ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY
ELECTRICAL & COMPUTER ENGINEERING
DEPARTMENT

Incorporating Different Queuing Delay Bound in Active Drop Queue

By
Mehari Teklie

A thesis submitted to the school of Graduate studies of Addis Ababa University in partial fulfillment of the requirements for the degree of

Masters of Science in Electrical and Computer Engineering

(Computer Engineering)

October, 2008
Addis Ababa, Ethiopia

FACULTY OF TECHNOLOGY

INCORPORATING DIFFERENT QUEUING DELAY BOUND IN ACTIVE DROP QUEUE

By

Mehari Teklie

APPROVAL BY BOARD OF EXAMINERS

_____	_____
Chairman Dept. of Graduate Committee	Signature
_____	_____
Advisor	Signature
_____	_____
Internal Examiner	Signature
_____	_____
External Examiner	Signature

Declaration

I, the undersigned student, declare that this thesis work is my original work, has not been presented for a degree in this or any other universities, and all sources of materials used for the thesis work have been fully acknowledged.

Name: Mehari Teklie Signature: _____

Place: Addis Ababa

Date of submission: October, 2008

This thesis has been submitted for examination with my approval as a university advisor.

Dr. V.N.V Manoj Signature: _____

Advisor's Name

Acknowledgments

I would like to thank Dr. V.N.V Manoj, my advisor, for his continual support and encouragement throughout this work. He always gave full consideration to my ideas. Without him, this thesis would not have been possible.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ADQ	Active Drop Queue
AQM	Active Queue Management
ATM	Asynchronous Transfer Mode
D-CBT	Dynamic Class-Based Threshold
HOL	Head of line
ITU	International Telecommunication Union
MTU	Maximum Transmission Unit
NS2	Network Simulator version 2
QoS	Quality of Service
RED	Random Early Detection
SCED	Service Curve Earliest Deadline
SCFQ	Service Curve Fair Queuing
TSQ	Traffic Sensitive Quality of Service controller

Abstract

Delay sensitive applications, such as streaming video, Internet games, VoIP, etc., often sacrifice throughput for lower delay to obtain better quality. Active Drop Queue (ADQ) is an active Queue Management (AQM) technique, which guarantees queuing delay in the form of traffic contract with the assumption that all delay sensitive traffics requires the same constant queuing delay. It is a class based active queue management technique; i.e. conformant real time traffic, excess real time traffic and best-effort traffic.

This thesis presents an extension of Active Drop Queue to incorporate different queuing delay bound in the original ADQ. This feature is achieved by sub-classing each real-time traffic class (conformant and excess) base on queuing delay requirements with dynamic queue size for each sub-class. The Proposed ADQ successfully incorporates different queuing delay bound in the existing ADQ.

The performance and complexity of the proposed and the original ADQ is analyzed by simulation. It has been observed that the total number operation in the modified ADQ has increased, however the total number of excess packet drop is smaller than the original one as it uses use the available bandwidth effectively. Sub-classing reduces unnecessary removal of non expired excess packets due to early synchronization and lack of association of real time packets specific to application level.

Chapter One

1. Introduction

The Internet protocol architecture is based on a connectionless end-to-end packet service using the IP protocol. The advantages of its connectionless design, flexibility and robustness, have been amply demonstrated. However, these advantages are not without cost: careful design is required to provide good service under heavy load. In fact, lack of attention to the dynamics of packet forwarding can result in severe service degradation or "Internet meltdown". This phenomenon was first observed during the early growth phase of the Internet of the mid 1980s, and is technically called "congestion collapse" [RFC 2309].

Recently, especially demand for real-time multimedia applications such as streaming video, Internet games, VoIP, etc., has been increasing. Despite the ever increasing speed of Internet backbone links, access links often remain congested and, therefore, introduce throughput and delay limitations. Since time sensitive applications have relatively limited ability to adapt to fluctuations in network, these applications requires Quality of Service (QoS) and level of performance to be effective.

Today's internet multimedia application use application-level techniques to mitigate the effect of delay or loss. In addition to application-level techniques congestion control algorithms such as service differentiation, delay guarantee are required. Congestion control provides quality of service over the best effort networks.

Each router in the network uses queue management and scheduling as two classes of algorithms that are related to congestion control. The queue management algorithms try to control the length of packet queues by dropping packets when appropriate. Scheduling algorithms on the other

hand, determine which packet to drop next and which to send and also used to manage the allocation of bandwidth among flows.

1.1. Problem Statement

Most active queue management (AQM) techniques do not provide queuing delay guarantee for real-time applications, however they tried to guarantee delay by assigning priorities in a queue. Active Drop Queue (ADQ) is one of active queue management that was designed for delay guarantees by hard limiting the amount of traffic the application can generate, i.e., to conform to a traffic contract[20]. It is a class based active queue management technique; i.e. conformant real time traffic, excess real time traffic and best-effort traffic.

Even though ADQ is designed for guaranteeing delay for real-time application, an end to end maximum delay for real-time applications is different from each other. For example, end to end delay for VoIP best quality 50ms, good quality 100ms, and for live video 30ms etc [19] [18]. In addition to delay bound requirements, a real-time packet that traverse a number of hops requires the queuing delay to be minimum compared to the one that traverse a small number of hops so that it can be reach to the destination faster.

ADQ scheduling algorithm enforces delay guarantees based on the fixed delay bound; assuming that all real time applications require same queuing delay. It has no mechanism to consider different queuing delay requirement of real-time applications. In this thesis the impact of different delay bound in transmitting two or more real time applications with different delay bound on ADQ scheduling and buffer manager algorithm will be tested and solution will be analyzed.

1.2. Objective of this thesis

The main objective of this thesis is to incorporate different queuing delay bounds in the existing ADQ technique with out introducing more complexity on scheduling and buffer management algorithms.

To do this, the following tasks must be performed but not limited to:

- Different applications with different delay bound on the existing ADQ will be analyzed.
- The existing ADQ scheduling and buffer management algorithm will be modified to support different queuing delay bounds.
- The proposed ADQ and the original ADQ will be compared up on performance and complexity.

1.3. Organization of the Thesis

This thesis is organized as follows. Chapter 2 reviews the background of real-time multimedia communication structure over an internet and previous AQM research for real-time applications. Definitions of measurement criteria are presented, various types of delays in real-time multimedia transmission, and a summary of the related work is given. Chapter 3 presents the existing ADQ scheduling and buffer management in detail. Chapter 4 presents the proposed version of ADQ. It discusses different issues on incorporating different delay bounds. Chapter 5 introduces the simulation methodology deployed in this investigation with the simulation tool NS-2, and described experimental procedures used throughout this study. The conclusions of this thesis and the future work are presented in Chapter 6.

Chapter Two

2. Literature Review

This chapter provides definitions and background information required for a better understanding of AQM and congestion control issues. Topics discussed include structure of real-time multimedia over IP networks and review of pertinent previous attempts to study AQM for real time communication researches.

2.1. Structure of Real-time Multimedia over an Internet

Figure 2.1 briefly introduces the three main parts of voice/video service over an IP network. Briefly, it is a process of transforming analog signals to digital signals, transmitting digital audio/video signals across an IP network in the form of packets, and the reassembly and playback of these signals at the receiver. Firstly, at the sender, the input audio/video signals are digitized and compressed by the encoder in units as frames, then frames are packetized to form the payload of a packet and the headers (e.g. IP/UDP/RTP headers) are added to the payload to form a packet. Next the packets are transmitted over the network at regular intervals (the packet interval or packetization delay) and may suffer different network impairments (e.g. packet loss, delay and jitter) within IP networks during transmission. At the receiver, the playout buffer compensates for network jitter at the cost of further delay (buffer delay), then the packet headers are stripped off and voice/video frames are extracted from the payload by the depacketizer. The de-jittered voice/video frames are decoded to recover the analog audio/video. [19]

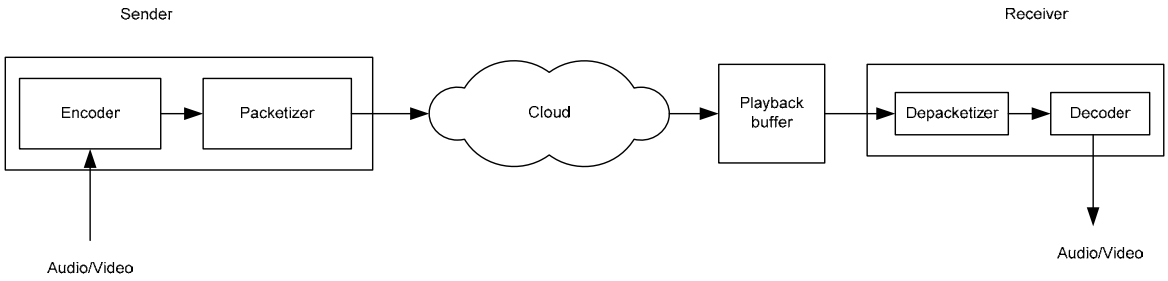

Figure 2.1 Structure of real-time multimedia over an Internet.

As mentioned above, different network impairments will occur due to the behavior of the best-effort IP network. To maintain the quality of received voice/video, much research has investigated how to provide QoS mechanisms suitable for these voice/video services in a packet-switched network.

2.2. QoS Requirements of Real-time Multimedia Applications

Quality of Service (QoS) refers to the capability of a network to provide better service to selected network traffic over various technologies, including Frame Relay, Asynchronous Transfer Mode (ATM), Ethernet and 802.1 networks, and IP-routed networks that may use any or all of these underlying technologies. The primary goal of QoS is to provide priority including dedicated bandwidth, controlled jitter and latency (required by some real-time and interactive traffic), and improved loss characteristics. Also important is making sure that providing priority for one or more flows does not make other flows fail [21].

The critical issue for supporting real-time multimedia communication is how to provide end-to-end performance guarantee and at the same time how to use system resources (network bandwidth, CPU cycles, storage etc.) efficiently by taking advantage of bursty and error tolerance nature of continuous media. The main factors which influence the quality of multimedia include delay, jitter and packet loss, which are the three main parameters that determine the QoS of real-time multimedia.

2.2.1. Delays in transmission

End to end delay is the one-way transmission time of a packet from source to destination. This end to end delay includes delay introduced at sending end, network and receiving end caused by various process during transmission.

Coding Delay

Codec delay is caused by the actual process of encoding and decoding multimedia samples and placing/recovering the samples into/out of a packet. Table 2-1 gives encoding and decoding delays for several typical voice codecs standardized by the International Telecommunication Union (ITU). As for video, the codec delay may be in the order of tens of milliseconds [19].

Coding standard	Compression algorithm	Bit rate (Kbps)	Encoding delay (ms)	Typical decoding delay (ms)	Total coding delay (ms)
G.711	PCM	64	0	0	0
G.729	CS-ACELP	8	15	7.5	22.5
G.723.1	ACELP	5.3/6.3	37.5	18.75	56.35

Table 2.1 Encoding and decoding delays for voice packet [19][22]

Packetization Delay

Packetization delay is the time taken to fill a packet payload with encoded (compressed) multimedia. This delay is a function of the sample block size required by the voice codec and the number of blocks placed in a packet.

See Table 2.2 for packetization delays associated with common packet voice codecs.

Coder	Bandwidth	Payload Size (Bytes)	Packetization Delay (ms)	Total Round-trip Delay (ms)
PCM, G.711	64Kbps	160	20	40
ADPCM, G.726	32Kbps	80	20	40
CS-ACELP, G.726	8.0Kbps	20	20	40
MP-MLQ, G.723.1	6.3Kbps	24	24	48
MP-ACELP, G.723.1	5.3Kbps	20	30	120

Table 2.2 Packetization Delays Associated with common Codecs[22].

Serialization Delay

Serialization delay is the fixed delay required to transmit all packet's bit onto the network interface, and it is directly related to the bandwidth available on the network interface. Depending on packet sizes and network interface speeds, typical serialization delays range from negligible to 5 ms. [22].

Propagation Delay

The time required to propagate from the beginning of the link to the next node is the propagation delay. The propagation delay is related to the physical medium of the link and its propagation speed, e.g. fiber optics, twisted –pair copper wire, and the distance between two nodes. The propagation delay is the distance between two nodes divided by the propagation speed of the link. In wide-area networks, propagation delay is typically of the order of milliseconds [19].

Queuing Delay

Queuing delay is the time the packet stays in a queue before transmission. Queuing delay occurs when the queue service rate is not fast enough to serve all the incoming packets from different sources using this queue, therefore, the packets accumulate in the queue and wait for service.

It is variable and is dependent on the trunk speed and the amount of traffic in an output queue; however all other delays are fixed i.e. they are configurable or can be easily calculated. Since real-time multimedia applications are sensitive to delay and needs delay guarantee, mechanisms should be devised to reduce this queuing delay. It is the main focus this thesis guarantee delay for real time multimedia.

Playback Delay or Jitter Compensation Delay

Jitter is the variation in the arrival time of packets introduced by the variable delays in the network. To allow for variable packet arrival times and to achieve a steady stream of packets, the receiver holds packets in a buffer for a while before playing them. This holding time causes further delay for the application.

2.2.2. Jitter

Jitter is a measure of smoothness of the audio/video playback. It is related to the variance of packet delays. A large buffer can hold enough frames to allow the frame with the longest delay to arrive, to reduce playback jitter. However, this increases latency and may not be desirable in real-time and interactive applications.

2.2.3. Packet Drop or Loss

Packet lost or drop occurs when the network is congested or there is no available space for the arriving packet in the queue. Further more for real-time traffic, loss also occurs when a packet arrives at its destination but is too late to be played out as part of a continuous bit stream. Although real-time applications can tolerate certain amount of loss, packet loss can cause severe damage to QoS for real-time audio and video traffic.

2.3. Active Queue Management

Because of the bursty nature of multimedia traffic, sometimes the amount of traffic exceeds the speed of a link and may result in congestion. Proper congestion avoidance mechanisms are required to prevent and avoid congestion. Since 1988 a number of research have been done on internet dynamics ; it has become clear that congestion avoidance mechanisms are not sufficient to provide good service in all circumstances, even though they are necessary and powerful [RFC2309].

Queue management and Scheduling are two main classes of router algorithm related to congestion control. Queue management algorithms manage the length of packet queues by dropping packets when necessary or appropriate, while scheduling algorithms determine which packet to send next and are used primarily to manage the allocation of bandwidth among flows. While these two router mechanisms are closely related, they address rather different performance issues.

2.4. Previous Related Work

Several Active Queue Management schemes have been proposed in recent literature to provide better real time communication. Some of the recently proposed AQM discussed below.

Shengquan Wang, Dong Xuan, Riccardo Bettati, and Wei Zhao in [15] propose and analyze a methodology for providing absolute differentiated services for real-time applications. This paper presents method for delay guaranteeing for real time application by assigning priorities on a class-by-class basis using static priority scheduler. Unlike traditional priority scheduler in differentiated services networks assign priorities on a class-by-class basis, with the same priority for each class on each router, it allows different routers to assign different priorities to classes, achieves significantly higher utilization bounds.

Jae Won Chung in [3] presents an extension of Random Early Detection (RED) to improve the multimedia performance on internet called Dynamic Class-Based Threshold (D-CBT) active queue management. D-CBT categorizes flows into TCP, tagged UDP and untagged UDP classes to protect flow-controlled multimedia flows from unresponsive multimedia flows, and encourage multimedia applications to use congestion avoidance mechanisms, which may be different than those of TCP. In determining the fair UDP thresholds, D-CBT calculates the fair average output buffer share of the tagged UDP class from the average queue length that is maintained by RED, and that of untagged UDP class from the RED's minimum threshold (plus a small allowance). Therefore, they are allowed to use the impending congestion state queue buffers (i.e. red_avg – red_min when red_avg > red_min) up to their fair share of the average. However, unresponsive (Untagged) flows, which have no ability (small fraction) to respond to network congestion, are not allowed to use the impending state queue buffers at impending congestion.

Paul Hurley and Jean-Yves Le Boudec in [16] propose on providing low delay for interactive application at the expense of fewer throughputs. The thesis presents a marking technique for identifying delay sensitivity of the packets. Best effort packet is marked as either green or blue. Green packets are guaranteed a low bounded delay in every router. In exchange, green packets are more likely to be dropped (or marked using congestion notification) during periods of congestion than blue packets. Choice and marking of packets is done by the application, real time packets will be marked green and non delay sensitive such as bulk data transfer will marked as blue.

The thesis presents router requirements that aim at enforcing benefits for all types of traffic, namely that green traffic achieves low delay and blue traffic receives at least as much throughput as it would in a flat (legacy) best effort network. The paper also presents the implementation analysis based on a new scheduling method called duplicate scheduling with deadlines. Blue

packets are always served at the latest their deadline permits subject to work conservation. Green packets are served in the meantime if they have been in the queue for less than *d* s (deadline), and dropped otherwise.

Traffic Sensitive Active Queue management based on delay hints (indicating the relative importance of delay versus throughput) presented in [8]. This paper presents a new QoS mechanism called the Traffic Sensitive Quality of Service controller (TSQ) that provides better delay performance for delay sensitive applications and higher throughput for throughput sensitive applications.

A queue management that guarantees a delay for a certain amount of real traffic based on scheduling and buffer management called Active Drop Queue proposed in [20]. It guarantees a delay by hard-limiting the amount of traffic the application can generate, i.e., to enforce to a traffic contract.

This thesis work is an extension ADQ, and the next chapter will discuss ADQ in detail.

Chapter Three

3. Active Drop Queue

As briefly discussed in chapter 1, the main objective of active drop queue is service guaranteeing in the form of delay guarantee, with explicit identification of the application traffic to which those guarantee apply, i.e., in the form of a traffic contract. It is based on traffic classes, it ensures that the delay bounds guaranteed to real time conformant class are met, transmit as much excess real time packets as possible with in the required delay bound and enforce link sharing between excess real time traffic and other service classes. The following section discusses types of traffic classes in active drop queue.

3.1. Traffic Classes in Active Drop Queue

As the main objective of ADQ guaranteeing a delay for specific class of traffic, traffic classes in Active Drop Queue are divided in to two big categories; real time and non real time (best-effort) traffic. Class of real time traffic that are contracted to be transmitted with in the required delay bound are classified it to conformant real time class. And real time packets that do not conform to the traffic contract are classified in to excess packets.

Marking of real time packets as conformant and excess relies on the application. The application is responsible in assigning a packet as a conformant or excess depends on there importance. A real-time packet which is considered as important will be marked as conformant and the others as excess. For example in transmitting multiple descriptors streaming, only one of the descriptor contains the basic quality of the original picture and others will contains extra quality of the picture. If all of the descriptors received at the destination, the original image can reconstructed. If some or all of the descriptors except that contains the basic quality are lost, the original image can be reconstructed with acceptable

quality. Figure 3.1 illustrates this concept. Figure 3.1 (a) shows when only descriptors that contains basic quality received at the destination. Figure 3.1 (b) shows when some of the descriptor that contains extra information received and figure 3.1(c) shows when all the descriptors received.

A. Only Basic quality received

B. Basic and Some of extra information

C. All descriptor received.

Figure 3.1 Multiple Descriptor image

3.2. ADQ Algorithm

The ADQ algorithm, shown in Fig. 3.2, combines scheduling and buffer management. The scheduling algorithm is responsible on:

- Enforcing delay guarantees to the conformant traffic.
- Link sharing between Excess and best effort traffic.

On the other hand the buffer management is responsible on

- Ensuring that transmitted excess packets meet their delay bound.
- Removing expired excess packet from the excess queue.

Preserving the ordering of real-time (conformant and excess) packets, when required, is supported by relying on both the scheduler and buffer management.

