Adaptive Explicit Congestion Notification (AECN) for Heterogeneous Flows

by

Zici Zheng

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

May 2001

APPROVED:

____

Dr. Robert Kinicki, Major Advisor

____

Dr. Micha Hofri, Head of Department

ABSTRACT

Previous research on ECN and RED usually considered only a limited traffic domain, focusing on networks with a small number of homogeneous flows. The behavior of RED and ECN congestion control mechanisms in TCP network with many competing heterogeneous flows in the bottleneck link, hasn't been sufficiently explored. This thesis first investigates the behavior and performance of RED with ECN congestion control mechanisms with many heterogeneous TCP Reno flows using the network simulation tool, ***ns-2***. By comparing the simulated performance of RED and ECN routers, this study finds that ECN does provide better goodput and fairness than RED for heterogeneous flows. However, when the demand is held constant, the number of flows generating the demand has a negative effect on performance. Meanwhile, the simulations with many flows demonstrate that the bottleneck router's marking probability must be aggressively increased to provide good ECN performance.

Based on these simulation results, an Adaptive ECN algorithm (AECN) was studied to further improve the goodput and fairness of ECN. AECN divides all flows competing for a bottleneck into three flow groups, and deploys a different max_p for each flow group. Meanwhile, AECN also adjusts min_{th} for the robust flow group and max_{th} to get higher performance when the number of flows grows large. Furthermore, AECN uses mark-front strategy, instead of mark-tail strategy in standard ECN. A series of AECN simulations were run in ***ns-2***. The simulations show clearly that AECN treats each flow fairer than ECN with the two fairness measurements: Jain's fairness index and visual max-min fairness. AECN has fewer packet drops and alleviates the lockout phenomenon and yields higher goodput than ECN.

Key words: ECN, RED, AECN, heterogeneous flows, fairness, goodput.

ACKNOWLEDGEMENTS

Throughout my graduate student career, I’ve been fortunate enough to have the help and support from many people. I would especially like to thank:

- Professor Robert Kinicki, my research advisor, for giving me a chance to have an insight in his research field of network congestion control, and for his patience, advice and guidance for helping me finish this thesis in time in the past year.
- Professor Mark Claypool, my thesis reader, for his insightful comment and advice on this thesis, and for his *nile* server.
- Professor David Brown, and Ph. D. student Janet Burge, for their allowing me to use the equipments in AI lab, which enabled me to do a lot of experiments.
- cc (congestion control) group members for their many useful and insightful discussions about network congestion control mechnisms. Special thanks to Jae Chang for his technical and academic help.
- My friends here, Jun Chen, Luping Quan, and Yan Huang, …

Finally, I would most like to thank my parents, my grandmother, my sister and brothers in China for their continual encouragement, guidance and support. Specially, I’d most like to thank my dear wife, Min Gao, for her generous and considerate support in the past two years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

1.1 Motivation and Goal

TCP is the dominant transport protocol used in the Internet today [THO97]. While Internet traffic continues to grow, it becomes more challenging to provide good throughput to millions of Web customers. When a packet is dropped before it reaches the destination, all of the resources the packet consumes in the transmission will have been wasted. In extreme cases, this situation can lead to congestion collapse [JAC88]. In the past decade, TCP and its congestion control mechanisms have been used in controlling packet loss and preventing congestion collapse across the Internet. Several variants of TCP (Tahoe, Vegas, Reno and NewReno) [FLO96] have been developed to provide host-centric mechanisms to combat high packet loss rates during heavy congestion periods. Traditionally, a router reacts to congestion by dropping a packet in the absence of buffer space. This is referred to as a *TailDrop* router. However, the resulting drop-tail behavior fails to provide adequate early congestion notification and produces bursts of packet drops that contribute to unfair service. Although TCP has built-in techniques (such as Fast Retransmit and Fast Recovery) to minimize the impact of losses from a throughput prospective, these mechanisms are not intended to help applications that are in fact sensitive to the delay or loss of one or more individual packets [STE97] [RAM99]. So, optimizing the congestion control mechanisms used in TCP has been the focus of numerous studies and undergone a number of enhancements.

Active queue management has been proposed as a solution for preventing losses due to buffer overflow. The idea behind active queue management is to detect incipient congestion early and convey congestion notification to the end-hosts, allowing them to back off before queue overflows and packet-drop occurs. Random Early Detection (RED), an active queue management technique proposed by Sally Floyd and Van Jacobson [FLO93], maintains an exponentially weighted moving average of the queue length to detect congestion. When the average queue length exceeds a minimum threshold, packets are randomly dropped with a given probability. In the current Internet environment RED is restricted to using packet drops as a mechanism for congestion indication. However, RED can instead mark a packet with an Explicit Congestion Notification (ECN) bit with a given probability when the average queue size is between minimum threshold and maximum threshold of the router queue.

ECN is an end-to-end congestion avoidance mechanism proposed by Floyd and has already been incorporated into RFC2481 [RAM99]. A connection receiving congestion notification in the form of an ECN marking, cuts its congestion window and reduces the slow-start threshold [RAM99] [STE97] in half just as if it had detected a packet loss. The probability that a packet arriving at the RED queue is either dropped or marked depends on the average queue length, the time elapsed since the last packet was dropped, and an initial probability parameter value. When the average queue length exceeds a maximum threshold, all packets are dropped.

Since the introduction of RED, many researchers have done investigations about the behaviors and performances of RED and ECN, and proposed a variety of enhancements and changes to router management to improve congestion control [RAG99] [FLO91] [AHM99] [FEN97]. But these previous researches usually considered only the following two cases separately: (1) the network with a small or medium number of competing TCP flows, (2) the network with homogeneous flows. The behavior of RED and ECN congestion control mechanisms in TCP network with many competing heterogeneous flows in the bottleneck link, hasn't been sufficiently explored. This is one of the motivations for this thesis. Hence, one goal of this thesis is to investigate the behavior and performance of RED with ECN congestion control mechanisms with many heterogeneous[1] TCP Reno flows using the network simulation tool, ***ns-2***. By comparing the simulated performance of RED routers and ECN routers, this study finds that ECN does provide better goodput and fairness than RED for heterogeneous flows. When the demand is held constant, the number of flows generating the demand has a negative effect on performance. Meanwhile, the simulations with many flows demonstrate that the bottleneck router's marking probability must be aggressively increased to provide good ECN performance.

[1] Heterogeneous flows differ only in their end-to-end round-trip times (RTTs) in this study.

Based on these simulation results, this study proposes an adaptive version of ECN to further improve for ECN on the goodput or throughput and fairness by properly adjusting the relevant ECN parameters. Thus, an adaptive version of ECN (AECN) is the other goal for this study. ECN parameters include maximum drop probability (max_p), maximum threshold (max_{th}) and minimum threshold (min_{th}) for the queue, the average queue size (*avg*), and the weighting factor for the average queue length computation (w_q). AECN divides all flows competing for a bottleneck into three flow groups, and deploys a different max_p for each flow group so that a fragile flow can have higher chance to get a proper share of bandwidth when competing with a robust flow. AECN also adjusts min_{th} for the robust flow group and max_{th} to get higher performance when the total number of flows changes. Furthermore, AECN uses mark-front strategy, instead of mark-tail strategy used in standard ECN[1], to mark the first unmarked packet of a corresponding flow group in the router queue to reduce the queue delay and speed up the notification of congestion to a sender. The simulation results show that AECN achieves better goodput and fairness than standard ECN in the network with many heterogeneous flows.

1.2 Structure of Thesis

This thesis is organized as follows. Chapter 2 presents the background of congestion control mechanisms in TCP/IP network and a summary of the related work in this area, and introduces the current implementation of ECN. Chapter 3 describes the simulation methods deployed in our experiments, including the performance metrics (goodput, throughput, fairness and delay) that are investigated in our study, simulation tool ***ns-2***, which is widely used in the network research community, and experimental procedures. Chapter 4 explains our performance study of ECN and RED with heterogeneous TCP flows. The various simulation scenarios are investigated for comparing the performance of ECN and RED on the characteristics of goodput, throughput, fairness and delay. Based on these simulation results, in Chapter 5, this study develops an adaptive version of ECN

[1] Standard ECN only marks an incoming packet probabilistically when the average queue size is between max_{th} and min_{th}. If the average queue size exceeds max_{th}, all incoming packets will be dropped at the congested ECN route.

(AECN), and presents the basic algorithm of AECN based on standard ECN and the implementation in ***ns-2*** and the ways for further refining AECN. The evaluation of the performance and behavior of AECN on the key performance indicator is also provided in this chapter. The conclusions of this thesis and the future work are presented in Chapter 6. Moreover, Appendix A and B present the code added for the implementation of AECN in ***ns-2***.

Chapter 2 Background and Related Work

This chapter gives an overview of the past and current work in congestion control and management mechanisms used in TCP/IP networks. Since the first congestion collapse episode in 1986, several variants of TCP (Tahoe, Vegas, Reno, NewReno, and SACK) have been developed and evaluated to provide host-centric mechanisms to combat high packet loss rates during heavy congestion periods. Meanwhile, researchers have proposed new congestion avoidance techniques for Internet routers. This chapter first presents the congestion control mechanisms, including the host-centric and router-centric, in TCP/IP networks, then describes the related work of current congestion control mechanisms, especially the active queue management algorithms, such as RED and ECN.

2.1 Background

2.1.1 TCP Congestion Control Mechanisms

2.1.1.1 End-to-end Congestion Control Issues

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 already been amply demonstrated [FLO00a]. However, these advantages are not without cost. In fact, lack of attention to the dynamics of packet forwarding can result in severe degradation, which caused researchers to develop end-to-end congestion control concerning the following several issues.

During the mid 1980s, the “Internet meltdown” phenomenon was first observed, which is also called “congestion collapse” [JAC88]. Originally, TCP included window-based flow control mechanism as a means for the receiver to control the amount of data sent by a sender. The flow control mechanism was used to prevent overflow of the receiver’s data buffer space available for the TCP connection. In 1986, in order to fix “Internet meltdown”, Jacobson developed the congestion avoidance mechanisms which are now required in TCP implementations. These mechanisms operate in the end-hosts to cause TCP connections to **backoff** during congestion. Those TCP flows are said to be responsive to congestion signals (i.e., packet loss) from the network. It is these TCP congestion avoidance algorithms that are still being used to prevent the congestion collapse of today's Internet.

