[15] Y. Huang and P. Gupta. (2002) Adq. [Online]. Available: http://einstein.seas.upenn.edu/mnlab/softwares/ADQ.html

[16] K. Thompson, G. J. Miller, and R. Wilder, "Wide-area internet traffic patterns and characteristics," *IEEE Network Magazine*, vol. 11, no. 6, November 1997.

PLACE PHOTO HERE

Yaqing Huang received B.E. degree from Shanghai Jiao Tong University, Shanghai, China, in 1999, and M.S. and Ph.D. degrees in electrical engineering from the University of Pennsylvania, Philadelphia, in 2000 and 2005, respectively. Her research interest includes network quality of service, multimedia, network design and provisioning.

PLACE PHOTO HERE

Roch Guérin (F '01 / ACM '98) received an engineer degree from ENST, Paris, France, in 1983, and M.S. and Ph.D. degrees in EE from Caltech in 1984 and 1986, respectively. He joined the Electrical and Systems Engineering department of the University of Pennsylvania in 1998, as the Alfred Fitler Moore Professor of Telecommunications Networks. Before joining Penn, he spent over twelve years at the IBM T. J. Watson Research Center in a variety of technical and management positions. From 2001 to 2004 he was on partial leave from Penn, starting a company, Ipsum Networks, that pioneered the concept of protocol participation in managing IP networks. His research has been in the general area of networking, with a recent focus on developing networking solutions that are both lightweight and robust across a broad range of operating conditions. Dr. Guérin has been an editor for a variety of ACM and IEEE publications, and has been involved in the organization of many ACM and IEEE sponsored activities. In particular, he served as General Chair of the IEEE INFOCOM'98 conference, as Technical Program co-chair of the ACM SIGCOMM'2001 conference, and as General Chair of the ACM SIGCOMM'2005 conference. In 1994 he received an IBM Outstanding Innovation Award for his work on traffic management in the BroadBand Services Network Architecture. He has been on the Technical Advisory Board of France Telecom since 2001 and on that of Samsung Electronics in 2003-2004.

PLACE PHOTO HERE

Pranav Gupta attained his B.S.E. in Finance from the Wharton School of Business and his B.S.E. in Computer Science Engineering, as a student in the combined Jerome Fisher Program in Management and Technology. He also received an M.S.E. in Telecommunications Engineering from the University of Pennsylvania. Furthermore, Pranav was awarded the prestigious Thouron Scholarship to earn his MSc in Economics from the London School of Economics. Pranav is now the Chief Executive Officer of FarBPO, LLC, a global business process outsourcing company with offices in New York and New Delhi, India. FarBPO does financial analysis and research for leading investment banks, hedge funds, and other financial services companies in the U.S.