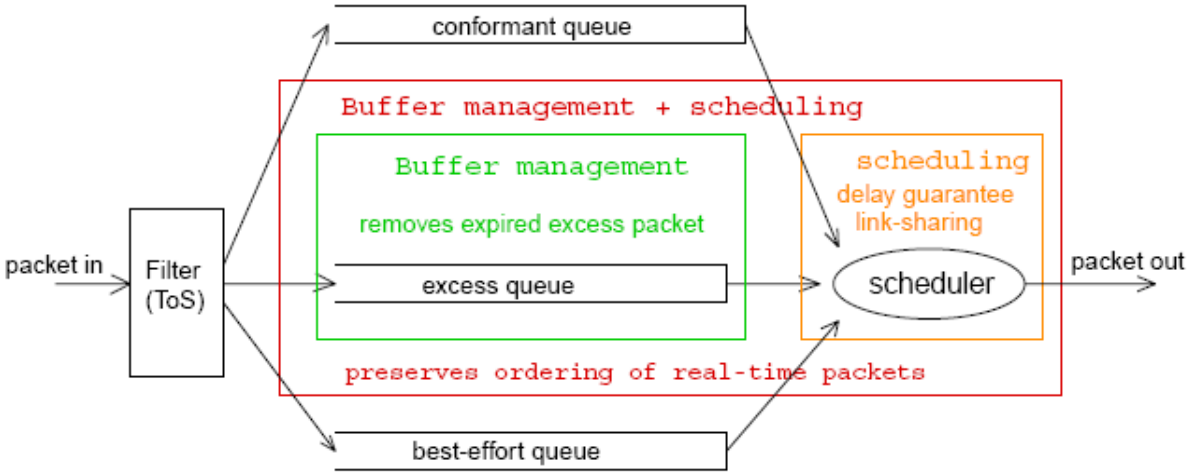

Figure 3.2 General Structure of ADQ

As shown in general structure above, ADQ keeps three separate, logical, FIFO (first in first out) queue for conformant, excess and best-effort traffics. Upon arrivals, packets are filtered to 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 the amount of buffer space needed to avoid packet losses can be computed. The remaining real-time buffer space is then allocated to the excess queue. When the incoming excess packet finds the excess queue full, it will not be dropped on arrival. Older packets from head of excess packets will be removed to make a room for the arriving packets.

Assuming that both conformant and excess packets generated from the same application , they require the same constant delay Δ, i.e. the maximum time a real-time packet is allowed to stay in buffer. 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 [20].

3.2.1. ADQ Scheduling Algorithm

There are two schemes of ADQ scheduling algorithm; JointQueue and ScfQueue. JointQueue is a combination of SCED+ [10] and SCFQ [2]. To get more transmission opportunities for excess packet, the jointQueue scheduler delays transmission of conformant packet until the last moment 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. On the other hand ScfQueue is a single and simpler scheduler, such as SCFQ, often transmits conformant packets earlier than necessary, which may decrease the transmission opportunities available to excess packets before they expire, thus reducing the excess traffic throughput[20].

JointQueue

Assuming a constant delay bound Δ and a token-bucket-modeled arrival curve *f*c, a service curve S as shown in Fig. 3.3 satisfies the main goal of jointQueue scheduler [20] . In this scheme, SCED+ is used to provide delay and lossless performance to conformant packets, i.e. the arrival rate should be equal to the service rate of conformant packet, and SCFQ is used to provide link-sharing between excess and best-effort packets. By assigning the proper service curves, SCED+ can schedule conformant packets as late as possible, while still meeting their delay and loss requirements. The buffer requirement of the conformant queue to avoid packet loss can be computed by Eq.(3.1) [20] and the eligibility time of a conformant packet c_k arriving at time t by Eq. (3.2) [20], so that c_k is transmitted before its deadline but as late as possible.

$$C_{Limit} = b + r(\Delta - \frac{b}{R}) \qquad 3.1$$

$$eligible_k^c = t + \Delta - \frac{b}{R} \qquad 3.2$$

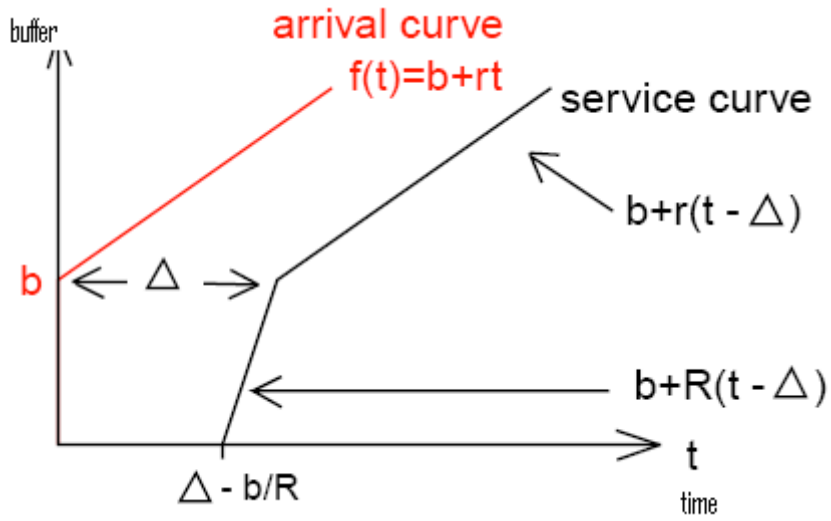

Figure 3.3 Arrival curve and service with maximum service rate R

The scheduler always starts at the head of the conformant queue. If the head of conformant queue packet is not eligible for transmission ,i.e., its eligibility time of time is longer than the current time, the scheduler chooses between the head-of-line (HOL) excess or HOL best-effort packets, based on which one has the smaller virtual time.

$$v_k^e = \frac{L_k^e}{e\_ratio} + \max\left(v(a_k^e), v_j^e\right) \qquad 3.3$$

$$v_k^b = \frac{L_k^b}{b\_ratio} + \max\left(v(a_k^b), v_{j-1}^b\right) \qquad 3.4$$

The virtual times of excess and best-effort packets, v^e_k and v^b_k respectively, are computed according to SCFQ based on pre-determined parameters, so that the two queues share the residual transmission opportunities at a ratio of *e_ratio: b_ratio*, when the scheduler is not scheduling conformant packets.

In Eq. (3.3) [20] and (3.4) [20], $v(a^e_k)$ and $v(a^b_k)$ are the system virtual times upon arrival of excess packet e_k and best-effort packet b_k respectively. v^e_j is the virtual time of the last transmitted excess packet before e_k. (Note that not all excess packets that have been queued may end up being

transmitted). v^b_{k-1} is the virtual time of the last transmitted best-effort packet before b_k . L^e_k and L^b_k are the lengths of packet e_k and b_k respectively [20].

The example in Fig. 3.4 illustrates the typical behavior of "JointQueue". For simplicity, the example assumes that all packets are 1000 bits, that the delay bound for real-time packets is 0.5 ms, and that the output link speed is 10 Mbps so that it takes 0.1 ms to transmit each packet. We also assume that the excess traffic and the best-effort traffic share the residual bandwidth equally.

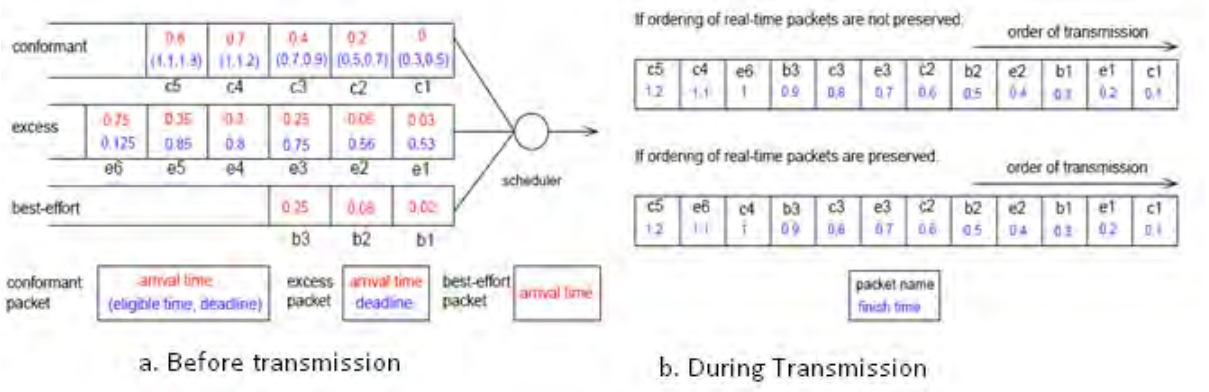

Figure 3.4 ADQ example

ScfQueue

ScfQueue Version of ADQ uses SCFQ as the only scheduler to provide both delay and lossless performance guarantees to conformant packets, as well as link-sharing between excess and best-effort traffic. SCFC is relatively easy for implementation. However, SCFQ will often transmit conformant packets earlier than necessary, and this may negatively affect the excess throughput.

Again, assuming a token bucket modeled traffic contract with average rate r and token bucket depth b, the maximum service rate of the scheduler is R. The service rate r_c can be calculated using eq. (3.5) [20] and the amount of

buffer required for conformant packets C_{limit} to avoid loss can be computed using eq. (3.6) [20].

$$r_c = max\left\{r, \frac{(b-M)p+(M+L^c_{max})(p-r)}{(p-r)\left(\Delta-\frac{2\times M}{R}\right)+(b-M)}\right\} \quad 3.5$$

$$C_{limit} = \begin{cases} b + r\left(\frac{L^c_{max}}{r_c}+\frac{2\times M}{R}\right), \frac{b-M}{p-r} \le \frac{L^c_{max}}{r_c}+\frac{2\times M}{R} \\ \frac{(b-M)(p-r_c)}{p-r}+M+rc\left(\frac{L^c_{max}}{r_c}+\frac{2\times M}{R}\right), p > r_c \quad and \quad \frac{b-M}{p-r} > \frac{L^c_{max}}{r_c}+\frac{2\times M}{R} \\ M + p\left(\frac{L^c_{max}}{r_c}+\frac{2\times M}{R}\right), otherwise \end{cases} \quad 3.6$$

p is the input peak rate of the conformant traffic, M is the maximum input packet size. L^c_{max} is the maximum conformant packet size. The remaining of the output bandwidth is shared between the excess traffic and the best-effort traffic. Therefore, based on the pre-determined link-sharing ratio e_ratio: b_ ratio, the service rate for the excess and the best-effort traffic are [20]:

$$r_e = (R - r_c) \times \frac{e\_ratio}{e\_ratio+b\_ration} \quad 3.7$$

$$r_b = R - r_c - r_e \quad 3.8$$

3.2.2. **ADQ Buffer management**

Excess packet which couldn't get opportunity of transmission before their dead line, are expired and should be removed from the queue. The key issue in supporting excess real-time packets is to be able to identify and remove *expired* excess packets, instead of wasting transmission opportunities on them. ADQ relies on buffer management to address this issue using two main procedures: "synchronization" and "clean-up". "Synchronization" is a procedure performed when transmitting a conformant packet. It locates and removes expired excess packets based on the arrival ordering between

conformant and excess packets, and by relying on the fact that all real-time packets have the same delay bound. In most cases, “synchronization” alone is sufficient. However, “clean-up” is needed occasionally when HOL excess packet expired and its main task is to identify non expired packet and when arriving packet finds the queue full.

Synchronization

Synchronization removes expired excess packets based on the fact that when a real-time packet p expires, all real-time packets arrived before p have also expired, because they all have the same delay bound. 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 both the ability to identify excess packets that arrived before a conformant packet, and on the validity of the 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 before their deadlines. Similarly, a scheduler such as SCED+ that delays the transmission of conformant packets as long as possible may result in better results than SCFQ does.

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. Segments are 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, we verify whether synchronization needs to be performed, by verifying the HOL excess packet's *syncIndex*. If it points to the conformant packet being transmitted, then the *segIndex* of the packet locates all the excess packets that need to be removed. See Fig. 3.5 for an example of how the segment pointers are arranged, and Fig. 3.6 for a summary of the major operations involved in a synchronization procedure [20].

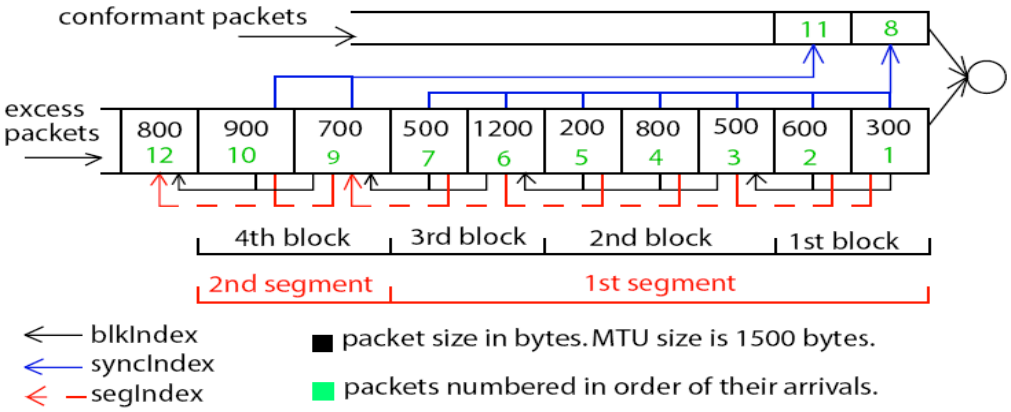

Figure 3.5ADQ Example of segment and block

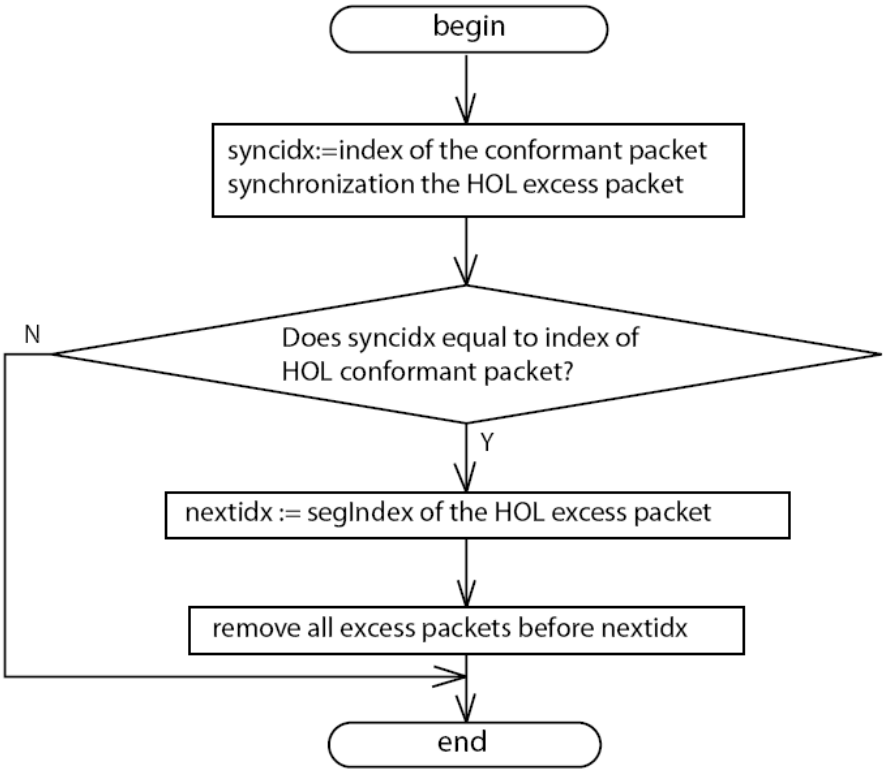

Figure 3.6 Synchronization

Clean-up

The synchronization procedure removes expired excess packets only after transmission of a conformant packet. When a large number 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 it has expired. Another procedure, "clean-up," is used to handle such rare cases.

Clean-up is called in either of the 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 of the clean-up 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 upon finding an expired HOL excess packet. In this second situation, the goal is to quickly find a non-expired excess packet in order to take advantage of the available transmission opportunity. This can always be accomplished simply 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 can cause the unnecessary removal of non-expired packets in the first segment. Thus, two intermediate steps introduced before resorting to removing the first segment [20].

A clean-up procedure involves two types 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 Maximum Transmission Unit (MTU). Fig. 3.7 gives an example of an excess queue with block pointers. We assume packets 1 to 4 are expired at the time the clean-up procedure is called.

Note that the first block in Fig. 3.7 is not complete because some of its packets have already been transmitted.

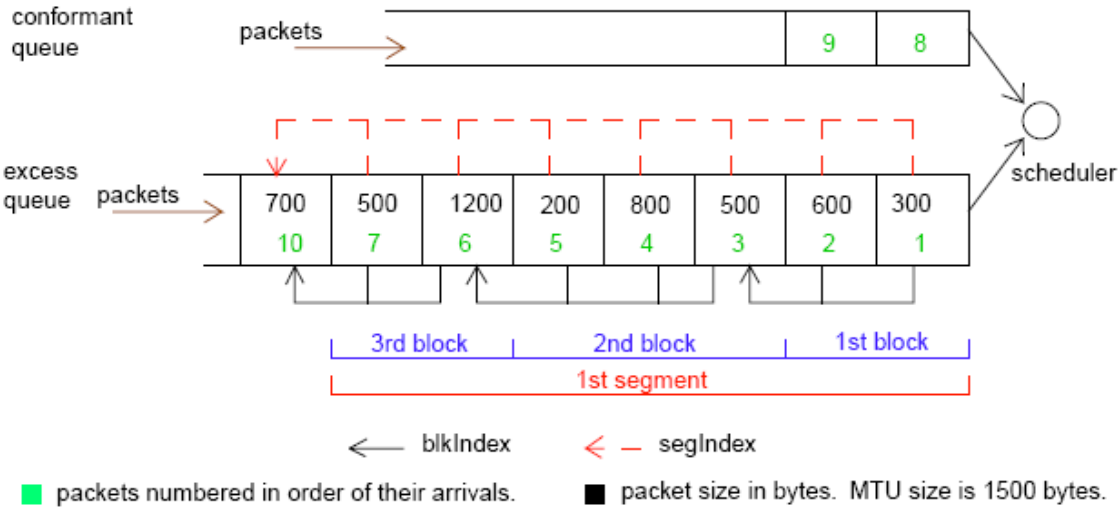

Figure 3.7 blkIndex pointers and segIndex pointers.

When clean-up is called when the arriving excess packet can't be accommodated, it first removes the first block in the excess queue. Although blocks are designed to contain more than an MTU worth of bytes, removing the first block may not be enough because the first block may not be complete. In that case, clean-up proceeds to remove the following block. This will guarantee enough space for any arriving packet since the second block is always complete. In this case, clean-up is called when enqueuing a packet, and the major operations involved are shown in Fig. 3.8.

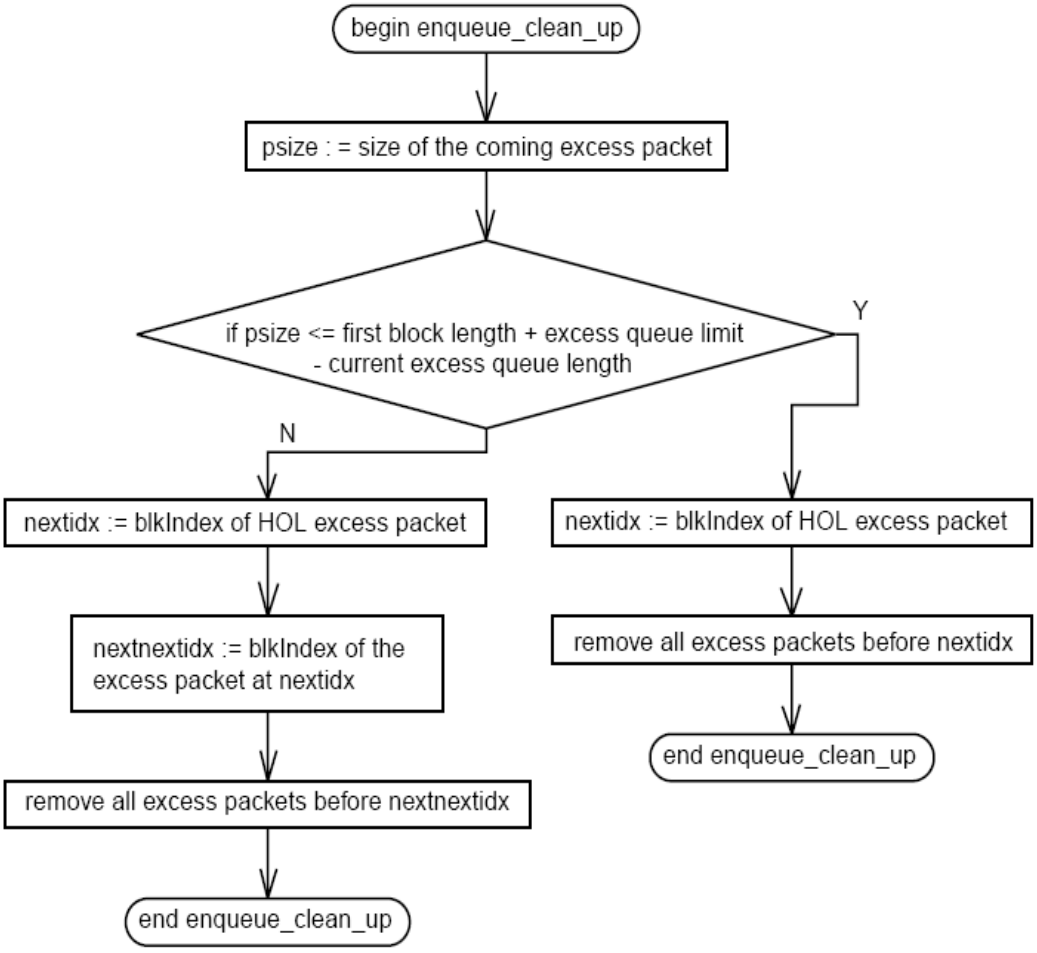

Figure 4.8 Clean-up during enqueuing

If clean-up is called because the HOL excess packet is found expired, its goal is to quickly identify a new no expired excess packet. Ideally, it should remove only the expired excess packets and make the first non-expired excess packet the new HOL packet. However, searching an ordered list typically requires O (log n) time [20].