In addition to the concern about congestion collapse, more concern has also been paid attention to fairness for best-effort traffic. Because TCP backs-off during congestion, a large number of TCP connections can share a single, congested link in such a way that bandwidth should be shared reasonably equitably among similarly situated flows. The issue of fairness among competing flows has become increasingly important for two main reasons. First, using window scaling, individual TCPs can use high bandwidth even over high propagation-delay paths. Second, with the growth of the Web, Internet users increasingly want high-bandwidth and low-delay communications, rather than the leisurely transfer of a long file in the background. The growth of best-effort traffic that doesn't use TCP underscores this concern about fairness between competing best-effort traffic in times of congestion. For the current Internet environment, where other best-effort traffic could compete in a FIFO queue with TCP traffic, the absence of fairness with TCP could lead to one flow "starving out" another flow in a time of high congestion.

Besides the prevention of congestion collapse and concerns about fairness, a third reason for a flow to use end-to-end congestion control can be to optimize its own performance regarding throughput, delay, and loss. In an environment like the current best-effort Internet, concerns regarding congestion collapse and fairness with competing flows limit the range of congestion control behaviors available to a flow.

2.1.1.2 TCP Built-in Techniques

Congestion can occur when data arrives on a big pipe (a fast LAN) and gets sent out a smaller pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs. Congestion avoidance is a way to deal with lost packets [JAC88]. There are two indications of packet loss: a timeout occurring and the receipt of duplicate ACKs. This section presents some of the particulars of TCP congestion control mechanisms:

i). Retransmit Timers

The TCP sender sets a retransmit timer to determine when a packet has been dropped in the network. When the retransmit timer expires, the sender assumes that a packet has been lost, sets *ssthresh* to half of the current window (W), and goes into slow-start, retransmitting the lost packet. If the retransmit timer expires because no acknowledgement has been received for a retransmitted packet, the retransmit timer is also backed-off, that is, doubling the value of the next retransmit timeout interval.

ii). Slow-start

The TCP sender cannot open a new connection by sending a large burst of data (e.g., a receiver's advertised window) all at once. The TCP sender is limited by a small initial value for the congestion window (*cwnd*). During slow-start, the TCP sender increases *cwnd* by the number of ACKs received in a round-trip time. Slow-start ends when the sender's congestion window is greater than the slow-start threshold (*ssthresh*).

iii). Congestion Avoidance

When *cwnd* is less or equal to *ssthresh*, TCP is in slow-start; otherwise TCP is performing congestion avoidance. Slow-start continues until TCP is halfway to where it was when congestion loss occurred, and then congestion avoidance takes over. Slow-start opens the window exponentially and increases *cwnd* by the number of ACKs received in a round-trip time; while congestion avoidance dictates that *cwnd* be increased by at most one per round-trip time when an ACK is received. Figure 2.1 shows an example of how TCP slow-start and congestion avoidance works.

iv). Fast Retransmit and Fast Recovery

Since a TCP sender doesn't know whether a duplicate ACK is caused by a lost packet or just by a reordering of packets, it waits for three duplicate ACKs to be received. Once the sender receives three duplicate acknowledgements, TCP supposes that a packet has been lost. Then the sender retransmits the missing packet, without waiting for a retransmission timer to expire. Meanwhile, the sender sets *ssthresh* to half of the current window, reduces *cwnd* to at most half of the previous *cwnd*.

After fast retransmit sends what appears to be the missing packet, congestion avoidance, but not slow-start is performed. This is fast recovery algorithm. It's an improvement that allows high throughput under moderate congestion.

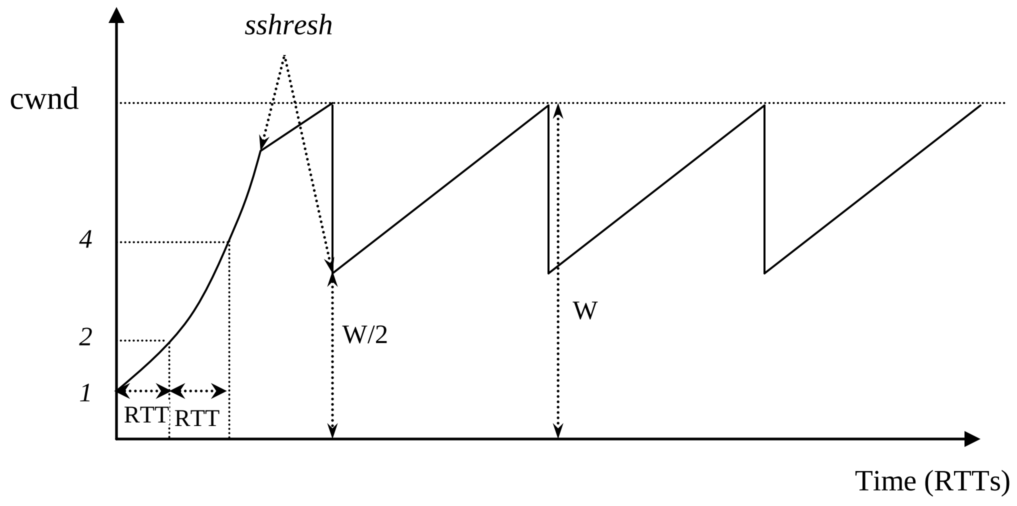

Figure 2.1: An Example of How TCP Slow-start and Congestion Avoidance works

2.1.1.3 TCP Variants

Early TCP implementations followed a go-back-n model using cumulative positive acknowledgement and requiring a retransmit timer expiration to resend data lost during transport [FLO00a]. These TCPs did little to minimize network congestion. Currently, there're several different TCP variants, which have the same specification but different implementations. These TCP variants are Tahoe, Reno, New Reno, SACK, and Vegas. The Tahoe TCP implementation added a number of new algorithms and refinements to earlier implementations. It includes slow-start, congestion avoidance and fast retransmit.

The Reno TCP implementation retained the enhancements incorporated into Tahoe, but modified the Fast Retransmit operation to include Fast Recovery. Reno's Fast Recovery algorithm is optimized for the case when a single packet is dropped from a window of data. The Reno sender retransmits at most one dropped packet per round-trip time. Reno significantly improves upon the behavior of Tahoe TCP when a single packet is dropped from a window of data, but can suffer from performance problems when multiple packets are dropped from a window of data.

The New-Reno TCP includes a small change to the Reno algorithm at the sender that eliminates Reno's wait for a retransmit timer when multiple packets are lost from a window. When multiple packets are lost from a single window of data, New-Reno can recover without a retransmission timeout, retransmitting one lost packet per round-trip time until all of the lost packets from that window have been retransmitted. New-Reno remains in Fast Recovery until all of the data outstanding when Fast Recovery was initiated has been acknowledged.

SACK TCP is another TCP variant [FLO96]. The main difference between the SACK TCP and the Reno TCP is in the behavior when multiple packets are dropped from one window of data. SACK augments TCP's cumulative acknowledge mechanism with additional information that allows the receiver to inform the sender which packets have been missed. By specifying this information, the TCP sender can make more intelligent decision in determining when packets have been lost and in identifying which packets should be retransmitted.

The TCP Vegas [BRA94] [BRA95], proposed by Peterson, L., uses source-based anticipation of congestion by monitoring gap between expected and actual (i.e., measured) throughputs to improve TCP congestion control. It's reported that it gives better 40-70% throughput than Reno.

2.1.2 Active Queue Management

The traditional technique for managing router queue length is to set a maximum length (in terms of packets) for each queue, accept packets for the queue until the maximum length is reached, then drop subsequent incoming packets until the queue decreases because a packet from the queue has been transmitted. This technique is known as "*TailDrop*". This method has served the Internet well for many years, but it has two serious drawbacks:

1. Lockout

   In some situations *TailDrop* allows a single connection or a few flows to monopolize queue space, preventing other connections from getting room in the queue. This lockout phenomenon [BRA98] is often the result of synchronization or other timing effects.

2. Global Synchronization

The *TailDrop* discipline allows queues to maintain a full (or, almost full) status for long periods of time, since it signals congestion only when the queue has become full. It is important to reduce the steady-state queue size, and this is perhaps the queue management's most important goal. The naïve assumption might be that there is a simple tradeoff between delay and throughput, and that the recommendation that queues be maintained in a "non-full" state essentially translate to a recommendation that low end-to-end delay is more important than high throughput. However, this does not take into account the critical role that packet bursts occurs in the Internet. Even though TCP constrains a flow's window size, packets often arrive at routers in bursts. If the queue is full or almost full, an arriving burst will cause multiple packets to be dropped. This can result in a global synchronization of flows throttling back, followed by a sustained period of lowered link utilization, reducing overall throughput. Queue limits should not reflect the steady state queue we want to maintain in the network; instead, they should reflect the size of bursts we need to absorb.

In the current Internet, dropped packets serve as a critical mechanism of congestion notification to end nodes. The solution to the global synchronization is for routers to respond to congestion before their buffers overflow, that is, to employ active queue management, like Random Early Detection (RED) [FLO93] and Explicit Congestion Notification (ECN) [FLO94] [RAM99]. By dropping packets before buffers overflow, active queue management allows routers to control when and how many packets to drop.

2.1.3 RED and ECN

2.1.3.1 RED

RED is a congestion avoidance mechanism implemented in routers that work on the basis of active queue management. RED addresses the shortcomings of *TailDrop*. In contrast to traditional queue management algorithms, which drop packets only when the buffer is full, a RED router signals incipient congestion to TCP by dropping packets probabilistically before the queue runs out of buffer space. This drop probability is dependent on an average queue size to avoid any bias against bursty traffic. A RED router randomly drops arriving packets, with the result that the probability of dropping a packet belonging to a particular flow is approximately proportional to the flow's share of bandwidth. Thus, if the sender is using relatively more bandwidth, it gets more of its packets dropped. The RED algorithm itself consists of two main parts: estimation of the average queue size (*avg*) and the decision of whether or not to drop an incoming packet.