First, the first block in excess queue will be removed. If the new HOL excess packet is still expired, all the packets in the second block will also be removed. If this fails again, the whole first segment will be removed, which guarantees that the new HOL packet is not expired. In this case, clean-up is called when dequeuing a packet and the major operations involved are shown in Fig. 4.9.

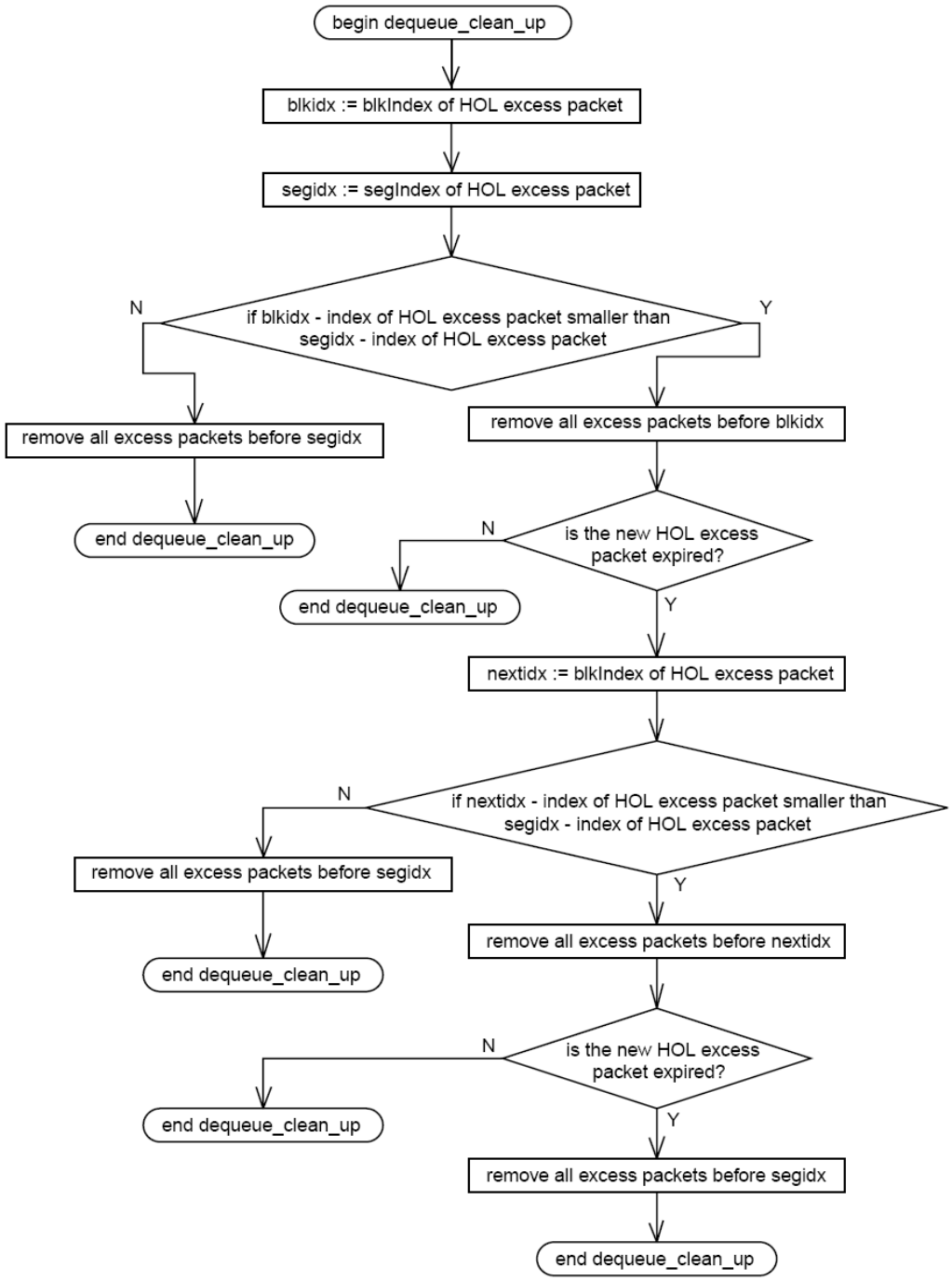

Figure 4.9 clean up used in dequeuing

Preserving the ordering of real-time packets

Preserving packet ordering is desirable when all real time-time packets are generated by the same application. Application may tolerate some amount of reordering e.g., through playback buffer, but will operate more smoothly if ordering can be preserved.ADQ can also be configured to preserve packet ordering between the conformant and excess queues when such a feature is desirable. However, enabling this feature can impact some what the excess throughput, as the ordering constraint will occasionally force the excess queue to “pass” on some transmission opportunities. The synchronization procedure guarantees that if an excess packet arrives earlier than a conformant packet, then that excess packet, if ever transmitted, will also be transmitted earlier. Preserving packet ordering, therefore, only requires an additional mechanism that ensures 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 e1 is chosen for transmission, it requires checking whether the HOL conformant packet c1 arrived earlier than e1. If c1 did arrive earlier, c1 will be transmitted instead of e1 to enforce the ordering [20].

Chapter Four

4. Proposed Solution for the Problem

In this chapter, the effect of transmitting more than one real time application with the same and different delay bound using ADQ as an active queue management is analyzed and solution on incorporating different delay bound will be proposed. The next sections 4.1 study the analysis of two real-time applications on the existing algorithm.

4.1. Analysis of two application on ADQ

As seen briefly in chapter 2, queuing delay is the time the packet stays in a queue before transmission. Queuing delay occurs when the queue service rate is not fast enough to serve all the incoming packets from different sources using this queue, therefore, the packets accumulate in the queue and wait for service.

The main objective of this thesis is to incorporate different delay bound in the existing active drop queue. However , to investigate how the schedulers behaves for different application with the same delay bound, we also need to check the scheduler on transmitting application with same delay bound. The following section will discuss on the scheduling algorithm of ADQ, while transmitting two applications with the same and different queuing delay bound.

4.1.1. Two applications with same queuing delay bound

The main advantage of jointQueue algorithm of ADQ is transmitting conformant packet just before it expires. The Fig. 4.1 below shows the behavior of the ADQ scheduling algorithm on transmitting two real time applications with the same queuing delay bound requirement. The example assumes that all packets are 1000 bits, the delay bound for two of real-time is 0.5ms and that the output link speed is 10Mbps. So that it takes 0.1ms to

transmit each packet. Excess traffic and best-effort traffic assumed to share the residual bandwidth equally.

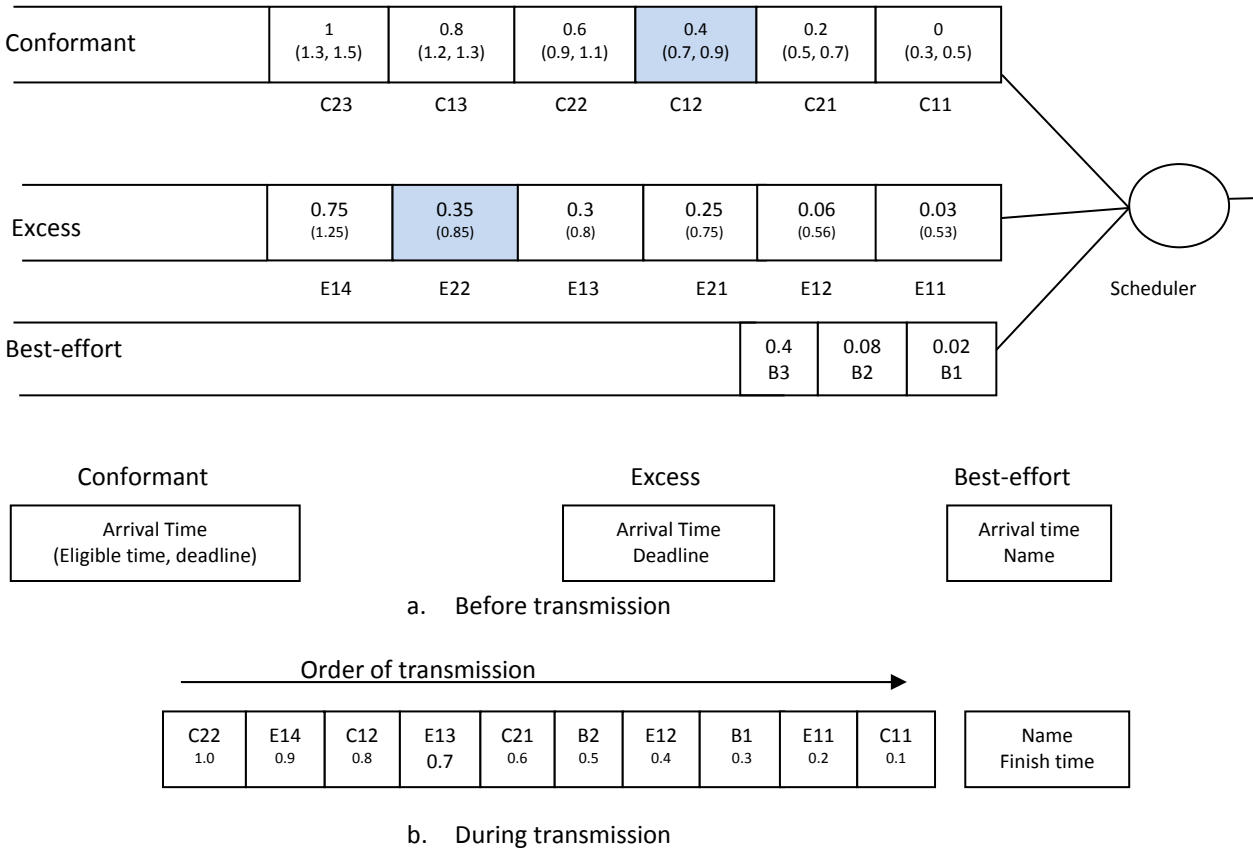

Fig 4.1 Two real-time applications with the same delay bound

As shown in the figure E_{22} dropped unnecessarily by synchronization after a conformant packet C_{12} (arrived after E_{22}) is transmitted. However E_{22} has arrived before the conformant packet C_{12} and was not expired; it couldn't get transmission chance before deadline of C_{22}.

Synchronization procedure is called after every transmission of conformant packet to remove expired excess packets that arrived before the conformant packet being transmitted. However in transmitting two or more real time applications, the synchronization procedure of ADQ has no mechanism to identify specific expired packets for the corresponding conformant packet while transmitting more than one real time application.

4.1.2. Two applications with same queuing delay bound

Example in fig. 4.2 in the next page shows, two real-time application with different delay bound 1ms and 0.5ms and ADQ is configured for delay bound of 0.5ms.

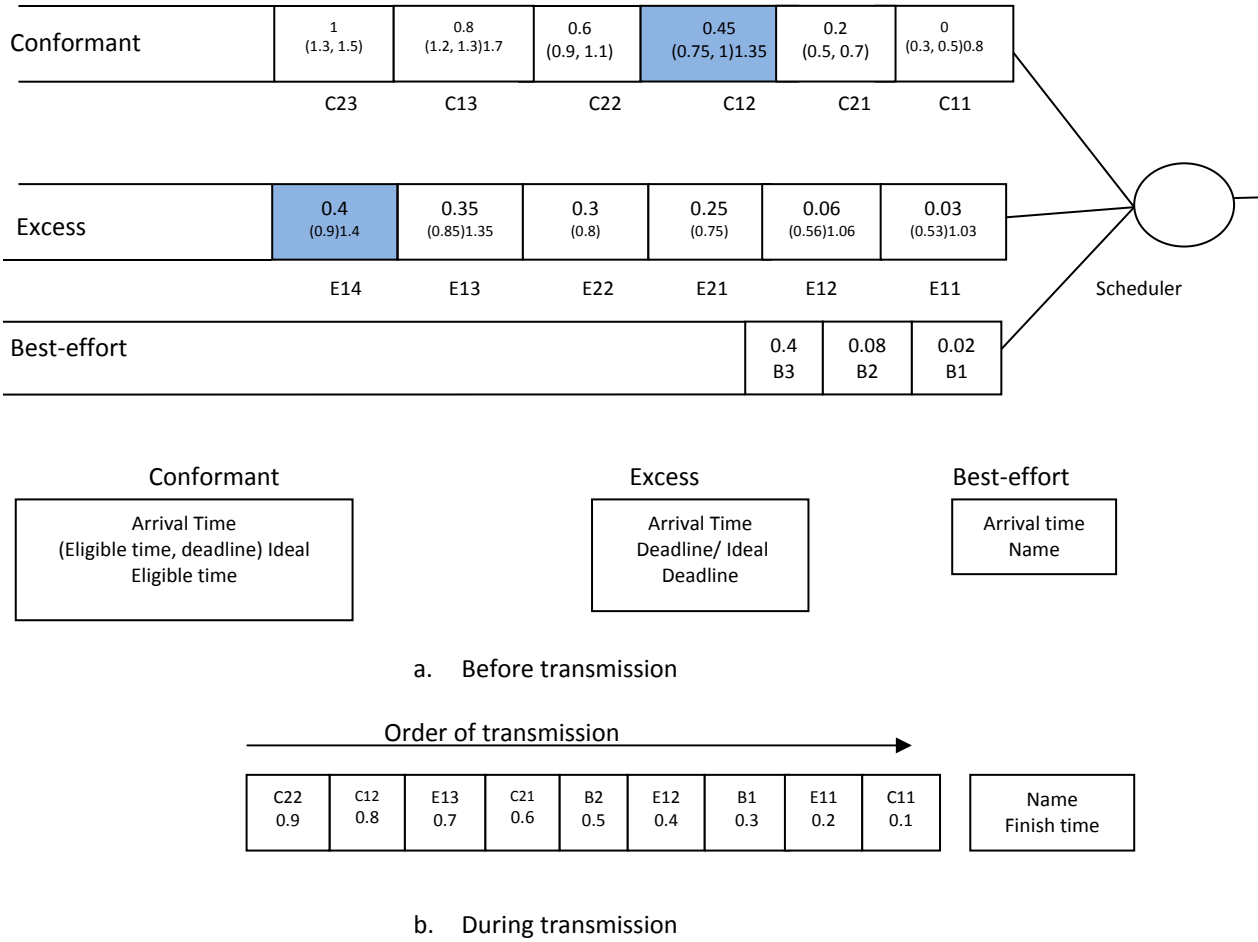

Figure 4.2 Two real-time applications with different delay bound.

Similar to the first example; as shown in the figure E_{14} is dropped unnecessarily by synchronization after a conformant packet C_{12} (arrived after $E_{14)}$ is transmitted. However e_{14} has arrived before the conformant packet c_{14} and was not expired; it couldn't get transmission chance before deadline of C_{12} as its deadline is 1ms. Since the first application can be delayed more time, which is 1ms delay bound, than the second one; excess packets of the first application would get more transmission opportunities.

In the first scenario we have seen, after serving a conformant packet of an application , the synchronization procedure removes all excess packets that have arrived before it even if it is not part of an application whose conformant packet is being transmitted. This is due to luck of association of conformant packet with excess packet in application specific level.

In the second scenario we have seen that, when transmitting data streams of two real time applications with different delay bounds, the ADQ should be configured with smaller delay bound to fulfill delay bound requirement of the two applications. A conformant packet of the application with longer delay bound will be transmitted near to its deadline (its deadline calculated with smaller delay bound), this results in early synchronization of conformant packet with excess and hence results in unnecessarily removal of non- expired packets. However, it's main advantage of jointQueue algorithm, delaying conformant packet till its deadline to get more opportunity for transmission of excess packets.

4.2. Proposed solution

The main objective of this thesis is to incorporate different queuing delay bound in the existing Active drop queue as different application require different delay bound. The figure 4.3 below shows the proposed general structure of ADQ. To reduce the early synchronization and problem of association of conformant packet with excess packets in application level; we need to divide each conformant and queue class in to sub-classes based on delay bound. Sub-classing of conformant and excess class helps:

- jointQueue algorithm to treat packets independently up on there delay requirement.
- Reduce the unnecessarily removal excess packets as ADQ has no mechanism to differentiation conformant and excess packets of one application from another.

Class differentiation as well as sub classing based on delay bound is done by application. The enqueuing procedure is responsible on forwarding each

arriving packets into corresponding virtual class queue and estimating eligible time and deadline of conformant packets based on its delay bound. The dequeuing procedure serves conformant traffic according to its delay requirement and link sharing between excess and best-effort traffics.