1. Estimation of Average Queue Size

   RED estimates the average queue size, either in the forwarding path using a simple exponentially weighted moving average queue length computation (w_q).

2. Packet Drop Decision

   RED decides whether or not to drop an incoming packet (See Figure 2.2). It's RED's particular algorithm for dropping that results in performance improvement for responsive flows. Two RED parameters, min_{th} and max_{th}, represent thresholds set by RED when to drop a packet. Min_{th} specifies the average queue size below which no packets will be dropped, while max_{th} specifies the average queue size above which all packets will be dropped deterministically (100%). As the average queue size varies from min_{th} to max_{th}, packets will be dropped with a probability pa that varies linearly from 0 to max_p, where pa is a function of the average queue size. As the average queue length varies between min_{th} and max_{th}, pa increases linearly towards a configured maximum drop probability, max_p.

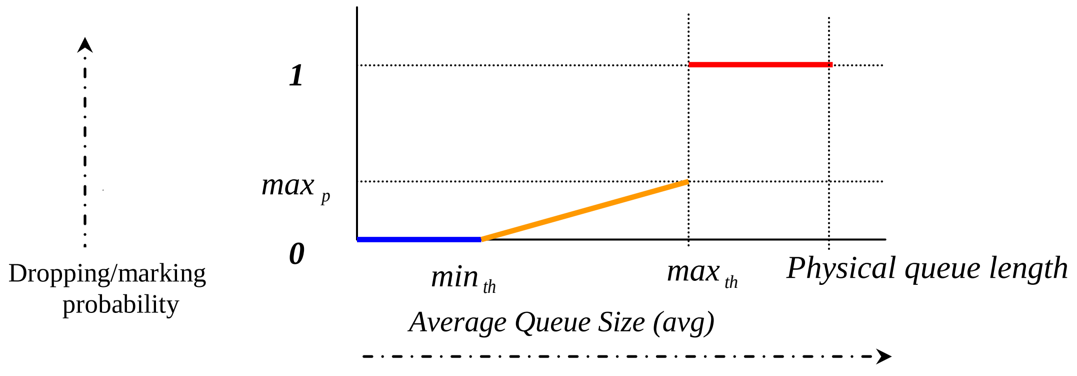

Figure 2.2. Relationship between average queue size and packet marking/dropping probability

Dropping packets in this way ensures that when some subset of the source TCP packets get dropped and they invoke congestion avoidance algorithms that will ease the congestion at the router. Since the dropping is distributed across flows, the problem of global synchronization is avoided.

2.1.3.2 ECN

ECN is an extension to RED [RAM99][RAM01]. It provides a light-weight mechanism for routers to send a direct indication of congestion to the source. When *avg* is between min_{th} and max_{th}, ECN marks, instead of dropping, an incoming packet probabilistically. The marking probability in ECN varies as RED. A connection receiving congestion notification in the form of an ECN marking, cuts its congestion window in half just as if it had detected a packet loss. When *avg* is above or equal to max_{th}, ECN also drops deterministically all incoming packets. Since ECN marks packets before congestion actually occurs, this is useful for protocols like TCP that are sensitive to even a single packet loss. Upon receipt of a congestion marked packet, the TCP receiver informs the sender (in the subsequent ACK) about incipient congestion which will in turn trigger the congestion avoidance algorithm at the sender. ECN requires support from both the router as well as the end hosts, i.e. the end hosts TCP stack needs to be modified. Packets from flows that are not ECN capable will continue to be dropped by RED. There are two main changes that need to be made to add ECN to TCP to an end system and one extension to a router running RED.

2.1.3.2.1 Changes at the router

Router side support for ECN can be added by modifying current RED implementations. For packets from ECN capable hosts, the router marks the packets rather than dropping them (if the average queue size is between min_{th} and max_{th}). It is necessary that the router identifies that a packet is ECN capable, and should only mark packets that are from ECN capable hosts. This uses two bits in the IP header. The ECN Capable Transport (ECT) bit is set by the sender end system if both the end systems are ECN capable (for a unicast transport, only if both end systems are ECN-capable). In TCP this is confirmed in the pre-negotiation during the connection setup phase. Packets encountering congestion are marked by the router using the Congestion Experienced (CE) (if the average queue size is between min_{th} and max_{th}) on their way to the receiver end system (from the sender end system), with a probability proportional to the average queue size following the procedure used in RED routers. Bits 10 and 11 in the IPV6 header are proposed respectively for the ECT and CE bits. Bits 6 and 7 of the IPV4 header DSCP field (Figure 2.3) are also specified for experimental purposes for the ECT and CE bits respectively.

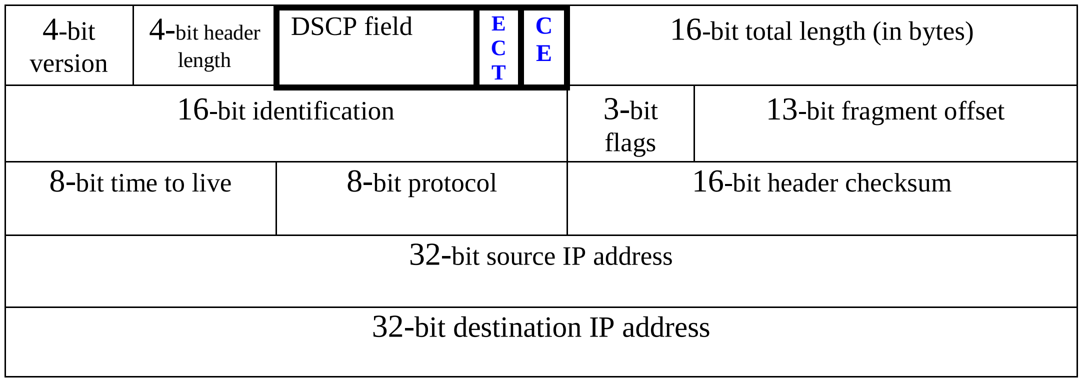

Figure 2.3. IP Header

2.1.3.2.2 Changes at the router TCP Host side

The proposal to add ECN to TCP specifies two new flags in the reserved field (Figure 2.4) of the TCP header. Bit 9 in the reserved field of the TCP header is designated as the

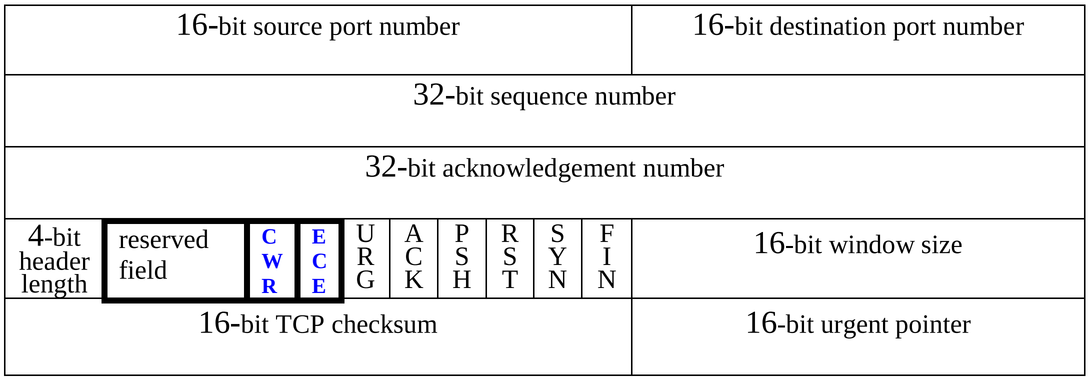

Figure 2.4. TCP Header

ECN-Echo (ECE) flag and Bit 8 is designated as the Congestion Window Reduced (CWR) flag. These two bits are used both for the initializing phase in which the sender and the receiver negotiate the capability and the desire to use ECN, as well as for the subsequent actions to be taken in case there is congestion experienced in the network during the established state.

1. TCP handshake phase

   The source and destination TCP have to exchange information about their desire and/or capability to use ECN. This is done by setting both the ECN-Echo flag and the CWR flag in the SYN packet of the initial connection phase by the sender; on receipt of this SYN packet, the receiver will set the ECN-Echo flag in the SYN-ACK response. Once this agreement has been reached, the sender will thereon set the ECT bit in the IP header of data packets for that flow, to indicate to the network that it is capable and willing to participate in ECN. The ECT bit is set on all packets other than pure ACK's.

2. Packet marking phase

   When a router has decided from its active queue management mechanism, to drop or mark a packet, it checks the IP-ECT bit in the packet header. It sets the CE bit in the IP header if the IP-ECT bit is set. When such a packet reaches the receiver, the receiver responds by setting the ECN-Echo flag (in the TCP header) in the next outgoing ACK for the flow. The receiver will continue to do this in subsequent ACKs until it receives from the sender an indication that it (the sender) has responded to the congestion notification.

3. ACK receipt phase

   Upon receipt of this ACK, the sender triggers its congestion avoidance algorithm by halving its congestion window, *cwnd*, and updating its congestion window threshold value *ssthresh*. Once it has taken these appropriate steps, the sender sets the CWR bit on the next outgoing data packet to tell the receiver that it has reacted to the receiver's notification of congestion. The receiver reacts to the CWR by halting the sending of the congestion notifications (ECE) to the sender if there is no new congestion in the network. Note that the sender reaction to the indication of congestion in the network (when it receives an ACK packet that has the ECN-Echo flag set) is equivalent to the Fast Retransmit/Recovery algorithm (when there is a congestion loss) in NON-ECN-capable TCP, i.e. the sender halves the congestion window *cwnd* and reduces the slow start threshold *ssthresh*. Fast Retransmit/Recovery is still available for ECN capable stacks for responding to three duplicate acknowledgments.

2.2 Related Work

In [FLO93], Sally Floyd and Van Jacobson introduce RED for congestion avoidance in packet-switched networks. Their simulations show that the RED gateway has no bias against bursty traffic and avoids the global synchronization of many connections decreasing their window at the same time. During congestion, the probability that the router notifies a particular connection to reduce its window is roughly proportional to that connection's share of the bandwidth through the router.

[FLO94] proposes the new guidelines for TCP's response to ECN mechanism, and explores the benefits and drawbacks of ECN in TCP/IP network. By simulations, Floyd shows that one of advantages of ECN is that it can avoid unnecessary packet drops, which avoids unnecessary delay for packets from low-bandwidth delay-sensitive TCP congestions. [FLO98a] presents the implementation and validation of ECN in the famous network simulator: ns. [RAM99], [RAM01] and [FLO00c] further present their proposal on the guideline for ECN implementation in TCP/IP networks. [FLO97] discusses the rules-of-thumb values for the RED parameters.

Uvaiz Ahmed and Jamal H. Salim [AHM99] have further shown that ECN enhancements on active congestion management improve both bulk and transactional TCP traffic over Reno TCP with one router. Compared with RED, ECN is fairer. The improvement is more obvious in short transaction types of flows because of two factors: (1) Fewer retransmissions occur with ECN, which means that less traffic is in the network, (2) ECN avoids timeouts by getting faster notification, which implies less time is spent during error recovery. In the experiments, they used a few homogeneous flows with one congested router in their testbed.

[QIU99] addresses a phenomenon they observed in TCP/IP networks when the number of connections competing for the same bottleneck router becomes large, TCP's ability to share the bottleneck fairly and efficiently decreases. Their analysis of packet traces suggests that the degradation of TCP is substantially due to the total loss rate observed in the Internet. This happens when many competing flows cause a higher loss rate in a bottleneck router. [GER99] finds by the simulations that the very pronounced unfairness trend is typical of the behavior of many flows. ECN works properly when the ECN router queue can hold several packets per flow. However, ECN also becomes grossly unfair when the queue size is not large enough. Even some of the connections never get a chance to transmit, which is the so-called lockout phenomenon.

[FLO91] discusses the bias in TCP/IP networks against connections with multiple congested routers and the bias of the *TailDrop* and Random Drop routers against bursty traffic. Using simulations and a heuristic analysis, Floyd shows that in a network with *TailDrop* routers a longer connection with multiple congested routers can receive unacceptably low throughput, while in a network with no bias against connections with longer roundtrip times and with no bias against bursty traffic, a connection with multiple congested routers can receive an acceptable level of throughput. A longer connection is disproportionately likely to have packets dropped at the router. She also points out that although using different measures of fairness have quite different implications, there's still no generally-agreed-upon definition for fairness in a computer network.

Recently, a number of research efforts have focused on possible shortcomings of the algorithms in RED and have proposed modifications and alternatives, e.g., BLUE [FEN99b], and SRED [OTT99]. Feng *et al*. [FEN97][FEN99a] found that one of the inherent weaknesses of RED and other proposed active queue management schemes is that the effectiveness of RED depends, to a large extent, on the appropriate parameterization of the RED queue. For example, as the number of connections becomes large, the impact of individual congestion notification decreases. Without modifying the RED algorithm to be more aggressive, the RED queue degenerates into a simple DropTail queue. On the other hand, as the number of connections becomes small, the impact of individual congestion notifications increases. In this case, without modifying the RED algorithm to be less aggressive, under-utilization can occur as too many sources back off their sending rates in response to the congestion notification. By the simulation experiments with 8, 32 and 64 homogeneous flows respectively and by the deployment of various values of max_p, they showed that when the number of flows increases, RED doesn't deliver congestion notification to a sufficient number of sources. Thus, the RED queue continually overflows causing it to behave more like a drop-tail queue. To overcome these shortcomings of RED, [FEN97] and [FEN99a] presents that adjusting RED parameters properly could effectively reduce packet loss while maintaining high link utilization under a range of network load. In [FEN99b], there are set of results from simulations of RED with ECN enabled in both routers and end-host TCP implementation. Their simulations focused primarily on the effects of the parameter w_q used to smooth measurements of the average queue size. Some of these simulations use a large number of flows (1,000 – 4,000) that generate traffic with Pareto on/off periods. Unfortunately, they only simulated with homogeneous flows.

In [CHR00], Christiansen, et al evaluate RED across a range of parameter settings and offered loads. Their results show that RED has a minimal effect on HTTP response times for offered load up to 90% of link capacity, and response times at loads in this range are not substantially effected by RED parameters, while in heavily congested network with 90%-100% load, RED parameters that provide the best link utilization produce poorer response time. They also find that except for min_{th}, which should be set to larger values to accommodate the highly burst character of Web traffic, the guidelines [FLO97] for RED parameter settings and for configuring interface buffer sizes (FIFO and RED) also hold for the Web-like traffic used in their experiments. This paper concludes that attempting to tune RED parameters outside these guidelines is unlikely to yield significant benefits. The authors didn't investigate the performance of RED with ECN-enabled in their experiments, and they didn't consider fairness at all.

For the performance of networks, delivering congestion signal in time is critical. [LIU01] presents that by providing the mark-front strategy for ECN to send even faster congestion signals, mark-front strategy reduces the buffer size requirement at the routers, and it also avoids packet losses and thus improves the link efficiency when the buffer size in an ECN router is limited. With simulations, [LIU01] show that mark-front strategy improves the fairness among old and new users, and alleviates TCP's discrimination against connections with large RTT.

In summary, since the introduction of RED, many researchers have done investigations about the behaviors and performances of RED and ECN, and proposed a variety of enhancements and changes to router management to improve congestion control. But these previous researches usually considered only the specific traffic domain space in the network with a small or medium number of homogeneous TCP flows. The behavior of RED and ECN congestion control mechanisms in TCP/IP network with many competing heterogeneous flows, hasn't been sufficiently explored. This is the main motivation for this thesis.

Chapter 3 Experimental Methodology

As mentioned in the previous chapters, the goals of this thesis include (1) investigating the behavior and performance of RED with ECN congestion control mechanisms with many heterogeneous TCP Reno flows; (2) exploring an adaptive version of ECN (AECN), which is based on the current standard ECN algorithm and more adaptive to heterogeneous flows and congestion conditions in such a way as to signal TCP hosts to adjust their congestion control mechanisms in time. To reach the goals, two main experimental steps are taken in the study: (1) running experiments to gather the experimental data on the key performance indicators to evaluate the behavior of ECN and RED with heterogeneous, (2) based on the results from Step 1, the basic algorithm of AECN is proposed, and then more experiments are run to compare the performance of AECN and ECN. This chapter describes our methodology for obtaining each metric data, the choice of network simulation tools and the approaches for data analysis.

3.1 The Selection of Measurement Criteria

To fully evaluate the behavior and performance of RED and ECN, and AECN, four key performance indicators used in our study are throughput, goodput, fairness and delay. Based on the observation, more efforts are put on the most important ones among these four indicators.

3.1.1 Throughput

Throughput is defined as the data rate at which a source can send packets (including retransmitted packets) to the sink. Assume a source sends out 10 packets in a specific time, and one of these ten packets are dropped by a router in the course of transmission, then the resulting throughput is 9*packet/time taken to transmit. Due to including retransmitted packets, throughput is normally a little higher than goodput.

3.1.2 Goodput

Goodput is defined as the effective data rate as observed at the user. For example, assume 10 data packets are transmitted from a source to a sink, and two of these ten packets are retransmitted packets, then the efficiency is 80%, and the resulting goodput is 8*packet size/time taken to transmit.

3.1.3 Fairness

Fairness has been defined in a number of different ways. Currently, there's still no generally-agreed-upon definition for fairness in a computer network. However, different measures of fairness have quite different implications [FLO91]. Two popular fairness measurement methods are used in our experiments: Jain's fairness index [JAI91] and max-min fairness [PET00].

1. Jain's Fairness Index

   Jain's fairness index postulates that the networks is a multi-user system, and derives the metric to see how fairly each user is treated. It's a function of the variability of throughput across each user. For a set of user throughput (x_1, x_2,..., x_n), Jain's fairness index to the set is defined as follows:

   $$f(x_1, x_2, \ldots, x_n) = (\Sigma_{i=1}^{n} x_i)^2 / (n * \Sigma_{i=1}^{n} x_i^2).$$

   The fairness index always lies between 0 and 1. A value of 1 indicates that all flows got exactly the same throughput.

2. Max-min Fairness

   Max-min fairness is another common fairness definition, which is also called bottleneck optimality criterion. A feasible flow rate x is defined to be max-min fair if any rate x_i cannot be increased without decreasing some x_j which is smaller than or equal to x_i. To satisfy the min-max fairness criteria, the smallest throughput rate must be as large as possible. Given this condition, the next-smallest throughput rate must be as large as possible, and so on [FLO94]. Many researchers have developed algorithms achieving max-min fairness rates. Computing the max-min fair vector requires global information, including information from networks and hosts. In our simulation result analyses, graphs are used to visually analyze the max-min fairness with respect to goodput for heterogeneous flows.

3.1.4 Delay

Delay is another important performance indicator used in reporting our simulation results. In this study, delay refers to one-way delay, from a source to a sink, and includes link delay, propagation delay and queue delay.

3.2 Experimental Tools

3.2.1 Network Simulator *ns-2*

The flexibility of exploring various simulation scenarios and unrestricted data access are the primary reasons for us to choose ***ns-2*** [NS201].

The network simulator, ***ns-2***, is widely adopted in the network research community. ***ns-2*** evolved as a part of the VINT (Virtual InterNetwork Testbed) project, a collaborative project among University of Southern California, Xerox PARC, Lawrence Berkeley National Laboratory, and the University of California, Berkeley. ***ns-2*** is a discrete event simulator which provides the following supports:

- Various network protocols (transport, multicast, routing);
- Simple or complex topologies (including topology generation);
- Agents, defined as endpoints where network-layer packets are constructed or consumed;
- Various traffic generators
- Simulated applications (FTP, Telnet, Web)
- Most queue management algorithms (TailDrop, RED, ECN) and packet scheduling schemes
- LAN/WAN, and wireless networks.

The code of the simulator is written in C++ and OTcl. There is one-to-one correspondence between a class in C++ (compiled hierarchy of classes in ***ns-2***) and a class in OTcl (interpreted hierarchy). This software architecture (also called split programming model) enables high-performance simulation of packet level routines (implemented in C++) and flexible configuration and control of the simulation using an interpreted language such as OTcl [BAJ99]. The following subsections explain the ***ns-2*** specific details for our simulations.