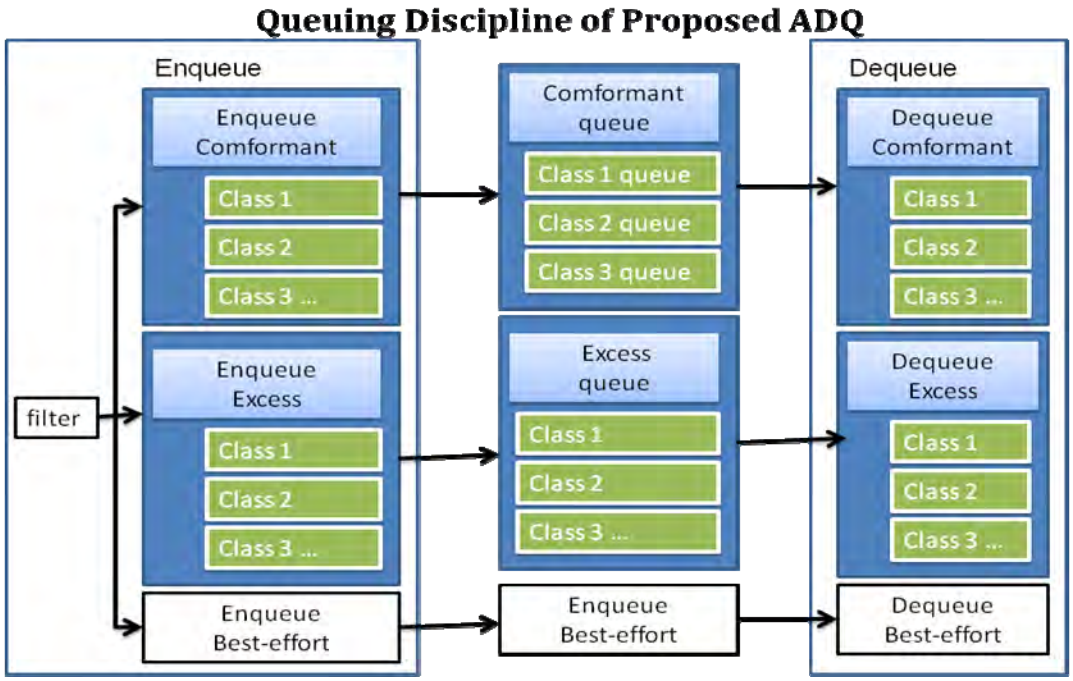

Figure 4.3 Proposed Structure of ADQ

Serving conformant packets with in its delay bound is the main aim of active drop queue. However, delay can be guaranteed only for certain amount of packets. The amount of buffer required for conformant packet to avoid packet loss i.e. when the network is busy on transmitting only conformant packets, can be calculated by eq.(4.1)[10] ,where R is maximum service rate, r is arriving rate, token bucket depth of b and Δ is delay bound assuming a token bucket modeled arrival curve[20]. The service curve show in fig. 4.4 as in the original ADQ can fulfill ADQ scheduler goal scheduling conformant packets as late as possible without violating their delay and loss requirements. The arrival rate of conformant should be equal to service rate.

$$C_{Limit} = b + r(\Delta - \frac{b}{R}) \qquad 4.1$$

$$eligible_k^c = t + \Delta - \frac{b}{R} \qquad 4.2$$

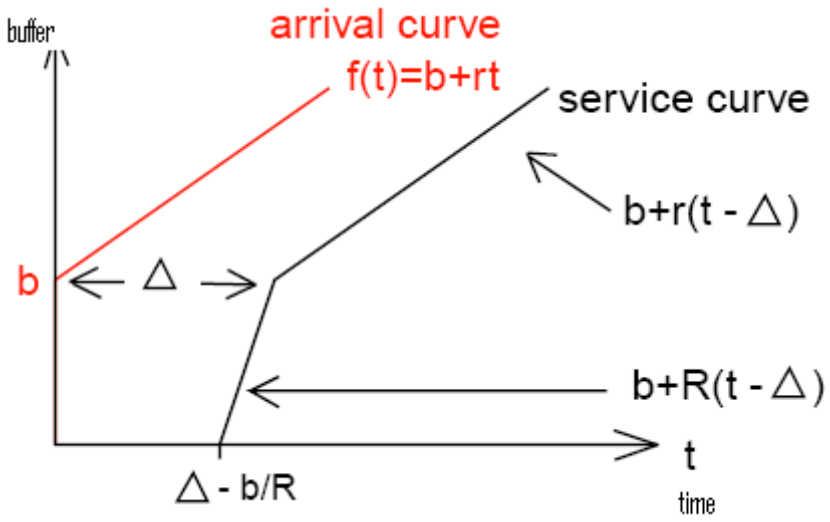

Figure 4.4 Arrival curve and Service Curve

4.2.1. Calculating Conformant Traffic Buffer Requirement

The existing ADQ uses eq. [4.1] to compute the amount of buffer requirement to avoid packet loss. Since our main aim to incorporate different delay bounds (i.e. Delta for each conformant class is different), EQ [4.1] could not be directly used to calculate the amount of buffer requirement. Assume n subclass, arriving rate r_1 to r_n and delay bound Δ_1 to Δ_n. The conformant limit can be calculated by adding each delay class contribution for conformant queue size.

$$C_{limit} = b + r_1\left(\Delta_1 - \frac{b}{R}\right) + r_2\left(\Delta_2 - \frac{b}{R}\right) + \dots\dots\dots\dots\dots\dots r_n\left(\Delta_n - \frac{b}{R}\right) \qquad 4.3$$

Assuming arriving rate of all subclasses are the same.

$$C_{limit} = b + r(\Delta_1 + \Delta_2 + \Delta_3 + \cdots + \Delta_n) + -\frac{nrb}{R} \qquad 4.4$$

4.2.2. Calculating Eligibility time

Suppose a_k is the arrival time of the k^{th} conformant packet from traffic stream, assuming that $a_k \leq a_{k+1}$ for each k. Packet k is assigned an initial eligibility time $e_k^{initial}$ and terminal eligibility time (dead line) $e_k^{terminal}$, where

$e_k^{initial} \leq e_k^{terminal}$. If x_k is the departure time of conformant packet K from AdQ scheduler, then $e_k^{initial} \leq x_k \leq e_k^{terminal}$ [8]. The eligibility time of the arriving conformant packet can be computer using EQ(4.2)[20].

Computing eligibility time is relatively easy. Difference on delay bounds doesn't have any effect on calculating eligibility time of a conformant packet; however eligibility time two or more conformant packets may be the same or may be with in the transmission time of a packet, i.e. eligibility time difference is less than transmission time of a packet. This would result contract violation of the guaranteed traffic (i.e. conformant packets will be delayed longer than its deadline).

Concurrency of eligibility time is one of the main issues in incorporating delay bounds in ADQ. Reducing the eligibility time of a packet which coincides with the existing by a multiple of transmission time can solve the problem. However, detecting occurrence of concurrency is also the other big problem.

Since eligibility time for all of the conformant packets lays between sum of packet arrival time and minimum delay bound and sum of packet arrival time maximum delay bound, concurrency of eligibility time occurs only with in these time frames. Figure 4.5 shows eligibility time concurrency region.

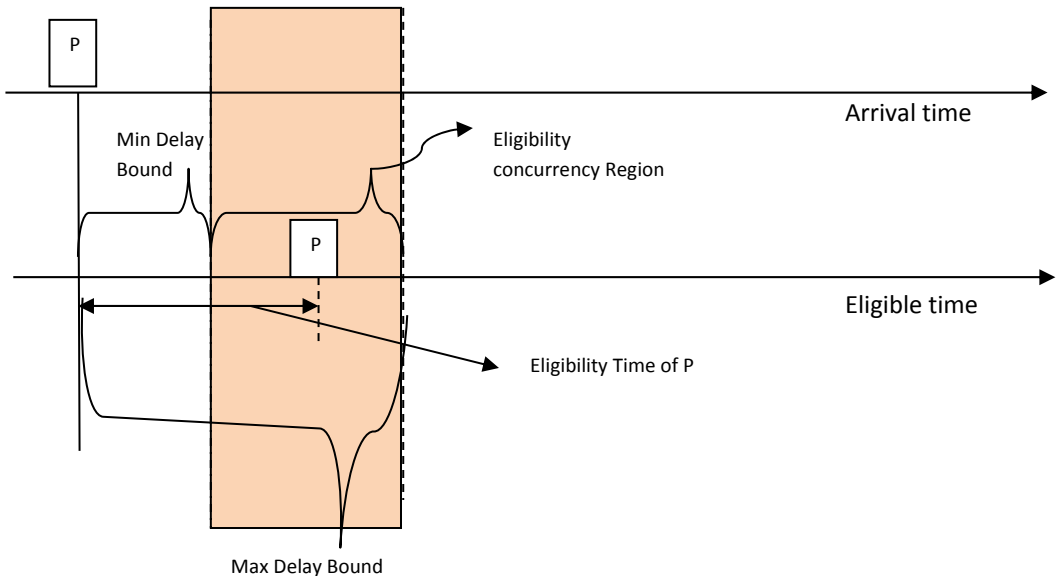

Figure 4.5 Eligibility time concurrency region.

4.2.3. Eligibility time concurrency detection and avoidance

Eligibility time Concurrence detection and avoidance are the two big issues on calculating eligibility time while enqueuing a conformant packet. Simple and straight forward solution for eligibility time concurrency is by reducing calculated eligibility time of all subclasses(i=1,2,3 to n) except sub class with smallest delay bound by a multiple of MTU transmission time i.e. (i-1)* $T_{MTU,}$ where T_{MTU} is MTU transmission time. This ensures that all conformant packets will be transmitted with in there delay bound. However reduction of eligibility time will result in early synchronization and hence reduce throughput of excess real time traffic.

The other technique resolves concurrency eligibility time by detecting after the occurrence of concurrency. We call this concurrency detection and avoidance. Concurrency detection is used to detect the concurrency of arriving packet eligibility time with the eligibility time of conformant packet in the queue. Concurrency avoidance is responsible for resolving the concurrency of eligibility time.

4.2.4. Concurrency detection

When a conformant packet P arrives at time t and its eligibility time P_e, before assigning eligibility time to the incoming packet the concurrency detection checks the eligibility time with the existing packet in a queue. The very simple solution for detecting concurrencies is checking all conformant queue packets from the rear of the queue to packets whose eligibility time with in eligibility time concurrency region. However, this has o (n) complexity time, where n is number of packets from rear of all conformant queue with in eligibility time concurrency region.

Concurrence detection can be implemented with O (1) complexity time by maintaining records of eligibility time at which concurrence would occur. As discussed above concurrence will occur only with in eligibility time concurrency region; we need to maintain records eligibility time only for those whose eligibility time with in this region. Assume max delay bound

Mx, min delay bound Mn and transmission time for MTU t, then the number of records we need to maintain for Concurrence detection will be (Mx-Mn)/t. This can be implemented easily using a simple circular array. Concurrence of eligibility time can be easily detected by checking whether the element of the array at eligible time of the arriving packet true or false. The required index of an array can be found by dividing eligible time of the arriving packet by transmission time t.

4.2.5. Eligibility time concurrence avoidance

When eligibility time concurrence occurs, the concurrency can be avoided by changing the eligibility time of one of the packets on which concurrence of eligibility time occurs. Changing eligibility time of the packet already in the queue is difficult compared to the arriving packet because of searching for the packet is not an easy task as seen in concurrency detection. However, eligibility time concurrency can be solved by reducing the eligibility time the incoming packet; so that it couldn't result to concurrency. Figure 4.6 below shows eligibility time record.

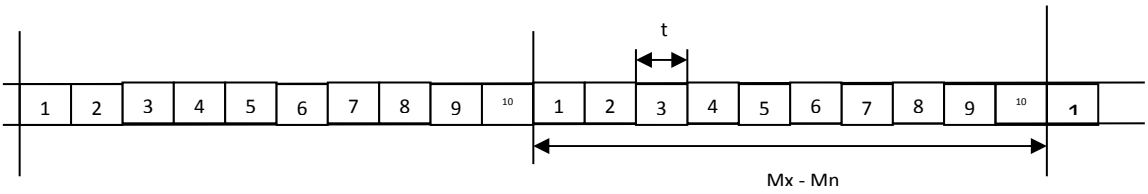

Figure 4.6 Eligibility time record.

4.3. Buffer Management

The other issue in incorporating different delay in active drop queue is identifying and removing of expired excess packets in each excess queue. The existing ADQ achieves this by synchronization and clean-up procedures. Synchronization and clean-up procedures of the proposed solution are the same as the existing ADQ except little modification to support number of classes. The following topic will discuss the

Synchronization and clean-up procedure and modification done in supporting different delay bounds.

4.3.1. Synchronization

The synchronization procedure is responsible on removing expired excess packets. When real time packet p expires, all packets real time packets arrived before p and with same packet delay bound as p have also expired. Packets in excess queue; which could not get opportunity of transmission before transmission of conformant packet arrived after them of the same delay class, are expired packets and should be removed from the queue. This is the idea behind Synchronization. Synchronization is performed after every transmission of conformant packet.

Synchronization needs associating of a conformant packet with the excess packet that arrived before it of the same delay class. Associating excess packets with a conformant packet effectively divide the excess packets in to segments. As shown in Fig. 4.7, segments are 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.

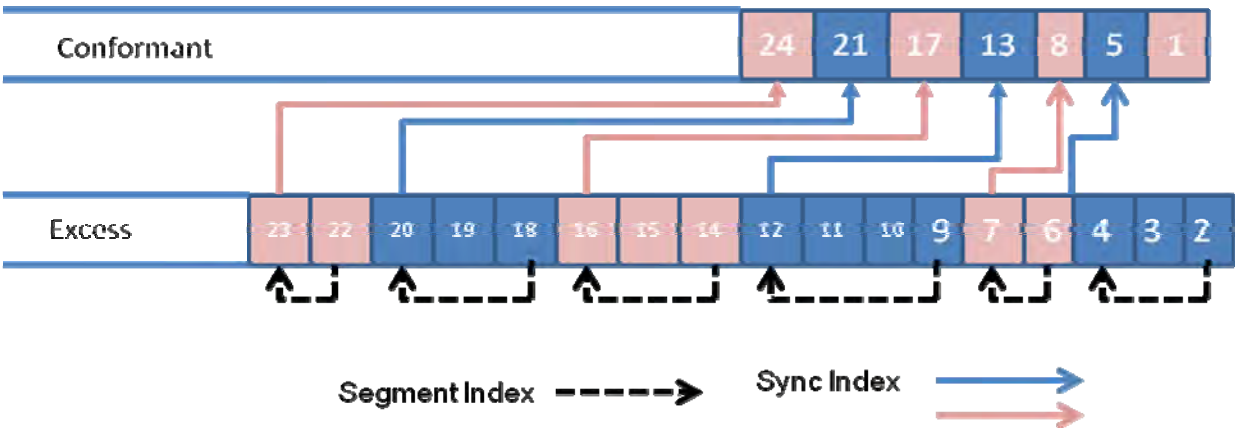

Figure 4.7. Synchronization and Segment Index.

The flowchart below shows the synchronization procedure of the proposed ADQ algorithm.

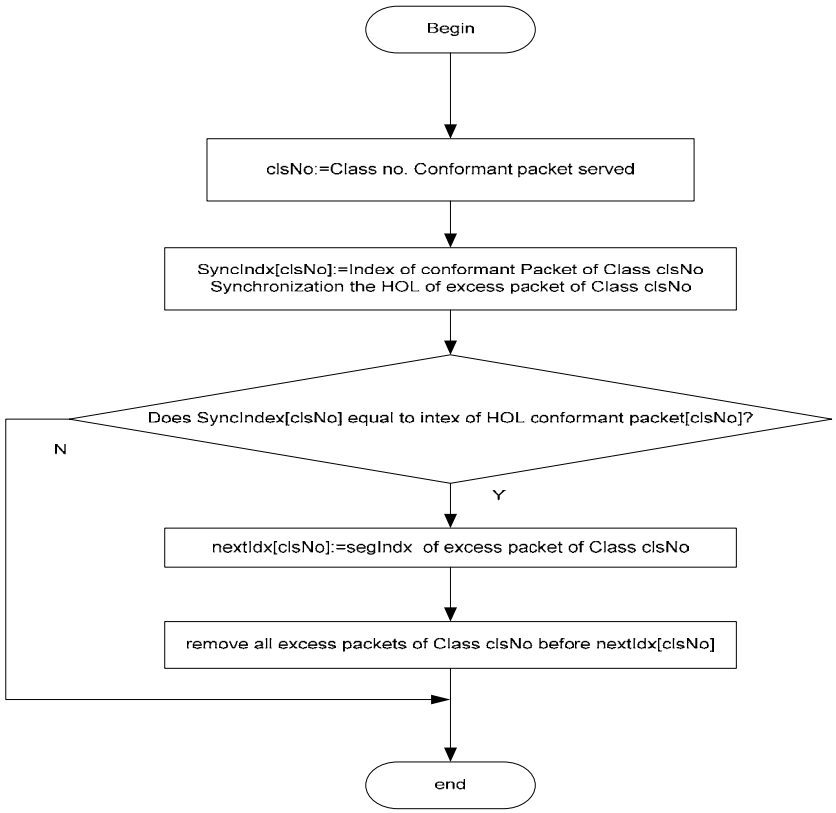

Figure 4.8 Synchronization flow chart of proposed solution

4.3.2. Clean up

As synchronization, clean up procedure can also used to remove expired excess packets in the queue. The clean up procedure can be called when a packet arrives (enqueuing) and departs (dequeuing) the queue.

Clean up on enqueuing

When an incoming excess packet finds a queue full, the task of clean up procedure is to remove enough excess packets from the head of excess packets queue with the same delay bound as the incoming packet. This may involve removal of non expired packets.

Since removal of one excess packet may not be enough for the incoming packet, clean up procedure needs to remove series of excess packets until it get enough room for the incoming packet. However it prohibits O (n) complexity time, where n is number of packets to be removed. The clean up procedure removes enough packets with O (1) complexity. Clean up

subdivide each segment of excess packets in to blocks. A block consists of one or more contiguous excess packets, so that there total length is at least equal to link MTU.

When clean up procedure is called, even though blocks are designed to contain more than one MTU the first block may be less than one MTU as some of its packets already been transmitted. So removing the first block may not be enough. In that case, clean up proceeds to remove the following block. These will guarantee enough space for any arriving packets since the second block is always complete. The flow chart below shows major operations clean up during enqueuing for proposed ADQ structure.

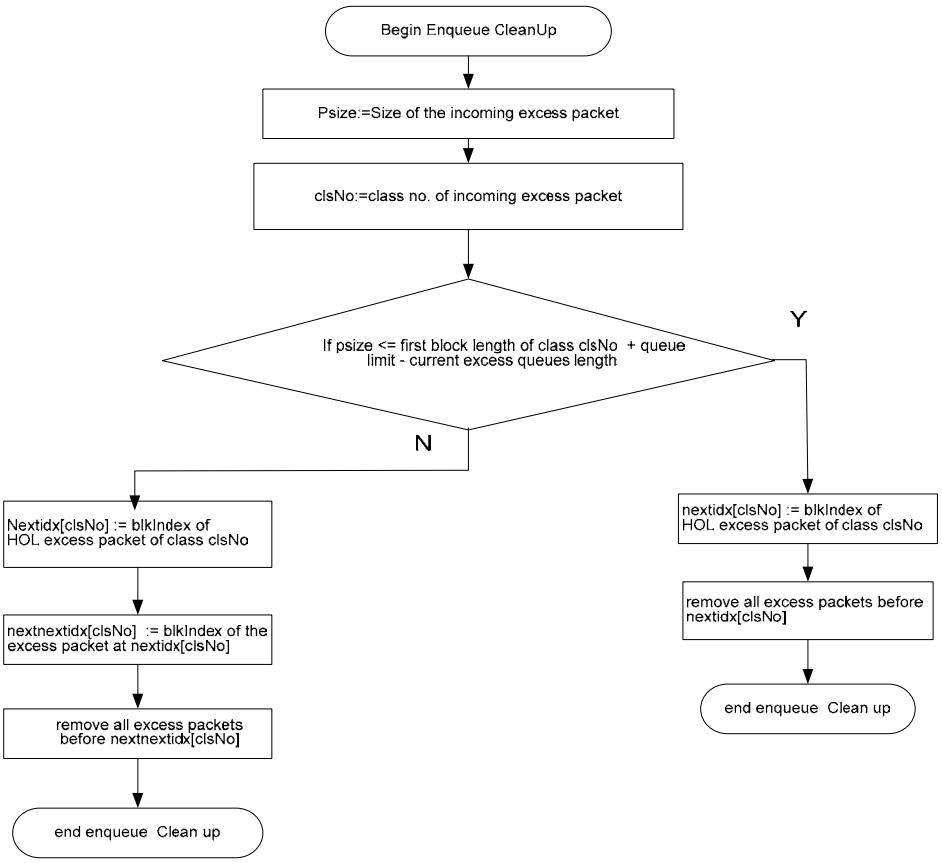

Figure 4.9 Clean-up on enqueuing

Clean up on Dequeuing

When a HOL of excess packet is expired, the main task of clean up procedure during dequeuing is to quickly identify a non-expired packet. Searching an ordered list typically requires O(log n) time. To find a non expired packet for simplicity ADQ first removes first blocks in excess queue. If the new HOL of excess packet is still expired, ADQ then proceeds to remove the next block. If this fails again, it removes whole first segment which guarantees that new HOL packet is not expired. The flow chart on next page shows major operations of clean up on dequeuing.

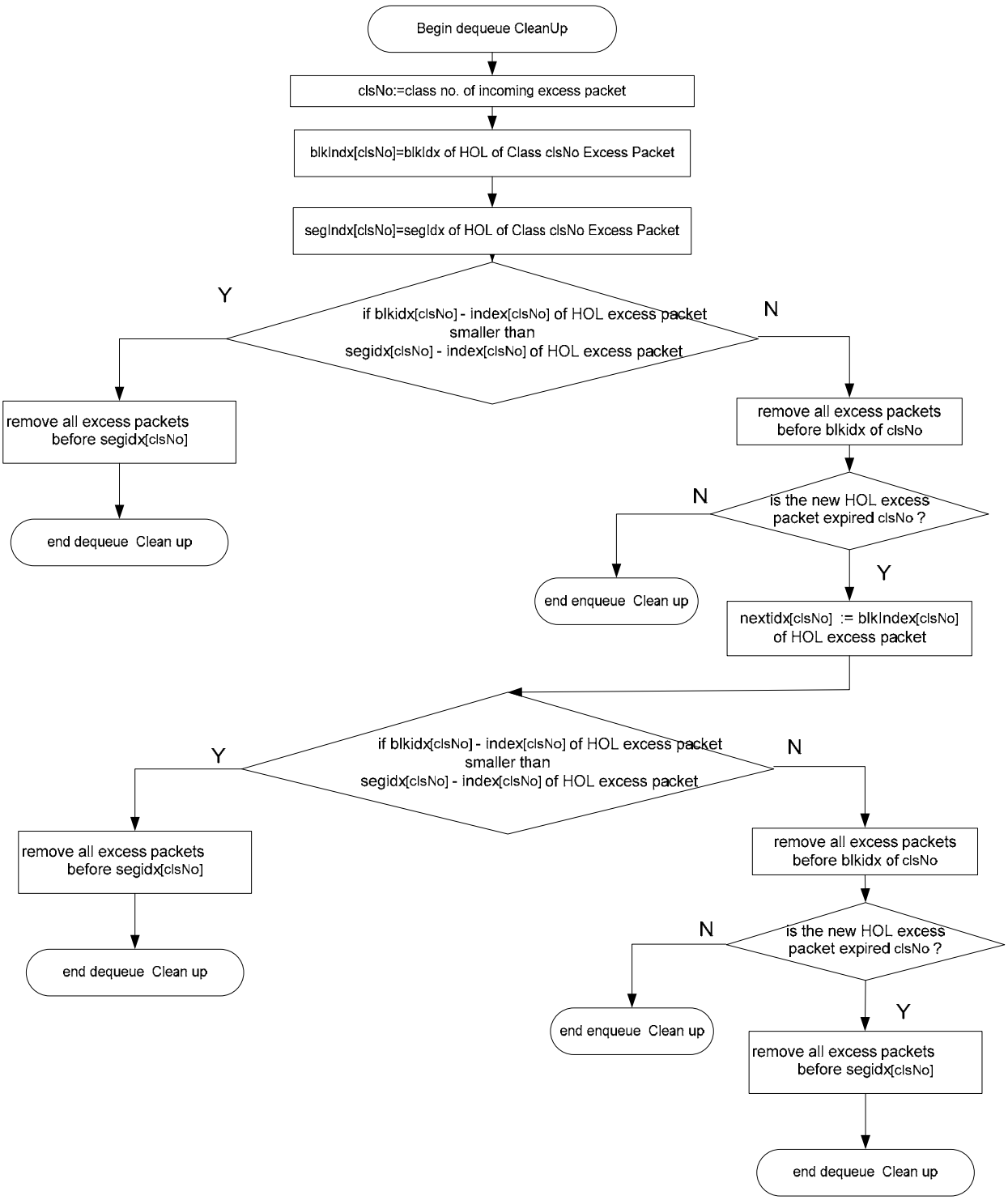

Figure 4.10 Flowchart clean up on dequeuing

4.4. Queuing Discipline of the Proposed Algorithm

As Shown in the figure 4.3, the queuing discipline of the proposed ADQ, the enqueuing function filters(identifies) the incoming packet into conformant, excess and best-effort based on IP header information of the incoming packet. Depending on the packet type, the incoming packet will be forwarded in to the corresponding queue and necessary changes on structure of synchronization and segment will be updated accordingly. The scheduling algorithm is implemented in the dequeue function of active drop queue. The dequeuing function is responsible on dequeuing conformant packets with in its delay bound and link sharing between excess and best-effort packets the remaining bandwidth. The following section describes operation of enqueuing and dequeuing functions.

4.4.1. Enqueuing

Traffic class and delay class information of a packet is stored in packet header. When a packet arrives at a node; the enqueuing function checks the packet header to identify its type (traffic class) i.e. conformant, excess or best-effort. After identifying the traffic class of a packet, if the packet is identified to be a real-time, it further checks packet header to identify its delay class of the packet and lower level enqueuing functions will be called accordingly. The lower level enqueuing functions are conformant, excess and best–effort enqueuing. Figure 4.11 shows flow chart of enqueuing function of the proposed ADQ.

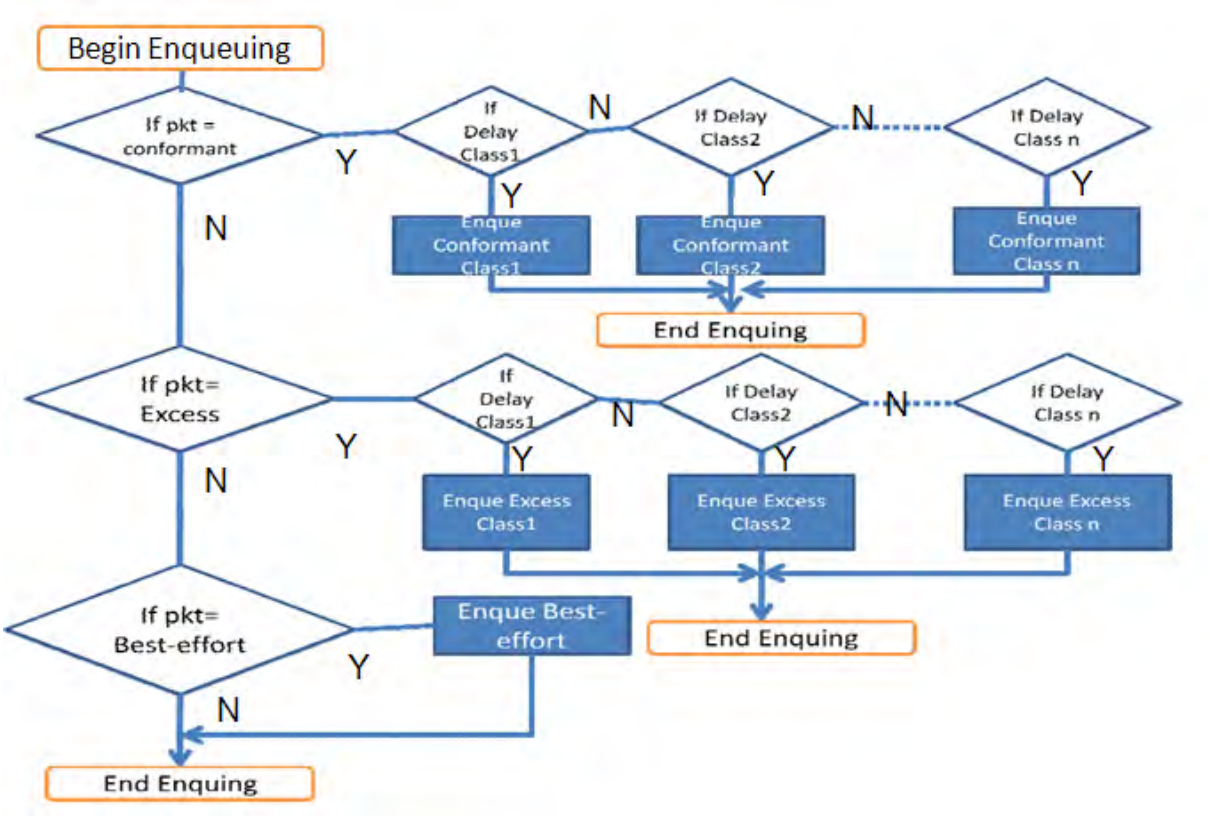

Figure 4.11. Enqueuing Flow chart

Enqueuing Conformant Packets

The following steps describe the procedures of enquueing a conformant packet

1. First the size of the incoming conformant packet compared with the remaining space intended for conformant queue.
2. The enqueuing function further checks the header information of the arriving to identify to which sub class (delay bound class) it belongs.
3. Deadline and eligibility time of the conformant packet will be computed according to its delay bound.
4. Concurrency detection and avoidance mechanisms compares the eligibility time with the existing conformant packets. If concurrency detected, new eligibility time accordingly will be assigned.
5. If the previous enqueued packet is an excess packet, i.e. there exists a segment of excess packets to be synchronized by the incoming conformant packet. Syncindex of those excess packets will be associated with the incoming packet.

Enqueuing Excess packets

The following steps describe the procedures of enqueuing an excess packet.

1. IP header of the arriving packet will be checked to identify to which sub class (delay bound class) it belongs.
2. If the size of the incoming packet greater than the reaming available space in a queue, clean up procedure will be called to remove excess packets from the head of excess queue of the delay bound as the incoming packet.
3. Depending on its delay bound, deadline will be computed.
4. Start a new segment, if the last enqueued packet is a conformant packet or excess queue was originally empty.
5. Start a new block if the current last block exceeds MTU size in length.

Enqueuing Best-effort

For simplicity the drop-tail used for enqueuing best-effort traffic.

4.4.2. Dequeuing

The main objective of the scheduling algorithm of ADQ is to transmit conformant packets before its dead line but as late as possible. When it is time to transmit, the dequeuing function always starts from the head of conformant packet with lower delay bound. If HOL conformant packet with smallest delay bound is not eligible for transmission, it will try from the second sub class to the one with longest delay bound. If all HOL conformant packets are not eligible the scheduler will transmit excess or best effort. Figure 4.12 shows flow chart of dequeuing function.

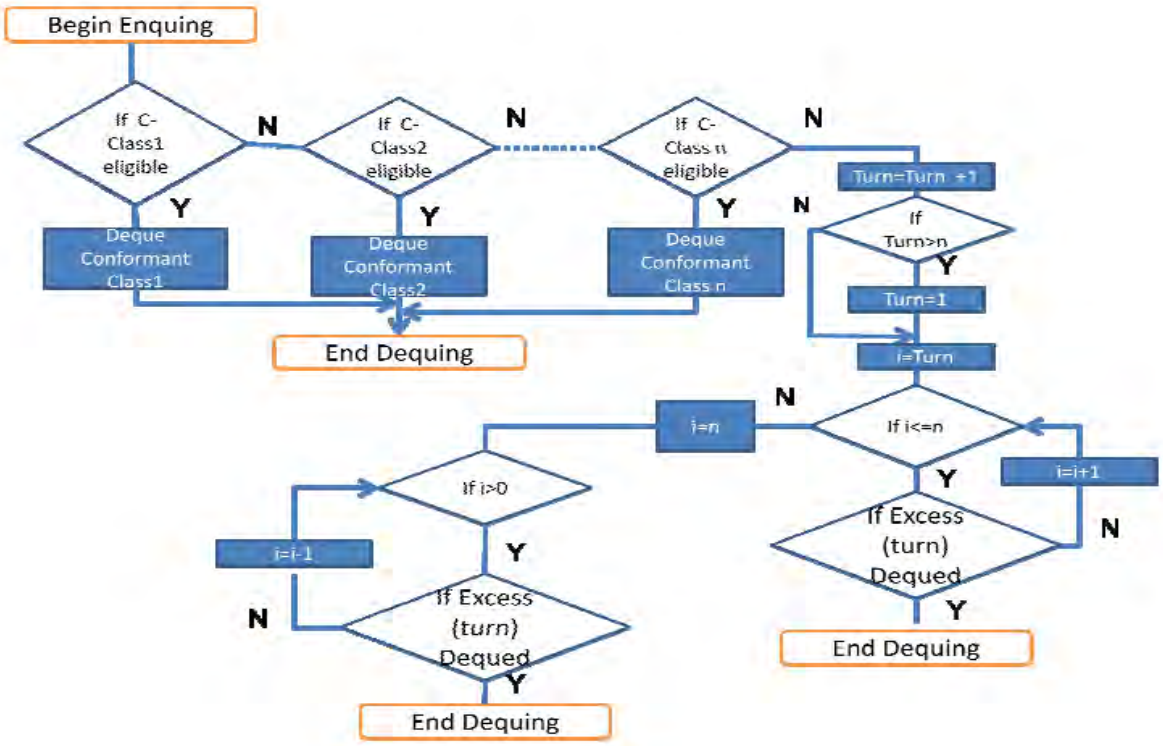

Figure 4.11 Dequeuing function.

Dequeuing conformant

The following steps describe the procedures of dequeuing conformant packet.

1. If HOL of conformant packet p is eligible for transmission, the dequeue function returns the pointer to the packet to be sent.
2. Synchronization. After a conformant packet is dequeued, excess packets which has arrived before it of the same delay bound, has to be removed from excess queue. This synchronizes the HOL of conformant queue with the corresponding excess queue.

Dequeuing Excess

If all HOL of conformant packets are not eligible, HOL of excess and HOL best-effort packets will be considered. As in the existing ADQ one with smaller virtual time will be chosen. However, we may have more than one excess queue. As in the dequeuing conformant starting from smallest delay bound will not work here, because excess packets with longer delay bound will expire before getting chance of transmission. With out taking delay bound into consideration, simple turn by turn opportunity for each excess queue will result in better link sharing among excess packets.

The following steps describe the procedures of dequeuing excess and best-effort packets.

1. Checks HOL excess packet whether it is expired or not. If HOL is expired, Clean-up will be called. If there is no non-expired packet, best-effort traffic packet will be dequeued.
2. Computes virtual time of HOL non expired excess packet and best effort traffic. One with smaller virtual time will be dequeued.

4.5. Data structure

The conformant, Excess and Bes-effort traffic queues in the original ADQ implemented in circular array and the size of each packet queue determined at compilation time. Implementing classes of queue in form of array makes synchronization and cleanup as well as maintaining syncIndex and segIndex easier. Since the size of an array can only be assigned at compiling time; the size of all classes of queue should be determined before implementation. However, subdividing the queue into number of classes with fixed buffer size will aggravate the queue full

problem, which one of the big problem in congestion management [RFC2309].

Since increasing of virtual classes with fixed sizes of buffer introduces a false queue full problem, I have devised a mechanism creating virtual queues with variable delay in the form of linked packets. As shown in the figure below two indices, enqueuing and dequeuing indices are used. The enqueuing index used for maintaining the synchronization record, where as the dequeuing index used during synchronization and clean up.

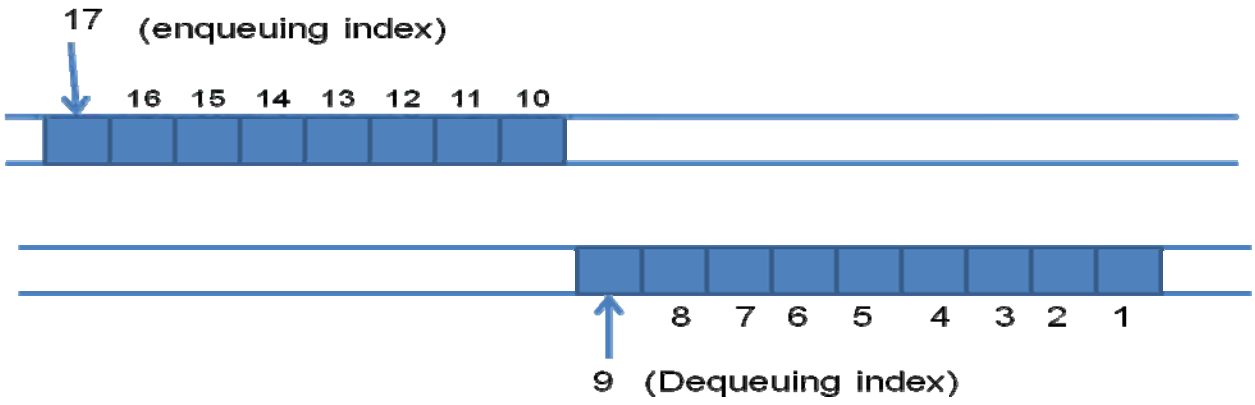

Figure 4.12 enqueuing and dequeuing index.

Chapter Five

5. Simulation Test and Analysis

This section describes the experimental setup and NS-2 simulation details associated with our comparison study of the performance of original ADQ with the modified ADQ. This includes simulation input and output traces, data extraction and analysis, experimental setup, and simulation scenarios with simulation network topology and simulation design specification.

5.1. Network Simulator

The Network simulator version 2 (NS2) [25] is an object oriented and discrete event driven simulator, written in C++, with an OTcl interpreter as a frontend. NS-2 simulates a variety of IP networks and includes network protocols such as TCP and UDP, traffic source behavior such as FTP, Telnet, Web, CBR and VBR, router queue management mechanisms such as Drop Tail, RED, PI and AVQ. NS-2 also supports simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

After the implantation of the existing and modified version Active Drop Queue written in C++ in NS2, the simulation script written in tcl is executed on NS2, results were extracted from the trace files. The procedure below shows simulation process in this thesis.

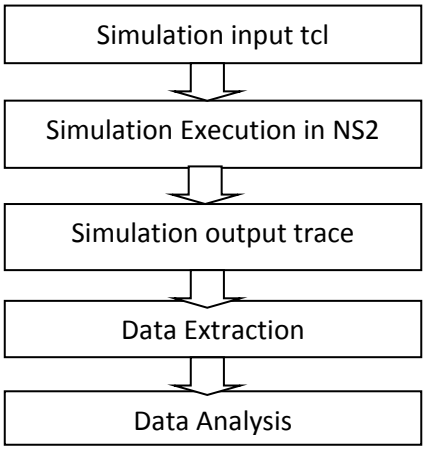

Figure 5.1 simulation procedures with NS2.

5.2. Simulation Input

The input tcl script defines the network topology and instantiates nodes, agents and applications. It also defines start and end time of simulation. NS2 has most of the common IP network components implemented including TCP (Tahoe, Reno and Vegas) and UDP transport agents, and RED router queues. However, NS-2 does not support any multimedia traffic generators. Therefore, we built a trace- driven MPEG traffic generators.

Standard MPEG encoders generate three distinct types of frames, namely I, P, and B frames, in a specific pattern such as IBBPBBPBB. The I frame is encoded in Intra mode, and is essential for the prediction coding of other frames. If part of an I frame is lost, then all frames in the group of pictures including this particular frame are impaired. The P frame is encoded in prediction mode, while the B frame is encoded in double prediction mode. As with the I frame, the P frame is also important. If part of a P frame is lost, the impairment propagates the particular P frame, previous B frames, and the following frames in the group of pictures that includes this P frame. Conversely, if part of a B frame is lost, the impairment propagates solely within that frame.

Throughout this thesis the 'Star War IV' MPEG[26] is used as input trace file for the MPEG-trace driven traffic generator, the I frame as conformant and P and B as excess.

5.3. Simulation Output

There are a number of ways of collecting output or trace data on a simulation. Generally, trace data is either displayed directly during execution of the simulation, or (more commonly) stored in a file to be post-processed and analyzed [17]. There are two primary but distinct types of monitoring capabilities supported by the simulator. The first, called *traces*, record each individual packet as it arrives, departs, or is dropped at a link or queue. Trace objects are configured into a simulation as nodes in the

network topology, usually with a Tcl “Channel” object hooked to them, representing the destination of collected data (typically a trace file in the current directory). The other types of objects, called *monitors*, record counts of various interesting quantities such as packet and byte arrivals, departures, etc. Monitors can monitor counts associated with all packets, or on a per-flow basis.

Since the trace file contains a lot of extra information than needed for analysis, extraction of necessary data is required. However, extraction of data requires another effort. During implementation of the algorithms we built our own trace file that traces information only needed for analysis. This is relatively easy than developing an extraction mechanism. Figure below show sample out put trace files.

```
Q: Conformant :              2   Excess:              23   Best-Effort:
   0
A:  C:       2592  E:       28506  B:          0
S:  C:       2590  E:       27264  B:          0
D:  C:          0  E:        1219  B:          0
DR:  F:          0  EX:        1219
-----------
=
Q: Conformant :              2   Excess:              13   Best-Effort:
   0
A:  C:       2592  E:       28507  B:          0
S:  C:       2590  E:       27265  B:          0
D:  C:          0  E:        1229  B:          0
DR:  F:          0  EX:        1229
-----------
Q: Conformant :              2   Excess:              12   Best-Effort:
   0
A:  C:       2592  E:       28507  B:          0
S:  C:       2590  E:       27265  B:          0
D:  C:          0  E:        1230  B:          0
DR:  F:          0  EX:        1230
-----------
```

Q=queue, A=Arrived, s=served, D=Drop, DR=Drop reason, C=conformant, E=excess, B=best-effort, F=Queue Full and Ex=Expired.

Figure 5.2 Part of output trace file.