3.2.1.1 Simulation Input Scripts

All the necessary information to configure and control a simulation run in ***ns-2*** is written in the form of an input OTcl script. The simulation objects (nodes, links and traffic sources) are instantiated with the script, and immediately mirrored in the compiled hierarchy. The input script defines the topology, builds the agents (sources and destinations), sets the trace files and sets the start time for the initial events in the simulation. The initial events might later generate new events. For example, the user can only specify the start of an FTP session and the number of packets to be transferred. This is an initial event indicating the start of the FTP session. When the FTP transfer starts, new events will be generated such as a packet arrival at a router queue, a check of ECT bit of the packet, en-queue and de-queue, an arrival at a receiver, a generation of an ACK packet, etc. The simulator always executes events in the order specified in the event list, which is always sorted by time.

3.2.1.2 Simulation Output Traces

Tracing in ***ns-2*** can be performed by using *trace* or *monitor* objects. Trace objects collect the data for each packet generation, arrival, departure, drop or mark. Monitor objects collect data on an aggregate level and are implemented as counters of specific parameters of interest (total number of packet or byte arrivals, departures or drops). Monitor objects are useful when basic information about the dynamics of the simulation is needed. However, in order to have a comprehensive understanding of each metric pattern, this study performs tracing on a per packet basis. The aggregate information that can be obtained by monitor objects is insufficient to support a detailed study of the observed stochastic process or a process such as packet marking.

An output trace in ***ns-2*** has a fixed format, as shown in Table 3.1.

event	time	from node	To node	pkt type	pkt size	flags	fid	Src Addr	dst addr	seq num	pkt id

Table 3.1 ***ns-2*** output trace format

Each trace line starts with an *event* (+, -, d, r) (See Figure 3.1) descriptor followed by the simulation *time* (in seconds) of that event, and *from* and *to node*, which identify the link on which the event occurred. The next information in that line before flags is packet type, which can be TCP data packet or ACK packet, and the size of that packet (TCP data packet is 1000 bytes, and ACK packet is 40 bytes as shown in Figure 3.1). Currently, ***ns-2*** records ECN events (including C— ECN Echo bit set as 1, E— CE bit set as 1, A— ECR bit set as 1) in the flag field. Next to the flag field is flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script. The fields src addr and dst addr contain source address and destination address in the form of *“node.port*”. The seq num field contains the network layer protocol’s packet sequence number. The last field is the unique id of the packet.

```
r    9.098704    60 61   tcp   1000   ------N   54   54.0    61.54   330   21908
+    9.098704    61 60   ack   40     C------   54   61.54   54.0    330   21983
-    9.098704    61 60   ack   40     C------   54   61.54   54.0    330   21983
r    9.098936    61 60   ack   40     -------   43   61.43   43.0    420   21975
+    9.098936    60 43   ack   40     -------   43   61.43   43.0    420   21975
-    9.098936    60 43   ack   40     -------   43   61.43   43.0    420   21975
r    9.099168    60 51   ack   40     -------   51   61.51   51.0    337   21964
+    9.099168    51 60   tcp   1000   ------N   51   51.0    61.51   339   21984
-    9.099168    51 60   tcp   1000   ------N   51   51.0    61.51   339   21984
-    9.099304    60 61   tcp   1000   ---AE-N   42   42.0    61.42   477   21928
r    9.099304    13 60   tcp   1000   ------N   13   13.0    61.13   36    21639
+    9.099304    60 61   tcp   1000   ------N   13   13.0    61.13   36    21639
d    9.099304    60 61   tcp   1000   ------N   13   13.0    61.13   36    21639
r    9.099368    56 60   tcp   1000   ---A--N   56   56.0    61.56   42    21973
---------------------------------------------------- NOTE -----------------------------------------------------------------
r : receive     (at to_node);    + : enqueue    (at queue);    - : dequeue    (at queue)
d : drop        (at queue)
------------------------------------------------------------------------------------------------------------------- ----
```

Figure 3.1 A sample of *ns-2* output trace

3.2.2 Data Extraction Tools

Having simulation trace data ready, the data of interest for computation of each metric needed to be extracted. A data extraction tool **C_Stat** was developed with *perl* for generating the report on throughput, goodput, Jain's fairness index and delay, and other statistical data, such as the distribution of marked or dropped packets for each flow. Figure 3.2 summarizes the simulation processes for each simulation experiment with these experimental tools.

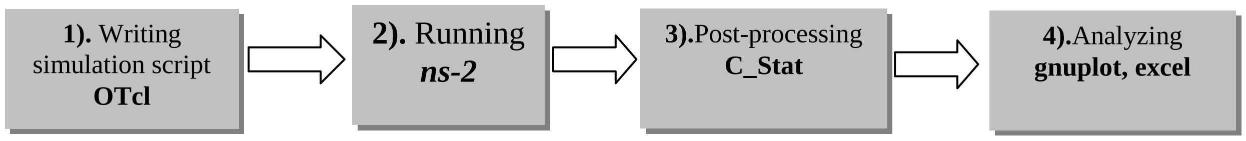
1). Writing simulation script OTcl
2). Running ns-2
3).Post-processing C_Stat
4).Analyzing gnuplot, excel

Figure 3.2 the Simulation Processes with *ns-2*

3.3 Experimental Setup and Validations

In the study, different simulation scenarios are setup. More details about the scenarios are presented in Chapter 4 and 5. The ***ns-2*** source code was modified for the implementations of ECNM[1] and AECN. In order to ensure these implementations are correct, validations were taken during the experiments to guarantee that the implementations were conformant to our design specification by visually inspecting the corresponding ***ns-2*** traces.

Each simulation experiment was run for 100 seconds, but data collected during the first 20 seconds was discarded to reduce startup and stabilization effects. These effects are illustrated in Figure 3.3 which shows the router queue distribution, and Figure 3.4 which shows a plot of mean aggregate throughput for all flows during each one second interval in a typical experiment. In each simulation with a specific number of flows, half of these flows start at second 0, and the rest start at second 2.

[1] ECNM, an extension to the standard ECN, is investigated to compare with the standard ECN. The only difference between ECNM and ECN is when the average queue size is above the maximum threshold of ECN router queue, the standard ECN will drop a incoming packet while ECNM will continue to mark the incoming packets.

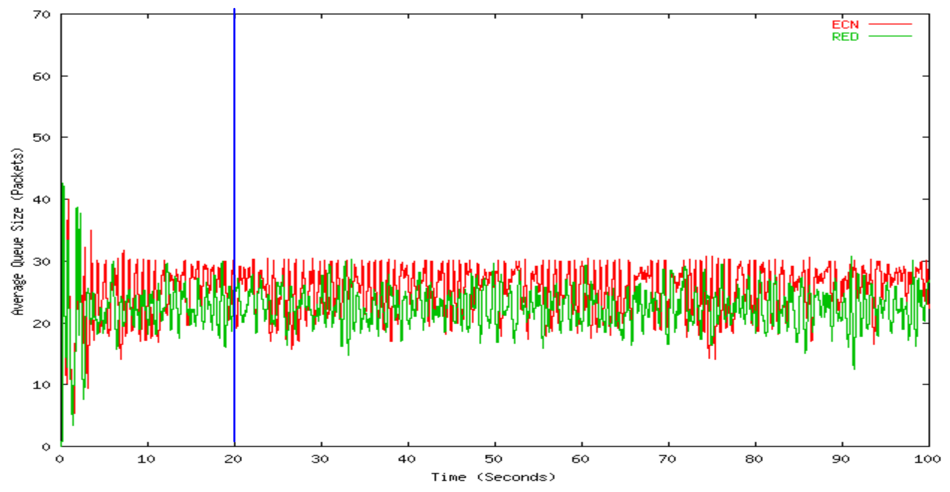

Figure 3.3 ECN router queue distribution for 60 flows,

max_p=0.1, min_th=10 packets, max_th=30 packets, cwnd=64 packets

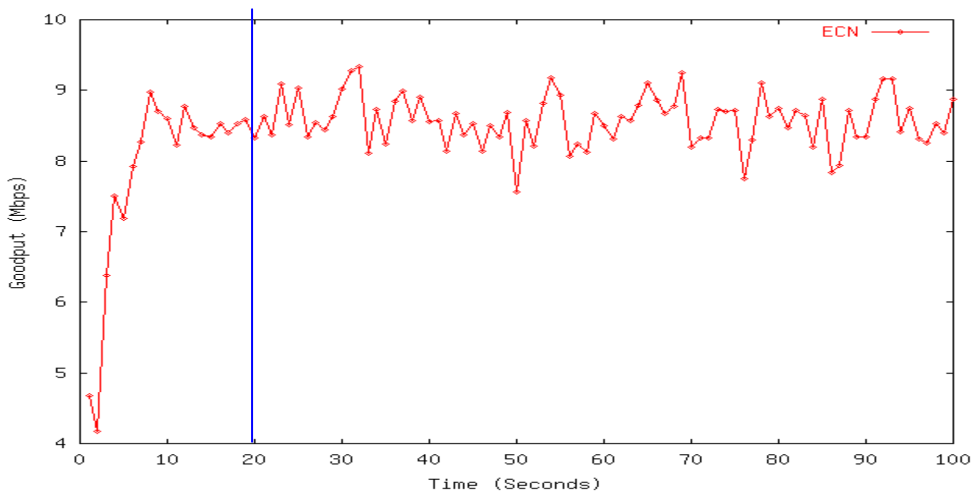

Figure 3.4. Mean aggregate goodput forECN with 60 flows

during each one second interval

max_p=0.1, min_th=10 packets, max_th=30 packets, cwnd=64 packets

Chapter 4

Evaluation of ECN with Heterogeneous TCP Flows

4.1 Introduction

This chapter focused on presenting the simulation results on the performance and behavior of RED routers and ECN routers with heterogeneous TCP flows. Several research studies have reported better performance for Explicit Congestion Notification (ECN) when compared against RED. These results add support to the Internet draft "Addition of ECN to IP" [RAM99]. However, most of these studies cover only a limited portion of the traffic domain space. Specifically, little attention has been given to evaluating the effects of a large number of heterogeneous flows. Although a couple of these studies consider fairness among competing homogeneous flows, ECN behavior with heterogeneous flows has not been thoroughly studied.

Therefore, as mentioned in previous chapters, one of the two goals of this thesis is to add to the existing information on ECN behavior specifically with regard to the impact of the number of flows, the effect of ECN tuning parameters on performance, and the effectiveness of ECN's congestion warnings when many flows cause the congestion. This chapter presents the simulation results of the evaluation of ECN performance with many heterogeneous flows.

Section 4.2 briefly defines a few measurement terms and reviews previous ECN studies to provide context for our experiments. Section 4.3 discusses experimental scenarios and details in this step. The next section analyzes the simulated results and the final section includes concluding remarks.

4.2 Definitions

4.2.1 Robust, Average and Fragile flows

Fragile TCP flows are defined as those from sources with either large round-trip time or small send window sizes and robust flows as having either short round-trip time or large send windows [LIN97]. This delineation emphasizes a flow's ability to react to indications of both increased and decreased congestion at the bottleneck router. To evaluate the behavior of ECN with heterogeneous flows, our experiments simulate three distinct flow groups (fragile, average, and robust flows). These flows differ only in their end-to-end round-trip times (RTTs). The maximum sender window is held fixed at 64 packets in all simulations to simplify the analysis.

4.2.2 ECNM

In this study, one variant of ECN, called ECNM (ECN with Marking) is also investigated to compare its behavior with standard ECN. ECNM differs from standard ECN in that ECNM marks packets when the average queue size exceeds max_{th} and drops packets only when the router queue overflows.

4.3 Simulation Scenarios

The simulation network topology (See Figure 4.1) consists of one router, one sink and a number of sources. Each source has a FTP connection feeding 1000 byte-packets into a single congested bottleneck link whose bandwidth is 10 Mbps with 5 ms delay time to the receiver. The one-way link delays to the router for the fragile (F_1, …, F_i), average (A_1, …, A_i) and robust (R_1, …, R_i) sources are 145 ms, 45 ms and 5 ms respectively. Thus, without considering the router queue delay, the fragile, average and robust flows have minimum round-trip time of 300 ms, 100 ms, and 20 ms. The bottleneck router has a physical queue size of 120 packets. Max_{th} is always three times than min_{th} in the simulations. Except for the maximum send window size of 64 packets, all other parameters use the ***n2-s*** default values.

A number of ***ns-2*** experiments were run such that the cumulative traffic flow into the heavily congestion router remains fixed at 300 Mbps even though the number of flows is varied across simulations. In all cases, the number of flows is equally divided among the three flow groups. Thus, 15 flows in the following graphs of this chapter implies 5 fragile, 5 average and 5 robust flows, and each flow with a 20 Mbps data rate whereas a figure point for 120 flows implies a simulation with 40 fragile, 40 average and 40 robust flows each with a 2.5 Mbps data rate. Simulations were run with the total number of flows set at 15, 30, 60, 240, 480, and 600 flows respectively.

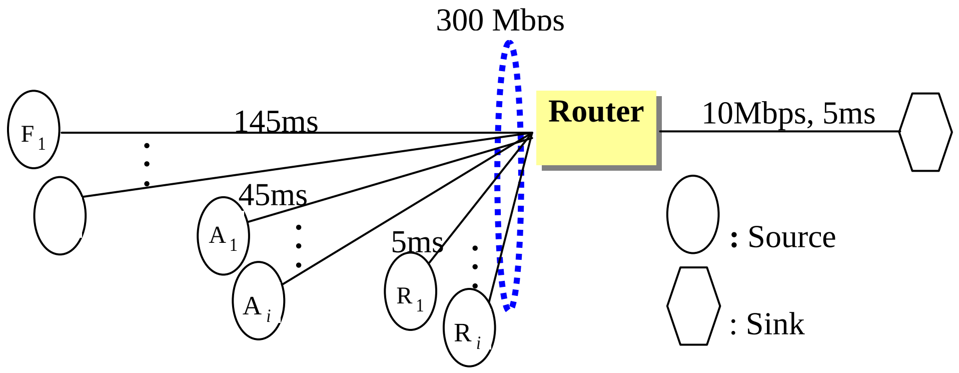

Figure 4.1 Simulation Topology

4.4 Simulation Result Analyses

Aggregate throughput (or goodput) in the following graphs is the sum of the throughput (or goodput) of all fragile, average and robust flows. Much of result analysis in this section appears in the paper [KNI01].

4.4.1 Throughput

Figure 4.2 shows the aggregate throughput distribution with the number of flows. As shown in this graph, when the number of flows is small, e.g. below 120 flows, ECN beats RED on throughput. But when the number of flows increases and the congestion becomes heavier, ECN may even lose to RED on throughput. In this figure, when the number of flows is higher than 120 flows, increasing max_p doesn't help ECN instead, which actually has lower aggregate throughput than RED. For example, when max_p is equal to 0.5 or 0.8, and the number of flows is 240, ECN has smaller throughput than RED. The reasons for that include: (1) ECN marks incoming packets when the average queue size is between max_{th} and min_{th}. This implies ECN normally has a higher current queue size and queue delay than RED; (2) When the number of flows is high and the congestion is heavy, many more packets will be marked by ECN router, which will cause the senders to slow down frequently. As show in Figures 4.3 and 4.4, ECN does have higher current queue length than RED when the number of flows is 240. And the router has more chance to be empty than RED, which would finally degrade the throughput of ECN. Meanwhile, when the number of flows is small, such as 15 flows or 30 flows, Figure 4.2 shows that max_p should be conservative enough to get higher throughput for both mechanisms. Figures 4.5 and 4.6 present the aggregate throughput distribution with different max_p when the number of flows is 30 and 120 respectively. These two figures show that there's an optimal max_p for ECN to get the best throughput for each different number of flows. For example, for ECN with 30 flows, $max_p = 0.1$ is the best to get the highest throughput when min_{th} and max_{th} both are equal to 10 and 30. Once the number of flows increases, max_p should also increase to make ECN more aggressive to notify enough sources to slow down frequently enough. Meanwhile, as shown in Figure 4.5 and 4.6, different values of min_{th} and max_{th} can also influence the throughput of ECN. When the number of flows increases, max_{th} should be large enough as to provide enough space at router queue for the incoming packets to get higher throughput.

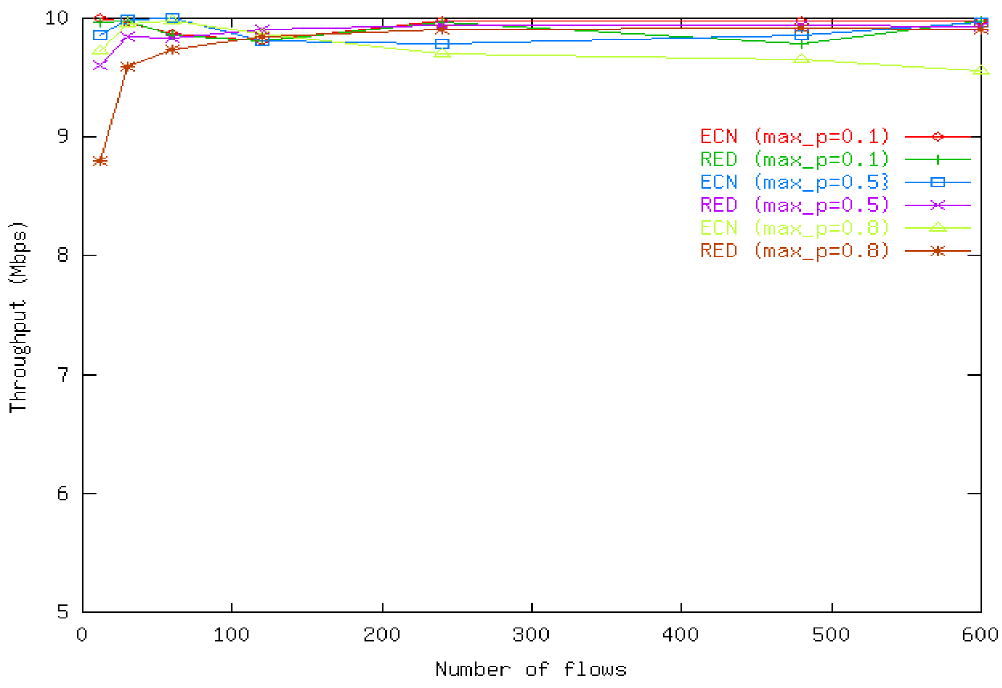

Figure 4.2 Aggregate Throughput Distribution with the Number of flows.

min_{th} = 10 packets, max_{th} = 30 packets,

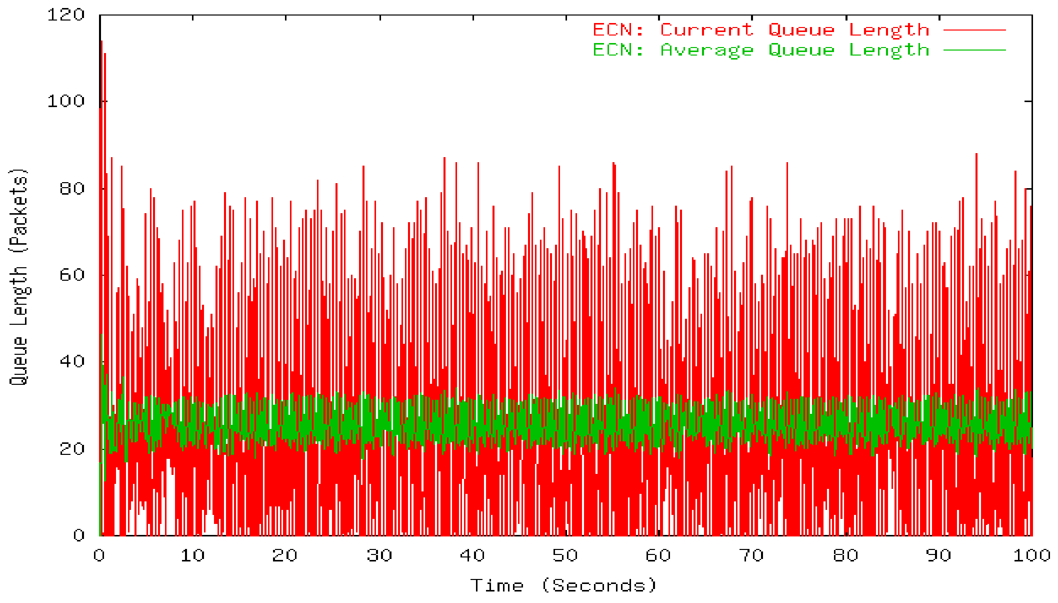

Figure 4.3 ECN: Queue Length Distribution with Time.

min_{th} = 10 packets, max_{th} = 30 packets, max_p =0.5, Number of flows=240.