5.4. **Data Analysis**

The data extracted from the traced files are fed into the data analysis tools, MS excel to produce graphs based on the data. The graphs show the performance of the simulation results clearly so that the performance metrics are compared and analyzed.

5.5. Simulation Scenarios

To evaluate the performance of this work, the existing ADQ, and the new ADQ with different and the same delay bound on NS2. Figure 5.3 below shows network topology that consists of two routers and a number of sources and a sink. The two routers, router A and router B are connected by a link of bandwidth of 1 Mbps and a propagation delay of 1msec. The link from router A to router B is managed by original ADQ and the modified ADQ of AQMs. All the sources of real-time and best effort are linked with router A at bandwidth of 10Mbps.

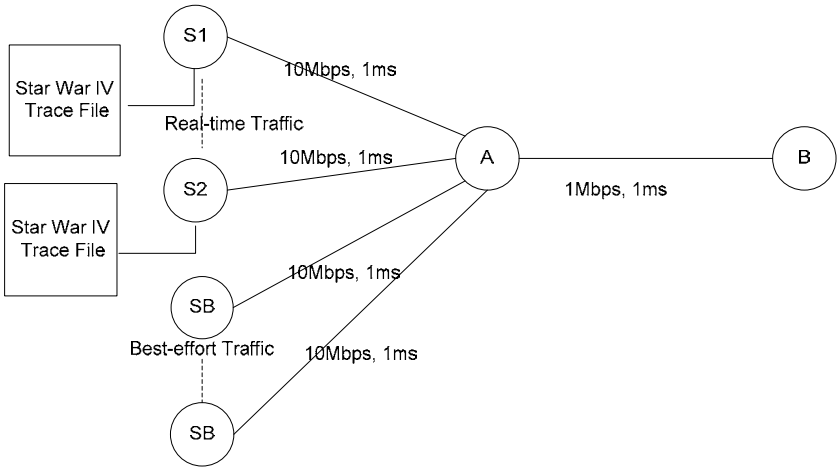

Figure 5.3 Simulation Network Topology.

The experiment is conducted in three different scenarios

1. With only real-time traffic, this helps us to evaluate the performance and complexity of the new algorithm clearly as the modifications of the algorithm only on real-real time traffic.
2. real time traffic and best effort traffic and
3. only best–effort traffic

5.6. Performance Evaluation

To evaluate the performance the proposed algorithm with the original ADQ the following criteria are used for evaluation:

5.6.1. Throughput of conformant traffic

The main task of the ADQ scheduler is delay guaranteeing for transmission of conformant packet, i.e. the throughput of conformant traffic should be identical with the input rate of conformant traffic. This criteria will help us to evaluate the performance the proposed algorithm whether sub-classing based on application delay and existence other traffics such as excess and best-effort traffic affects or not.

5.6.2. Total throughput of real-time traffic

As seen in previous chapter sub-classing conformant and real time traffic based on queuing delay bound reduce early synchronization while transmitting two or more real time applications. However, this is achieved by tradeoff computation time. Total throughput of real time traffic will be used for evaluation of the performance of proposed solution. This consists of all real-time packets, both conformant and excess, that was 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.

5.7. Performance Evaluation and Analysis

Simulation Set 1

The objective of this simulation to evaluate the performance of modified ADQ, i.e. the throughput of conformant packet and utilization remaining bandwidth after transmission of conformant packet.

In this simulation only real-time traffic are used as source of traffic. Each simulation carried out three times by configuring the original ADQ and modified ADQ with same delay bound and different delay bound as an active queue management of Router A. The delay bound for the original ADQ is

configured to 20ms and delay bound for modified version is configured from 20ms to 60ms by increasing 10ms for each source. The figure 5.4 shows the simulation scenario for two real-time applications.

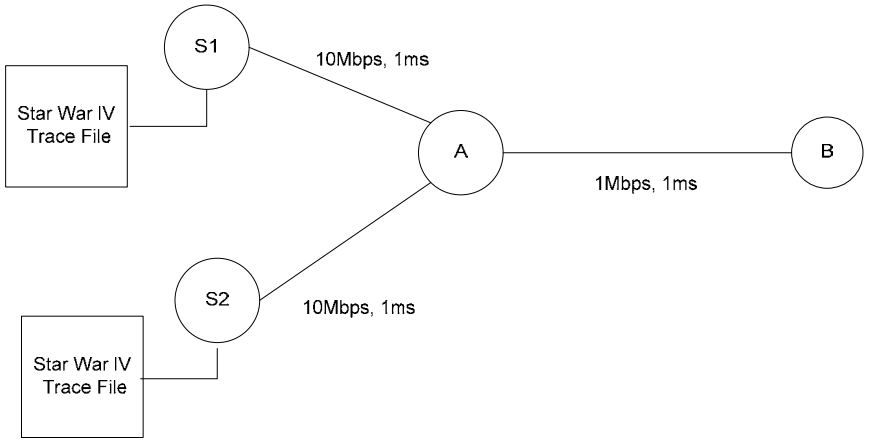

Figure 5.4 Simulation setup 1 with two real-time sources.

Results and Analysis

The following tables (5.1) show results of simulation of two real-time applications by varying the arriving rate of conformant packet from 125kbps to 400kbps. The table and graphs below shows only simulation result of two real-time traffic applications.

Simulation Set	Parameter	Conformant packet arriving rate					
		125kbps	150kbps	200kbps	250kbps	300kbps	400kbps
ADQ (Delay Bound=20ms)	No. Excess drops	37498	54994	89995	109885	119997	135834
	No. Operation	1293722	1233206	1046554	915954	884978	775508
	No. Buffer Access	524988	509140	457969	434658	419993	406102
Modified ADQ with same delay bound.(Delay bound=20ms)	No. Excess drops	35995	53994	89993	107990	118782	134991
	No. Operation	2012600	1904541	1622309	1399425	1324107	1208068
	No. Buffer Access	524988	524986	486183	458708	444960	433401
Modified ADQ (Delay Bound=20ms, 30ms)	No. Excess drops	35993	53993	89991	107992	119977	134989
	No. Operation	2086258	1958341	1672304	1450231	1327465	1215894
	No. Buffer Access	524988	523488	479994	453577	438719	423274

Table 5.1 Simulation result of simulation set 1.

In all of the simulation conformant packets are transmitted before their deadline, this assures that throughout of conformant packet is identical to the arriving rate of conformant packets. Graph in figure 5.5 and figure A.1 show ideal throughput and actual throughput of conformant packets.

Results in table 5.1 and table A.1 shows that with smaller arriving rate of conformant packets; the total number of excess packet drop using the modified ADQ as AQM is smaller than the original ADQ. As discussed in section 4.1.2, during synchronization ADQ has no mechanism to identify excess packets that are associated with transmitting conformant packet of the same application; as a result it removes non expired packets unnecessarily.

Even though the modified ADQ, with different delay bound configuration, treats each real-time traffic packets depends on its delay bound class; the third part(with different delay bound) of the simulation shows that the total number of excess packet drop almost equal to second one (with same delay bound). Only sub-classing with out considering delay bound difference effectively use all the remaining bandwidth; so that delay bound difference could not add significant effect on the throughput of excess packets except supporting different queuing delay requirements.

However; when arriving rate of conformant packets increased, the total excess packet drop of the modified ADQ has no significant difference with the original ADQ. This is because of reduction of residual bandwidth for excess packets as it allocates of most of the bandwidth to serve conformant packets. Excess packet drop because of early synchronization and unnecessarily removal of excess packets because of association of excess packets with conformant of the original ADQ will be reduced as the available remaining bandwidth decreased. Therefore sub-classing of traffics has no significant advantage during fast arriving rate of conformant packets except supporting different delay bound.

Figure 5.6 and figure 5.7 show that the number of operation and buffer access has increased; this is because of complexity of the algorithm to support different delay classes. The following graph shows the performance comparison.

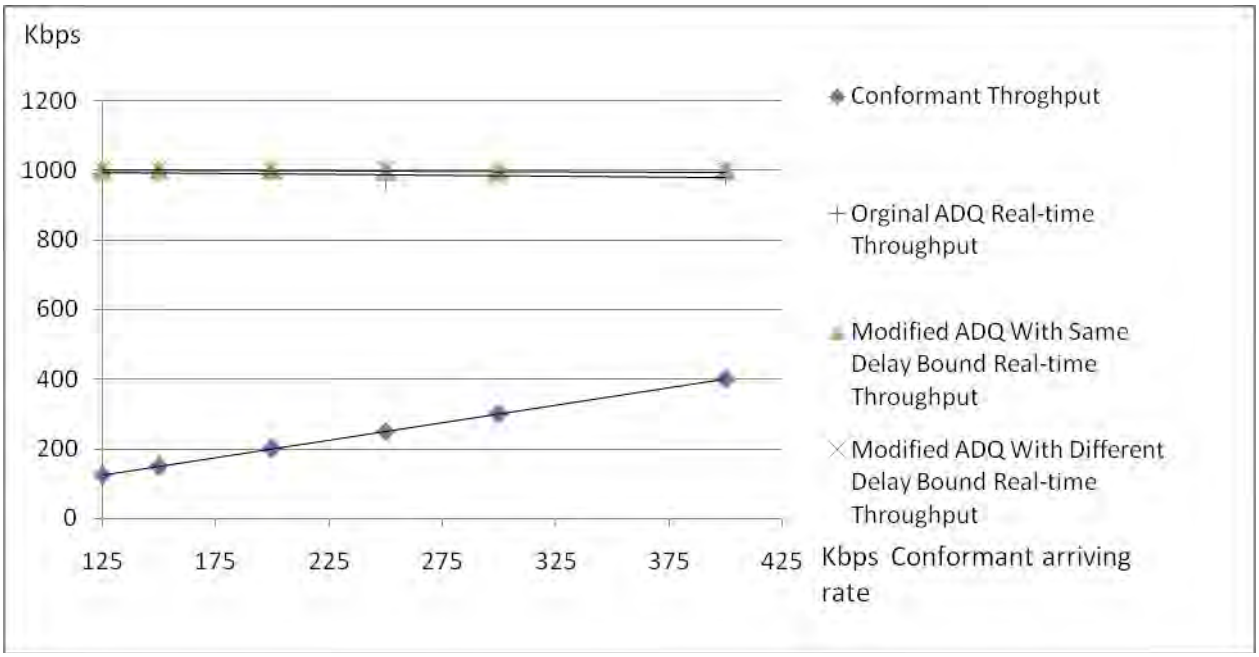

Figure 5.5 Simulation set 1 result graph of real-time traffic throughput.

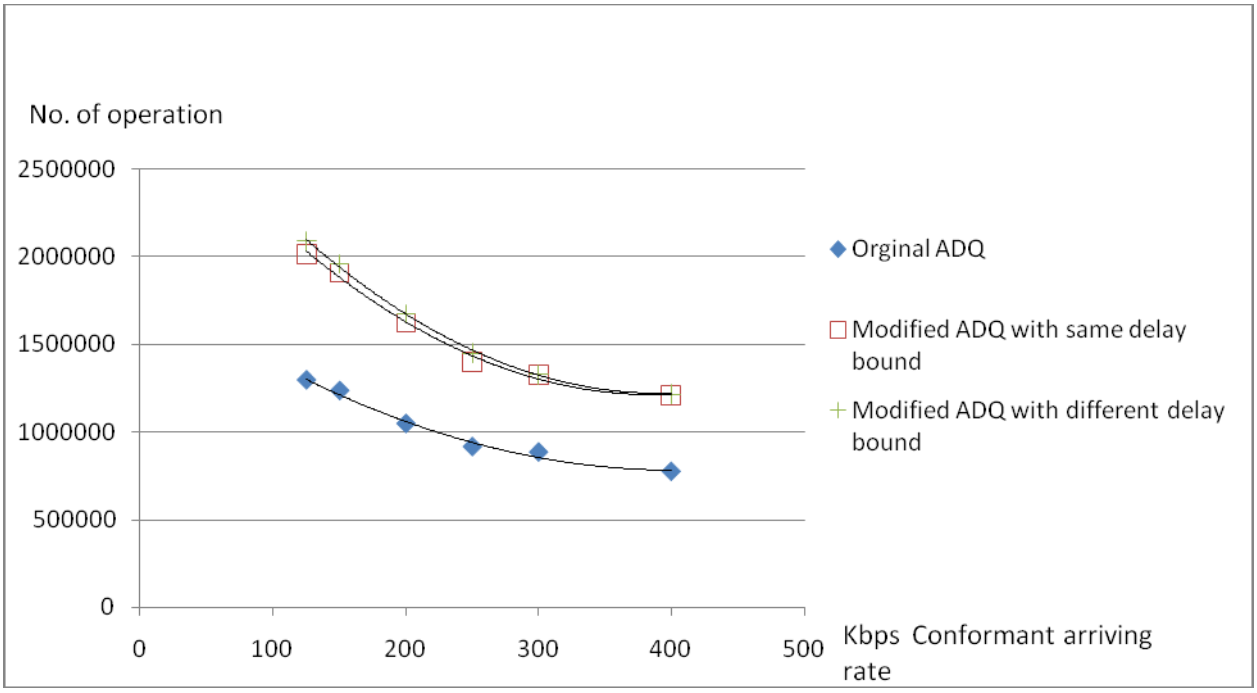

Figure 5.6 Number of operation in simulation set 1

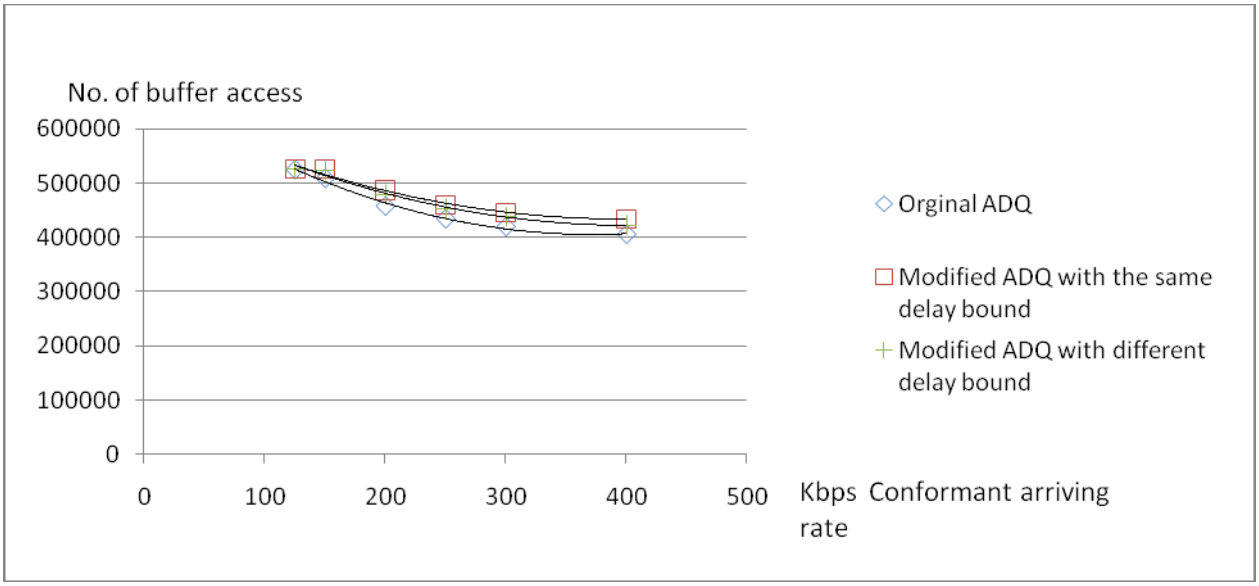

Figure 5.7 Number of buffer access in simulation set 1

Simulation Set 2

This simulation helps to evaluate the performance of the modified version of ADQ, whether the existence of other traffic affect the total and conformant real-time throughput. The simulation setup is the same as simulation set 1 except best-effort traffics added at the source. 5 long-lived FTP flows sources are used as best effort traffic generator that are enough to create congestion in the bottleneck link. Figure below shows network topology of simulation set 2.

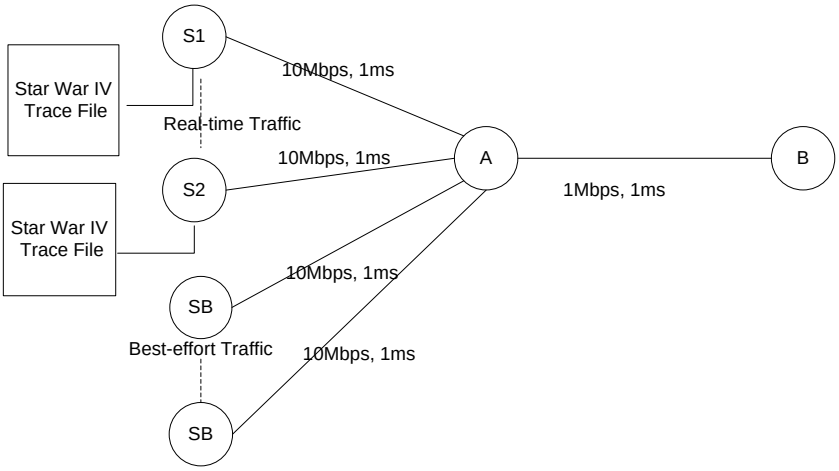

Figure 5.7. Simulation Setup 2.

In this simulation all ADQs are configured to use 40% of the remaining bandwidth after transmission of conformant packet for excess and 60% for best effort traffic.

Results and Analysis

The following figure shows results of simulation set2 with two real-time traffic sources.

Simulation Set	Parameter	Conformant packet arriving rate			
		125kbps	150kbps	200kbps	250kbps
ADQ (Delay Bound=20ms)	No. Excess drops	114713	121901	135456	142843
	No. B-effort Served	77676	67326	45467	34513
	No. Operation	1441067	1323724	1097047	977911
	No. Buffer Access	731568	679710	583573	532276
Modified ADQ with same delay bound.(Delay bound=20ms)	No. Excess drops	114298	121675	135461	142702
	No. B-effort Served	78303	67682	45467	34716
	No. Operation	2320710	2133619	1717582	1500830
	No. Buffer Access	763159	711868	614875	559391
Modified ADQ with different delay bound.(Delay bound=20ms and 30ms)	No. Excess drops	114276	121677	135452	142742
	No. B-effort Served	78267	67682	45408	34755
	No. Operation	2441458	2185036	1797315	1547738
	No. Buffer Access	940289	860428	725280	642268

Table 5.2 Simulation result of simulation set 2.

The simulation result shown in table5.2 shows the existence of best-effort traffic has no effect on throughput of conformant packet, i.e. arrival rate of conformant packet equal to throughput of conformant packet. Throughputs of excess real-time packets using two versions of ADQ haven't shown significant difference compared to number of excess packet drop difference in simulation scenarion1. This is because of the existence of best-effort traffic that reduces the remaining available bandwidth for transmission of excess packets. As the available bandwidth for excess packets reduced, the effect of sub-classing in the modified algorithm such as early synchronization also reduced. Figures below show graphical representation of results.