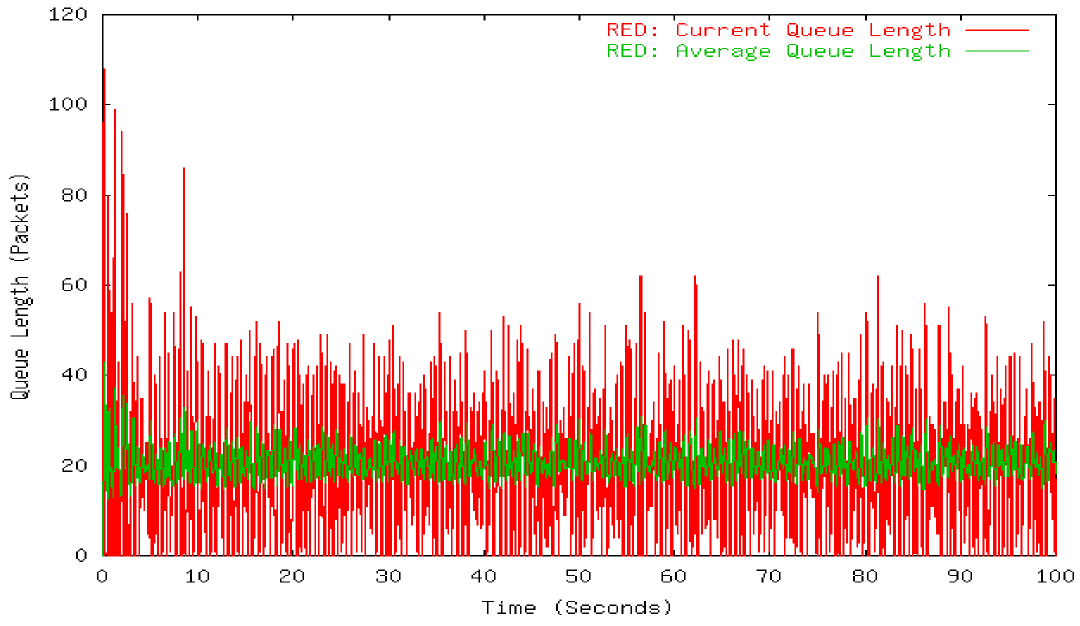

Figure 4.4 RED: Queue Length Distribution with Time.

min_{th} = 10 packets, max_{th} = 30 packets, max_p =0.5, Number of flows=240.

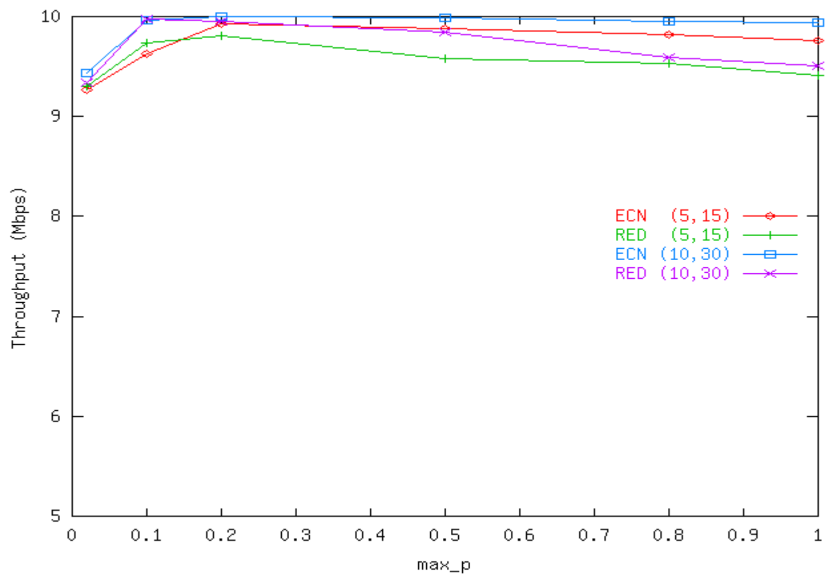

Figure 4.5 Aggregate Throughput Distribution with max $_p$, Number of flows = 30.

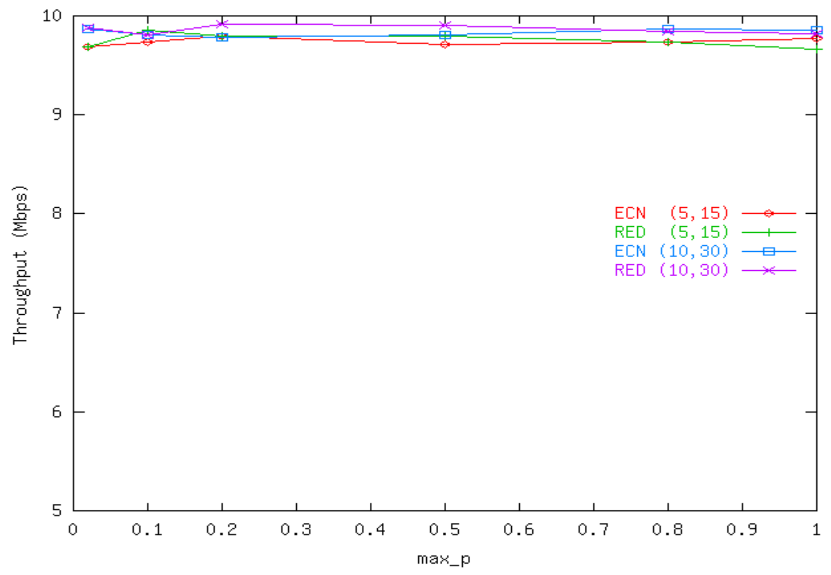

Figure 4.6 Aggregate Throughput Distribution with max $_p$, Number of flows = 120.

4.4.2 Goodput

Figure 4.7 gives ECN and RED aggregate goodput with the number of flows varying from 15 to 600. ECN with higher *max* $_p$ provides better goodput than RED in all cases except 15 flows. When *max* $_p$ is equal to 0.1, ECN and RED both have large drops in goodput beginning at 60 flows.

ECN with higher min_{th} and max_{th} provides better goodput even when the number of flows is large. The main reason why ECN has better goodput in this case is due to the fact that ECN always marks the incoming packet when average queue size is between min_{th} and max_{th}, which causes less packet drops and retransmissions than RED. For the case with 15 flows, too high a value for max_p also causes lower goodput for ECN compared with a lower value of $max_p = 0.1$. This indicates when the number of flows is small and congestion is light, max_p should not be too aggressive. On the other hand, while the number of flows is large and congestion is heavy, max_p should increase to make ECN more aggressive so that enough sources are informed with an enough frequency to back off during heavy congestion and to get fewer packets dropped and further improve the aggregate goodput.

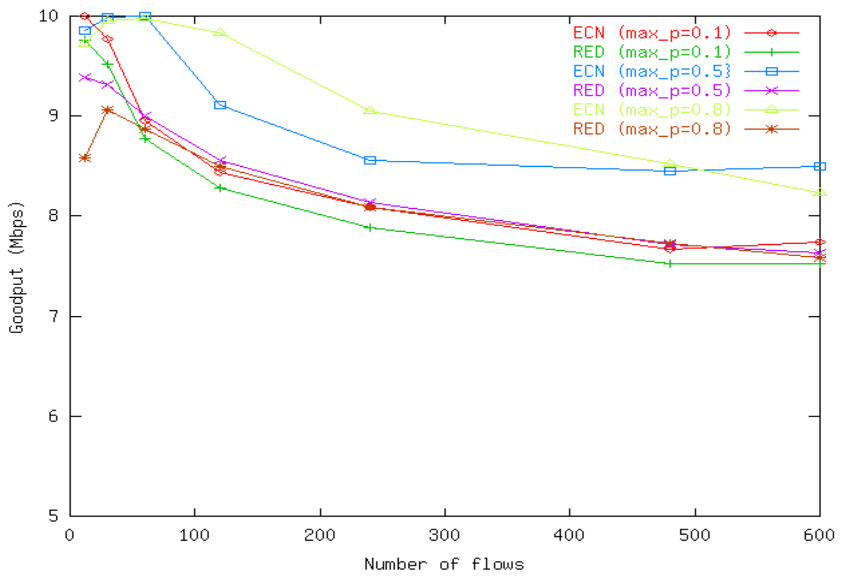

Figure 4.7 Aggregate Goodput Distribution with the Number of flows.

min_{th} = 10 packets, max_{th} = 30 packets

Figures 4.8 and 4.9 track the effect on aggregate goodput distribution by varying max_p, min_{th} and max_{th} in simulations with 30 and 120 flows respectively. Figure 4.8 shows that the values of min_{th} and max_{th} have an obvious effect on the aggregate goodput between RED and ECN. ECN gets a clear advantage over RED on goodput. But once *max* $_p$ is above *0.2*, the goodput doesn't change much ECN or RED with 30 flows. In Figure 4.9 where 120 flows provide the same flow demand as 30 flows in Figure 4.8, ECN with *max* $_p$ *=0.8*, *min* $_{th}$ =10, and *max* $_{th}$ =30 yields the highest aggregate goodput and there's no *max* $_p$ setting for RED that works well.