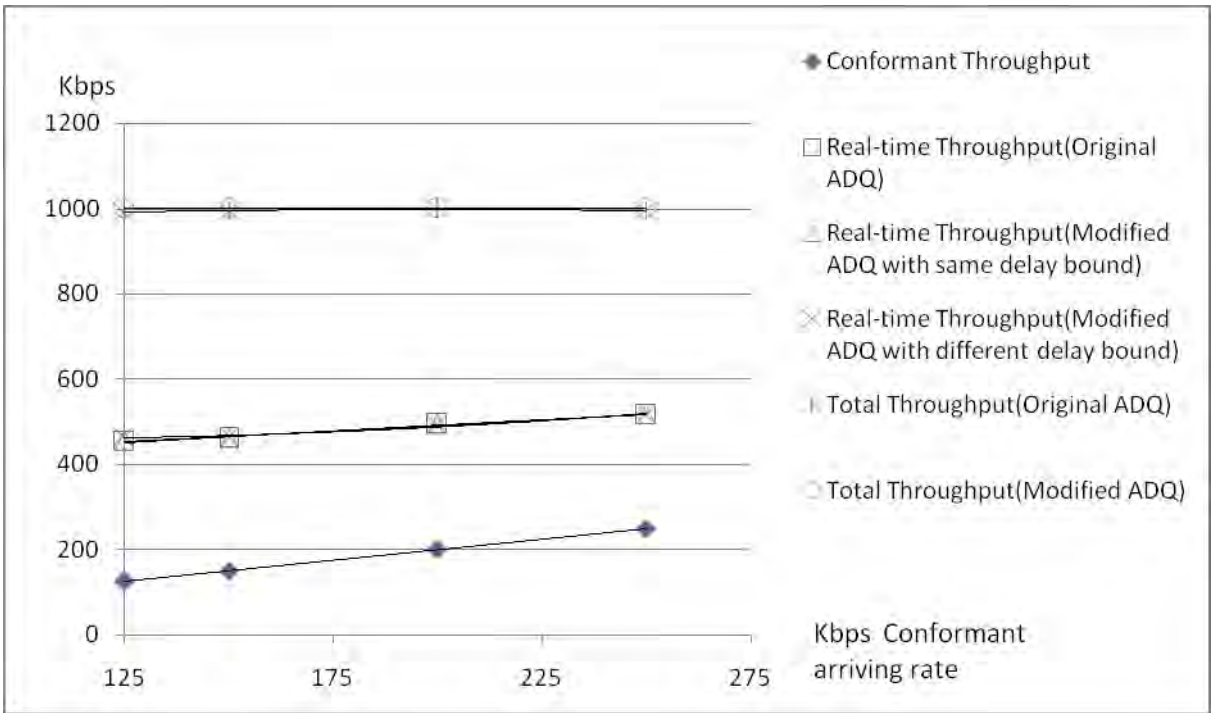

Figure 5.8 simulation result of simulation set 2

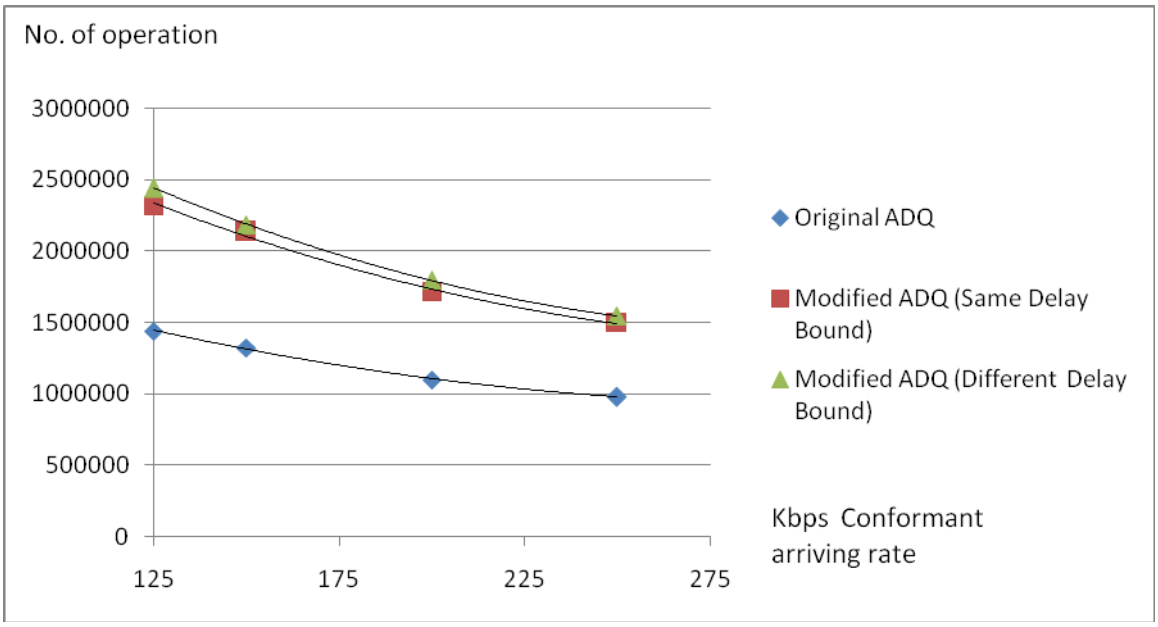

Figure 5.9 Number of operation in simulation set 1

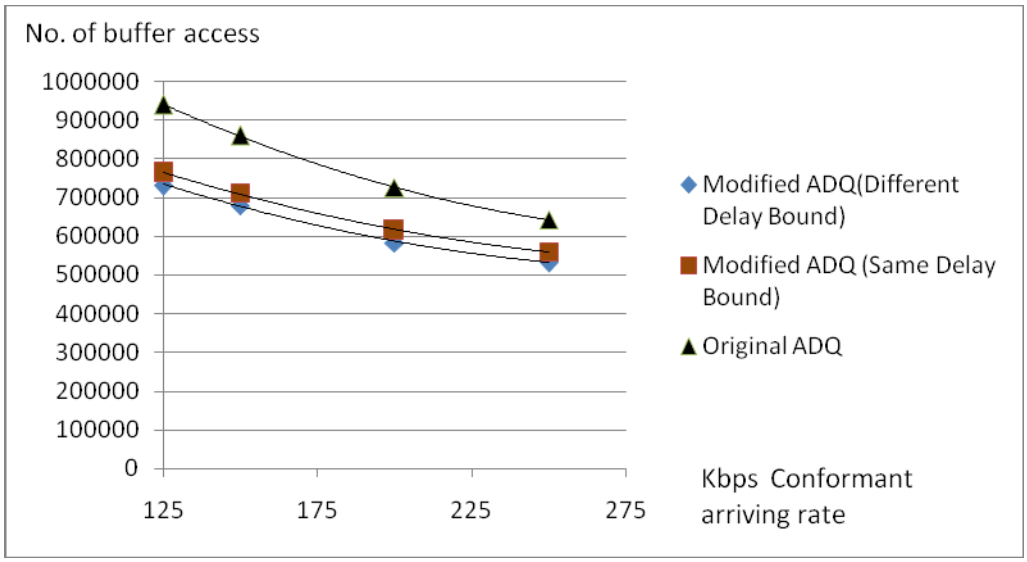

Figure 5.10 Number of buffer access simulation set 1

Simulation Set 3

In this simulation only best-effort traffic is used as source. The objective of this simulation to evaluate the performance of the modified ADQ when there is no real time traffic. Figure 5.11 shows network topology of this simulation.

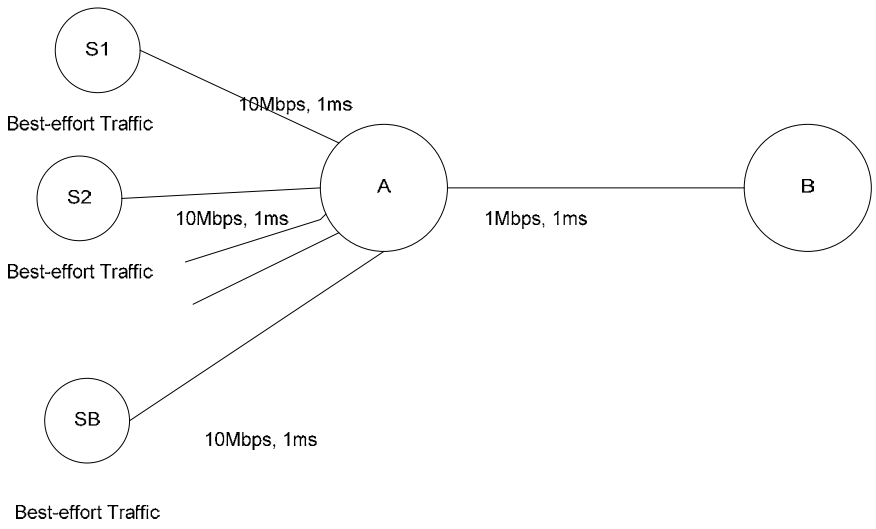

Figure 5.11 Simulation setup 3.

The two routers, router A and router B are connected by a link of bandwidth of 1Mbps and a propagation delay of 1msec. The link between router A and router B is managed by ADQ and Modified ADQ with a queue size of 200 packets. 20% 40% percent of the queue size is allocated for conformant and excess real-time packets respectively in original ADQ configuration. All sources are connected with router A by bandwidth of 10Mbps. The same as in simulation set 2; five long lived ftp traffic source are used as source of best-effort traffic.

Graph in figure 5.12 shows the queue size as a function of time. As can be seen in the graph the original ADQ has lower queue length than the modified one. The original ADQ reserves fixed amount of buffer for real time traffic even if there is no real time traffic. Waiting for real-time traffic, the original ADQ wastes resource. However; the modified ADQ has a capability of dynamically adjust queue sizes for each traffic class and sub-classes and thus it utilizes the available buffer effectively.

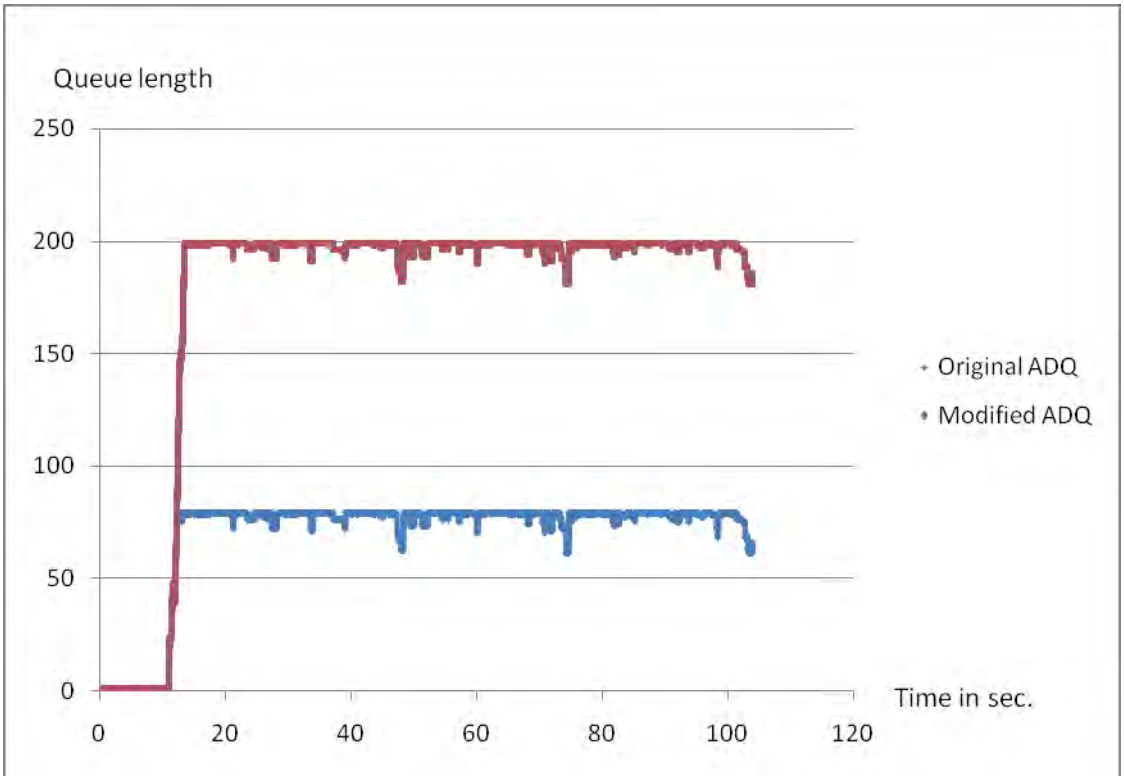

Figure 5.11 queue size

Chapter Six

6. Conclusion and future work

This thesis presents an extension of Active Drop Queue active queue management; ADQ was designed for queuing delay guarantee in the form of traffic contract with the assumption that all delay sensitive traffics requires the same constant queuing delay bound. ADQ is based on buffer management and scheduling, the scheduler is responsible for delay guaranteeing and the link sharing while the buffer manager is responsible on removing expired excess packets. To do this ADQ associates every arriving excess packet with a conformant packet.

This work is on incorporating different queuing delay bound in the original ADQ. We have proposed a modified version of ADQ; sub-classing each real–time traffic class (conformant and excess) based on queuing delay requirements with dynamic queue size for each sub-class. We have evaluated the performance and complexity of the modified ADQ and compare the results with the original ADQ simulations.

Even though Sub-classing of traffic classes makes implementation of scheduling and buffer management algorithms complex; in all simulation we have seen that the overall performance of the modified version ADQ is not reduced significantly. The first two simulations sets results show that the total number of excess packet drop in the modified version of ADQ is less than the original as the original ADQ has a problem in association excess with conformant packets and constant delay bound assumption while serving two or more real-time applications. As it servers more excess packets the original ADQ, the number of buffer access in modified ADQ has also increased. However the number operation has increased by larger number in the modified ADQ; currently machine computation speed is too high to suppress computation time issue.

The dynamic class queue size implementation of makes the modified ADQ to utilize its entire buffer; while there is no real time traffic. However, the original ADQ wastes some amount of buffer space as it allocates during compiling time for real time traffic.

I have successfully incorporated different queuing delay bound with out degradation of the performance by sub-classing conformant and excess traffics.

Recommendation for Future Work

The extension of this work can be in determining the optimum number of sub-classes and delay bound of each class. This requires studying the behavior of real-time multimedia applications such as voice, video, online gaming etc. over an internet. In addition to this studying about network topology and traffic growth particularly at the back bone level is also required.

7. Reference

1. D.Chen, Daqing Gu and J.Zhang," Supporting Real-time Traffic with QoS in IEEE 802.11e Based Home Networks", MERL – A MITSUBISHI ELECTRIC RESEARCH LABORATORY, TR-2004-006 February 2004.
2. 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.
3. J. Chung, "Dynamic-CBT – Router Queue Management for Improved Multimedia Performance on the Internet", Msc. thesis, Worcester Polytechnic Institute, May 2000.
4. Lirong He, Ian Rogers, Lisha He," A NEW REAL-TIME MULTIMEDIA ONTROL PROTOCOL FOR DISTANCE LEARNING", Msc. Thesis, University of Manchester, 2005.
5. L.Le, J.Aikat, K.Jeffay and F.D. Smith," The Effects of Active Queue Management on Web Performance", SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany.
6. Mohd Farhan Ngatman1 , Md Asri Ngadi2 , Johan M. Sharif3 " Comprehensive Study of Transmission Techniques for Reducing Packet Loss and Delay in Multimedia over IP", IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.3, March 2008
7. M.Kim, "Proportional Integrator with Short-lived flow Adjustments", Msc. thesis, WORCESTER POLYTECHNIC INSTITUTE,2004.
8. M. Claypool, R. Kinicki , and A. Kumar, "Traffic Sensitive Active Queue Management", Msc. thesis, Worcester Polytechnic Institute, 2005.
9. 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.
10. R. L. Cruz, "SCED+: Efficient management of quality of service guarantees," in Proc. IEEE INFOCOM'98, 1998.
11. [RFC2309] B. Braden etc., "Recommendations on Queue Management and Congestion Avoidance in the Internet", Apr. 1998.
12. R. Gu´erin, S. Kamat, V. Peris, and R. Rajan, " Scalable QoS Provision Through Buffer Management", IBM, T.J. Watson Research Center USA ,2000.

13. S. Kunniyurz,"An Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management", December 2002.
14. S. Golestani, "A self-clocked fair queuing scheme for broadband applications," in Proc. IEEE INFOCOM'94, 1994.
15. S.Wang, D.Xuan, R.Bettati, and W. Zhao , "Providing Absolute Differentiated Services for Real-Time Applications in Static-Priority Scheduling Networks ",IEEE/ACM TRANSACTIONS ON NETWORKING, April 22-26, 2001.
16. Srisankar Kunniyur and R. Srikan ", ABE Stable, Scalable, Fair Congestion Control and AQM Schemes that Achieve High Utilization in the Internet", IEEE Network, May/June 2001.
17. T. korkeakoulu," PERFORMANCE EVALUATION OF MULTICAST NETWORKS AND SERVICE DIFFERENTIATION MECHANISMS IN IP NETWORKS", Helsinki University of Technology Networking Laboratory, 2005.
18. V. Phirke , "Traffic Sensitive Active Queue Management for Improved Quality of Service", Msc. thesis, Worcester Polytechnic Institute, May 2002.
19. X. Wang, "Active Queue Management for Real-time IP Traffic", Worcester Polytechnic Institute, PhD Thesis, University of London, October 2006.
20. Y. Huang, R. Guerin, and P. Gupta, "Supporting Excess Real-time Traffic with Active Drop Queue", Univ. of Pennsylvania, Tech. Rep., 2005.
21. Internetworking Technologies Handbook, 4th Edition, Cisco Systems, Inc, 2003.
22. "The Impact of Packet Voice Transport on Network Echo Cancellers", Ditech, March 2005.
23. "Supporting Real-time Traffic Preparing Your IP Network for Video Conferencing", White Paper, Global Services, January 5, 2006.
24. Y. Huang and P. Gupta. (2002) Adq. [Online]. Available: http://einstein.seas.upenn.edu/mnlab/softwares/ADQ.html.
25. Network Simulator ns-2, see http://www.isi.edu/
26. Y. Huang and P. Gupta, "ADQ," http://einstein.seas.upenn.edu/mnlab/softwares/ADQ.html.

Appendix

A. Simulation Result of Three Applications

Simulation Set	Parameter	Conformant packet arriving rate			
		125kbps	150kbps	200kbps	250kbps
ADQ (Delay Bound=20ms)	No. Excess drops	74993	111478	149986	179997
	No. Operation	1903328	1679371	1462263	1323590
	No. Buffer Access	782480	721002	673560	644107
Modified ADQ with same delay bound.(Delay bound=20ms)	No. Excess drops	71994	107993	147591	179992
	No. Operation	2893362	2659200	2344735	2017683
	No. Buffer Access	785982	779110	722594	659493
Modified ADQ with different delay bound.(Delay bound=20ms ,30ms and 40ms)	No. Excess drops	71991	107990	147588	179989
	No. Operation	3063271	2807315	2400415	2124146
	No. Buffer Access	784482	776356	713698	669109

Table A.1 Simulation output of three real time applications.

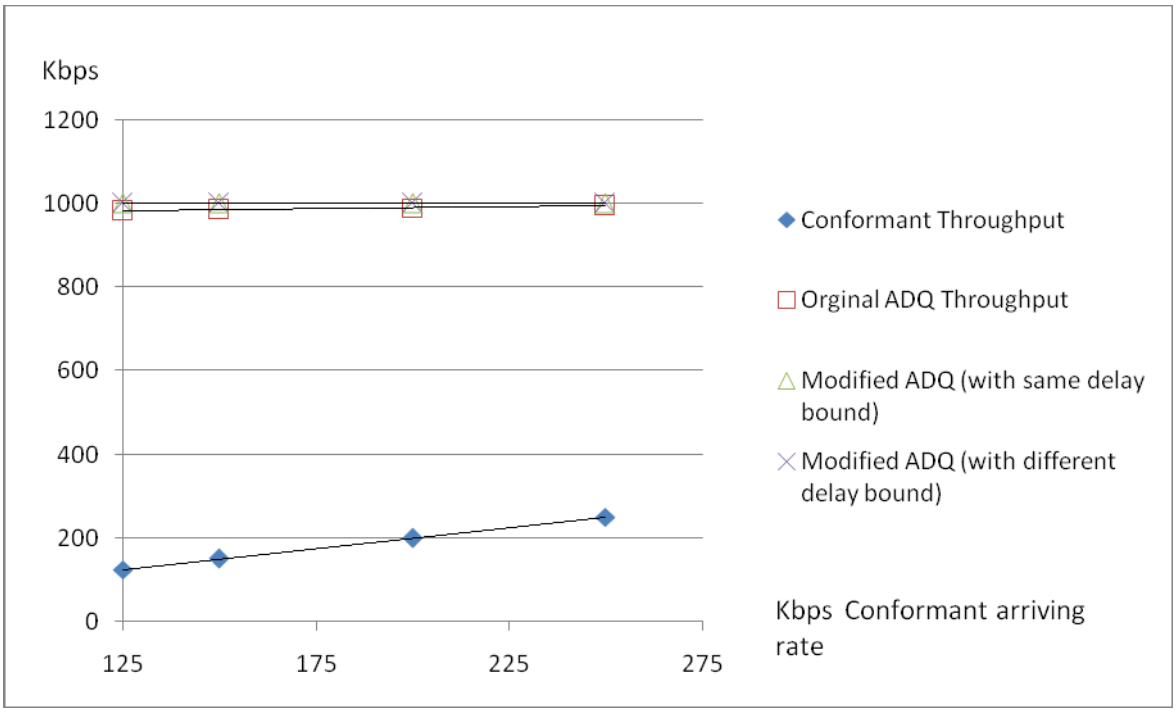

Figure A.1 Simulation set 1 result graph of real-time traffic throughput (three real-time applications).

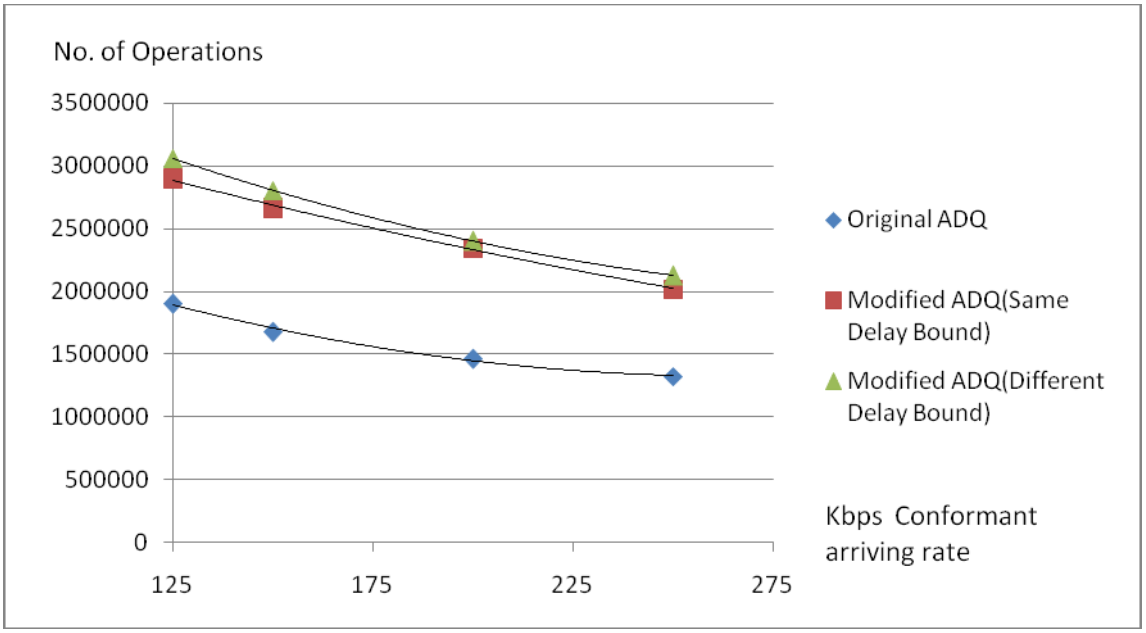

Figure A.2 Number of operations in simulation set 1(three real-time applications).

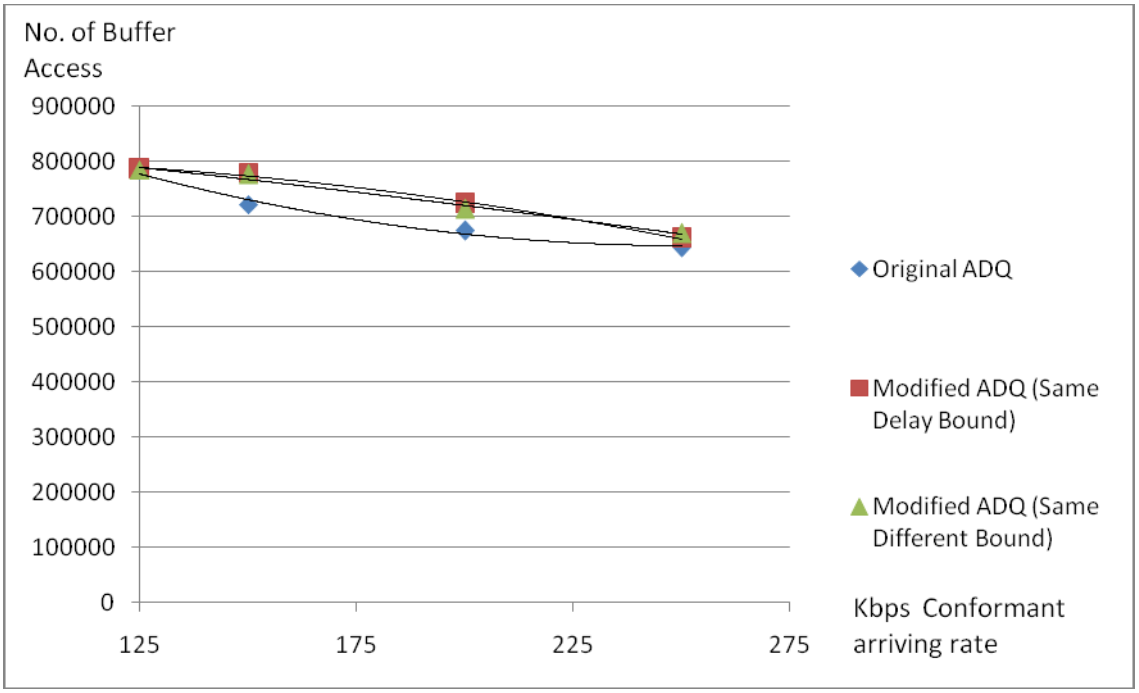

Figure A.3 Number of buffer access in simulation set 1(three real-time applications).

B. Source Code of Modified ADQ

AdQ.h

```
#include "queue.h"
#define NO_SUBCLASS 3    // no. of sub classes
#include "constants.h"
struct segQueue{         // Segment structure
       long front;
       long rear;
       long segindex[10000];
                  long syncindex[10000];
           };
class ADQueueG : public Queue {
 public:
        ADQueueG()
           {
              qBestEffort  = new PacketQueue;
              BestEffortStatus=0;
              for (int i=0; i< NO_SUBCLASS; i++)         // initialize sub-classes
                {
                   qConformant[i]= new PacketQueue;
                   qExcess[i]    = new PacketQueue;

                   seg[i].rear=0;   // initialize segment front and rear
                   seg[i].front=1;
                  ce[i]=cd[i]=ee[i]=ed[i]=0; // set current index of packet queue 0
                   typeIndicator[i]=2;       // set last enqueued packet 2=nothing
                   confStatus[i]=ExcessStatus[i]=0;  // set queue length 0
               }
              startFrom=0;
              Q_limit= 200;    // legth queue limit
              adqDelay[0]=0.02; // Delay bound of class 0 20ms,1:30ms, 2:40ms ......
              adqDelay[1]=0.03;
              adqDelay[2]=0.04;

              er_ =4;                //excess ratio
              br_ = 6;               //Best-effort ratio
              excess_vt=besteffort_vt=sys_vt=0;   //Set virtual time =0

              strcpy(opfile, "adqoutNewC.txt");    //output trace file
              opFile = fopen(opfile, "w");
             openfile=1;

              strcpy(opfileA, "adqArrivals.txt");   //arrivals trace file
              opFileA = fopen(opfileA, "w");
             openfileA=1;

            printf( "\n-----------------------------------------------------\n\n");
            printf( " Modefied Active Drop Queue [ ns-2 implementation]\n");
            printf( " ns-2 implementation by Mehari Teklie <mehari@gmail.com>\n");
            printf( " Orginal ADQ algorithms proposed by Yaqing Huang ,Roch Guerin
           and Pranav Gupta.\n");
            printf( "\n Simulation result at ns-allinone-2.32/bin folder. \n");
            printf( " Addis Ababa University \n");
            printf( " 2008\n");
            printf( "-----------------------------------------------------\n\n");

             strcpy(opfileEQ, "adqGEnStatE.txt");  //Enqueuing statistics
             opFileEQ = fopen(opfileEQ, "w");
             openfileEQ=1;

             strcpy(opfileDQ, "adqGDqStatD.txt");  //Dnqueuing statistics
            opFileDQ = fopen(opfileDQ, "w");
             openfileDQ=1;

             fprintf( opFileEQ ,"*********************************\n");
```

```
            fprintf( opFileEQ ,"Queue Packets Status During Enquing \n");
            fprintf( opFileEQ ,"Time  Class Conformant Excess     Best-Effort \n");
            fprintf( opFileDQ ,"***********************************\n");
            fprintf( opFileDQ ,"Queue Packets Status During Dequing \n");
            fprintf( opFileDQ ,"Time  Class Conformant Excess     Best-Effort \n");
        };

     void enque(Packet*);
     Packet* deque();
protected:

    PacketQueue*  qConformant[NO_SUBCLASS];   // conformant queue
    PacketQueue*  qExcess[NO_SUBCLASS];       // excess queue
    PacketQueue*  qBestEffort;                // best-efort queue

    long Q_limit;  /*max queue size in packets */
    int cl;                                   // sub -class nomber
    int startFrom;                        // turn no. during dequing excess queue

    /* enqueuing and dequeing index of conformant and best-effort traffic */
    long ce[NO_SUBCLASS];   // conformant enqueuing index
    long cd[NO_SUBCLASS];   // conformant dequeuing index
    long ee[NO_SUBCLASS];   // excess enqueuing index
    long ed[NO_SUBCLASS];   //  excess dequeuing index

    struct segQueue seg[NO_SUBCLASS];  // segment structure
    int typeIndicator[NO_SUBCLASS];    // type of last enqueued packet

    int getNotExpiredExcess(int i);//find non expired excess packet of sub-class i from
                                   //HOL
    int cleanup(int i);                // clean up procedure
    int OthersEmpety(int i);           // checks if other queues are emety
    int NoExcessBefore(int i);
    Packet* DequeConformant ();        // dequeue conformant
    Packet* DequeNonConformant ();
    Packet* DequeExcess(int i);
    double elgibilityTime(double arTime, int i);  // calculate elgibility time

    long confStatus[NO_SUBCLASS];         // conformant queue status in mumber
    long ExcessStatus[NO_SUBCLASS];       // excess queue status in mumber
    long BestEffortStatus;                // best-effort queue status in mumber
    double adqDelay[NO_SUBCLASS];       // Queuing delay bound of each sub-class

    long excess_vt;                       // excess virtual time
    long besteffort_vt;                   // best-effort virtual time
    long sys_vt;                          // system virtual time
    int er_ ;                             // excess ratio
    int br_ ;                             // best-effort ratio

   /* Variables for statistic */
    long cArrival[NO_SUBCLASS];
    long eArrival[NO_SUBCLASS];
    long beArrival;
    long cServed[NO_SUBCLASS];
    long eServed[NO_SUBCLASS];
    long beServed;
    long cDrops[NO_SUBCLASS];
    long eDrops[NO_SUBCLASS];
    long beDrops;
    long eFullDrops[NO_SUBCLASS];
    long eExpiredDrops[NO_SUBCLASS];
    long bufAccess;
    long numOp;


    char opfile[100];                 //out put files
    FILE *opFile;
    int openfile;
```

```
        char opfile1[100];
        FILE *opFile1;
        int openfile1;

        char opfileA[100];
        FILE *opFileA;
        int openfileA;

        char opfileEQ[100];
        FILE *opFileEQ;
        int openfileEQ;

        char opfileDQ[100];
        FILE *opFileDQ;
        int openfileDQ;
  };

#endif
```

ADQ-G.cc, Enqueuing and Dequeuing

```
#include "adq_G.h"

static class ADQueueGClass : public TclClass {
public:
        ADQueueGClass() : TclClass("Queue/ADQ-G") {}
        TclObject* create(int, const char*const*) {
                return (new ADQueueG);
       }
} class_active_drop_g_queue;


void ADQueueG::enque(Packet* p)
{
  hdr_cmn* cmh = hdr_cmn::access(p);

  switch(cmh->trafficclass_){  // checks the header to identify traffic class
                               // conformant, excess or best-effort
     case 1:
          cl =cmh->delay_class_ ;     //    Identify conformant packet delay Class
          cArrival[cl]++;             //    Increment the arrival counter
          numOp++;                    //    Increment number of operations

          //If No. of conformant packets exceed, drop the incoming
          if( (confStatus[cl]+ 1) > c_limit )
             {
                 numOp++;
                 drop(p);              // drop the incoming packet
                 cDrops[cl]++;         // Increment the drop count
                 break;
              }
           confStatus[cl]++;           // Increment Conformant Queue Length
           ce[cl]++;                   //increment the enqueued packet index

           if( typeIndicator[cl] == 1){ /*last enqueued packet excess */
                 numOp++;
                          //  update sync index excess pointer of conformant packet
                 seg[cl].syncindex[seg[cl].rear]=ce[cl];
            }
           typeIndicator[cl] = 0;    // set last enqueued packet conformant packets
           numOp++;
                                     // calculate eligibility time
           cmh->eligible_time_=elgibilityTime( Scheduler::instance().clock() ,cl) ;
           cmh->ts_ = Scheduler::instance().clock();    // put time stamp
           bufAccess++;                // increment no. buffer acess

           qConformant[cl]->enque(p);  // enqueue the packet
           Q_len++;                    // increment enqueued packet length
           break;


     case 2:              /*if the arriving packet is Excess */
          cl =cmh->delay_class_ ;     //    Identify excess  packets delay Class
          eArrival[cl]++;             //    Increment arrival no. excess packets
          numOp++;

         if( (Q_len + 1) > Q_limit  ){   //if the queue
             int ned=cleanup(cl);        // call clean up and drop some packets
             eDrops[cl]=eDrops[cl]+ned;  // increment no. of excess drops
             eFullDrops[cl]=eFullDrops[cl]+ned; //increment no. of excess drops of Q. full

             if (ned==0)                   // if nothing droped
                {                          // drop the incoming
                   if (qExcess[cl]->head()!=NULL)
                       {
```

```
                    bufAccess++;
                    Packet* pp=qExcess[cl]->head();
                    qExcess[cl]-> remove(pp);
                     drop(pp);
                    ExcessStatus[cl]--;
                    ed[cl]++;
                    eDrops[cl]++;
                    eFullDrops[cl]++;
                       Q_len--;
                   }
              }
         }
         ExcessStatus[cl]++;       // increment no. of excess packets
         ee[cl]++;                 //increment excess enqueued index
         Q_len++;                  // increment length of queue

          if( typeIndicator[cl] != 1)
               {  /*last enqueued packet is not Excess packet or the queue was empty
                  create  a new segment     */
                   numOp++;
                   seg[cl].rear++;
                   seg[cl].segindex[seg[cl].rear]=ee[cl];
                }
          else
               { /*last enqueued packet is Excess packet or the queue was empty */
                //    update segment index exess pointer

                 seg[cl].segindex[seg[cl].rear]=ee[cl];
                 numOp++;
                 }

           typeIndicator[cl] = 1;  // set last enqueued packet excess
           numOp++;
           cmh->ts_ = Scheduler::instance().clock();

           bufAccess++;
           qExcess[cl]->enque(p);  // enqueue the packet
           break;

case 3:   // if the arriving packet is best-effort
    qBestEffort->enque(p);  // enqueue the packet
    BestEffortStatus++;     // increment enqueued best-effort packet no
    beArrival++;            // increment best-effort arrivals
    bufAccess++;            // increment buffer access
    numOp++;                // increment no. of operations
     Q_len++;                // Increment queue length
    cl=100;

    if (Q_len +  1 > Q_limit ) {  // if queue full drop the incoming
        qBestEffort->remove(p);
        drop(p);
        BestEffortStatus--;
         Q_len--;
        beDrops++;
        bufAccess++;
      }
     break;
default:
    BestEffortStatus++;     // increment enqueued best-effort packet no
    beArrival++;            // increment best-effort arrivals
    bufAccess++;            // increment buffer access
    numOp++;                // increment no. of operations
     Q_len++;                // Increment queue length
    cl=100;


    cl=100;
    if (Q_len +  1 > Q_limit ) {  // if queue full drop the incoming
```

```
            qBestEffort->remove(p);
            drop(p);
            BestEffortStatus--;
             Q_len--;
            beDrops++;
            bufAccess++;
          }
         break;

      }

   fprintf( opFileEQ ,"%f,%d,%12d,%12d,%12d
\n",Scheduler::instance().clock(),cl,confStatus,ExcessStatus,BestEffortStatus);


}
Packet* ADQueueG::deque()
{

   fprintf( opFile,"= \n");
   for (int j=0;j < NO_SUBCLASS;j++)
    {
     fprintf( opFile,"Q: Time %f,Conformant : %10d  Excess:  %10d  Best-Effort: %10d
\n",Scheduler::instance().clock(),confStatus[j],ExcessStatus[j],BestEffortStatus);
    fprintf( opFile,"A:  C: %10d  E:  %10d  B: %10d  \n",cArrival[j],eArrival[j],beArrival
);
    fprintf( opFile,"S:  C: %10d  E:  %10d  B: %10d  \n",cServed[j],eServed[j],beServed  );
    fprintf( opFile,"D:  C: %10d  E:  %10d  B: %10d  \n",cDrops[j],eDrops[j],beDrops  );
    fprintf( opFile,"D:  F: %10d  E:  %10d  \n",eFullDrops[j],eExpiredDrops[j]  );
    fprintf( opFile,"No.Op.:   %10d  BA :  %10d  \n",numOp,bufAccess);
     }
   Packet* p;
   numOp++;
   for (int i=0;i < NO_SUBCLASS;i++)  // start dequing conformant from sub-class with smallest delay bound
     {
        if (confStatus[i]!=0)   // conformant class i is not empety
             { numOp++;
                // read header information
               hdr_cmn* dph= hdr_cmn::access(qConformant[i]->head());
               // if it is eligible for transmission
               if ( dph->eligible_time_ <  Scheduler::instance().clock() ||  (
                     ExcessStatus[i]==0 && OthersEmpety(i)==1 ) || NoExcessBefore(i)==1)
                   { numOp++;
                      if  (cd[i] +1 == seg[i].syncindex[seg[i].front]) // if there is
                     excess packet arrived before if
                   {
                     int dc= cleanup(i);                 // remove expired excess packets
                     eDrops[i]=eDrops[i]+dc;
                     eExpiredDrops[i]=eExpiredDrops[i]+dc;

                    }
                     fprintf( opFile1,"D,C, %f,%f%,%f,%d,%d,%d,%d \n",dph->ts_,dph-
>eligible_time_,Scheduler::instance().clock(),dph->frameseq_ ,dph-
>delay_class_,ExcessStatus[i],confStatus[i]);

                     cServed[i]++;       // increase  served conformat packets
                     cd[i]++;            // increase conformant dequeue index
                     confStatus[i]--;    // decrease conformant queue status
                       Q_len--;            // decrease queue length
                     bufAccess++;        // increase number of buffer access
                     p =  qConformant[i]->deque();  // deque conformant
                     return (p);
                 }
              }
      }

   numOp++;
   for (int k=startFrom ;k < NO_SUBCLASS;k++)    // start from last sub-classs
      { numOp++;
        if (ExcessStatus[k]!=0)
```

```
              {
               Packet* Ncp=DequeExcess(k);        // search for non expired excess
class delay i & deque excess
               if (Ncp!=NULL)
                 {
                     startFrom++;                  // change the excess queue turn
                     numOp++;
                     if (startFrom >= NO_SUBCLASS ) // if it reaches to max , start
from 0
                        {   startFrom =0;  }

                      return Ncp;
                    }
             }
       }
    numOp++;
    for (int k=startFrom  -1 ;k >=0;k--)
    {    numOp++;
          if (ExcessStatus[k]!=0)
              {
              Packet* Ncp=DequeExcess(k);        // search for non expired excess
class delay i & deque excess
              if (Ncp!=NULL)
                 {
                   startFrom++;                  // change the excess queue turn
                   numOp++;
                   if (startFrom >= NO_SUBCLASS )  // if it reaches to max , start
from 0
                     {   startFrom =0;  }

                         return Ncp;
                      }
             }
      }

    numOp++;
    if (BestEffortStatus!=0)              // best-effort dequuing
       {
         sys_vt = 100/br_ + besteffort_vt;
         besteffort_vt = sys_vt;

          Packet* pb =  qBestEffort->deque();           //dequeue best effort
         BestEffortStatus--;
          Q_len--;
         beServed++;
         bufAccess++;
         return (pb);
        }
    else { return NULL; }


 }
```