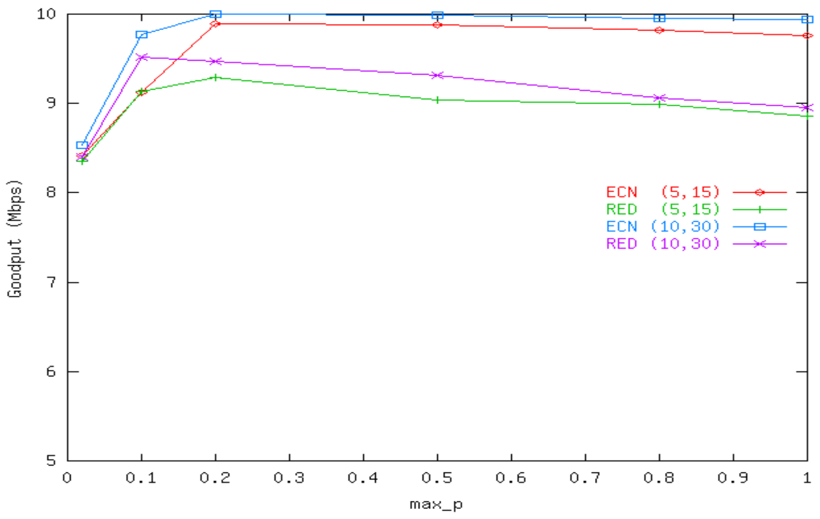

Figure 4.8 Aggregate Goodput Distribution with max $_p$, Number of flows = 30.

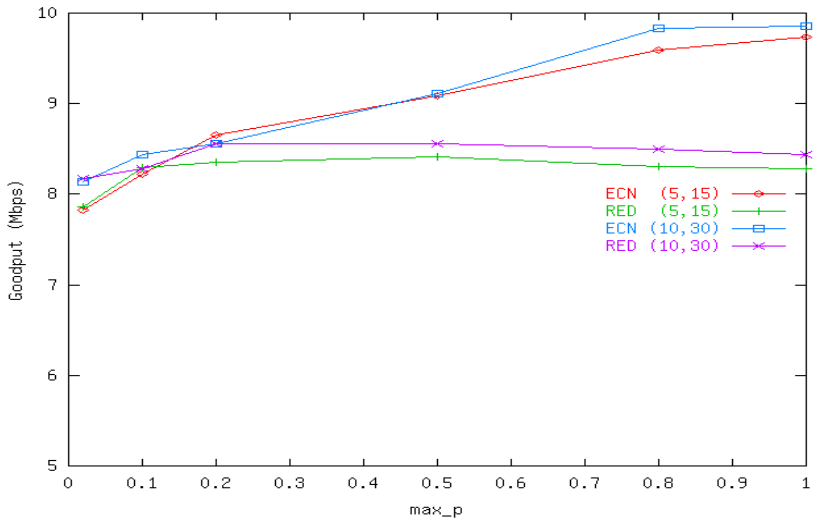

Figure 4.9 Aggregate Goodput Distribution with max $_p$, Number of flows = 120.

4.4.3 Delay

Figure 4.10 describes the one-way delay distribution with the number of flows among these three flow groups. As shown in this figure, the robust flows have a clear advantage over the fragile and average flows on delay. Furthermore, the one-way delay for each flow group increases a little with the number of flows increasing since more packets enter the router queue, the average queue size and queue delay increases as well. Meanwhile, ECN has a little higher one-way delay than RED. That's mainly due to the fact that ECN router normally has higher average queue size than RED router (See Figure 4.11 and 4.12), which means ECN has a little higher queue delay. The ECN goodput improvement is offset by a small increase in the one-way delay for ECN. However, even though ECN doesn't win RED on delay, ECN has an absolute advantage than RED on goodput in most cases. As shown in Figure 4.10, ECN can get around 1% higher than RED on the mean one-way delay of all flows while Figure 4.7 shows that ECN can get about 10% higher than RED on goodput.

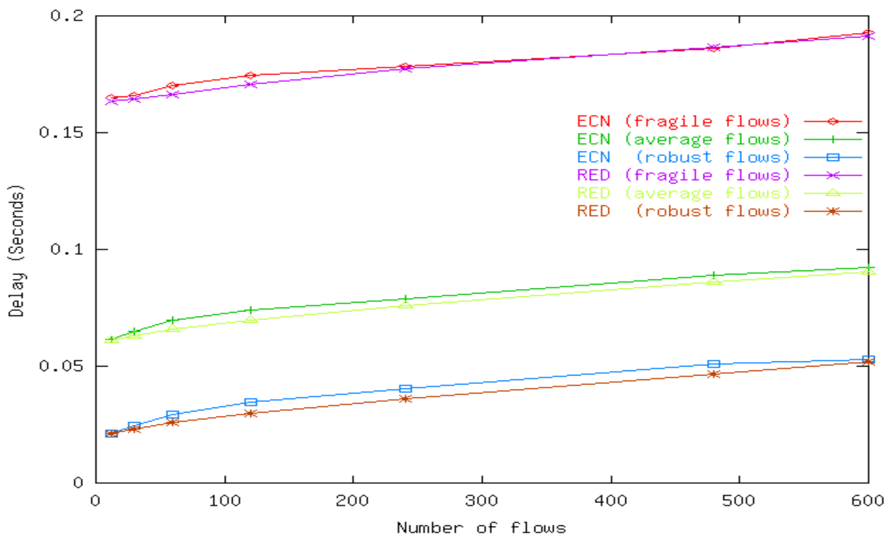

Figure 4.10 Delay Distribution among Each Flow group with the Number of flows.

min_{th} = 10 packets , max_{th} = 30 packets

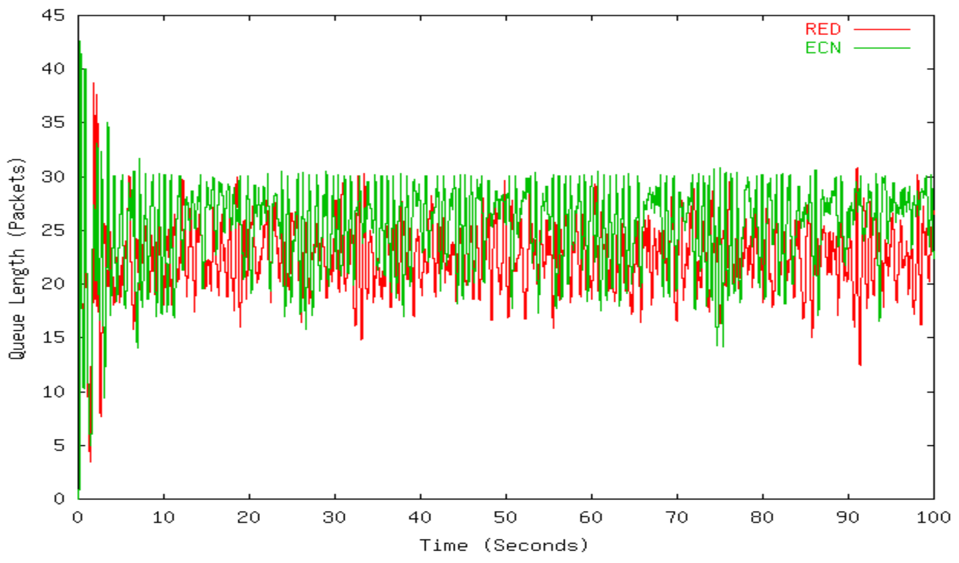

Figure 4.11 Average Queue Size Distribution (30 flows).

min_{th} = 10 packets, max_{th} = 30 packets, $max_p = 0.1$

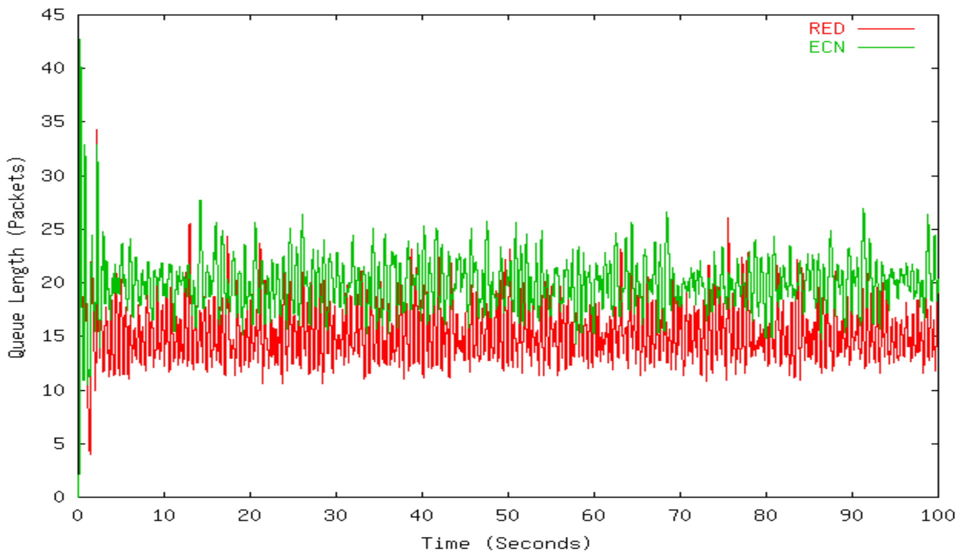

Figure 4.12 Average Queue Size Distribution (60 flows).

min_{th} = 10 packets, max_{th} = 30 packets, $max_p = 0.5$

4.4.4 Fairness

In this section, two fairness methods, Jain's Fairness Index and Visual Max-min Fairness, are used to measure the fairness metric of ECN and RED.

4.4.4.1 Jain's Fairness Index

Figure 4.13 employs Jain's fairness to quantify ECN and RED behavior. ECN is fairer than RED in almost all situations. Since perfect fairness has a Jain's fairness index of 1, it's clear that as the number of flows goes above 120 none of the choices prevent unfairness. The fact, that ECN with max_p =0.1 is fairest at 30 flows while max_p =0.5 is the fairest at 60 flows and max_p =0.8 at 120 flows, implies the marking probability should be dynamically adjusted based on a flow count estimator. The unfairness at a high number of flows can also be partially attributed to a lockout phenomenon, where some flows are unable to get any data through the congested router for the duration of the simulation. Locked out flows begin to appear for both ECN and RED above 120 flows (See Figure 4.14 and 4.15).

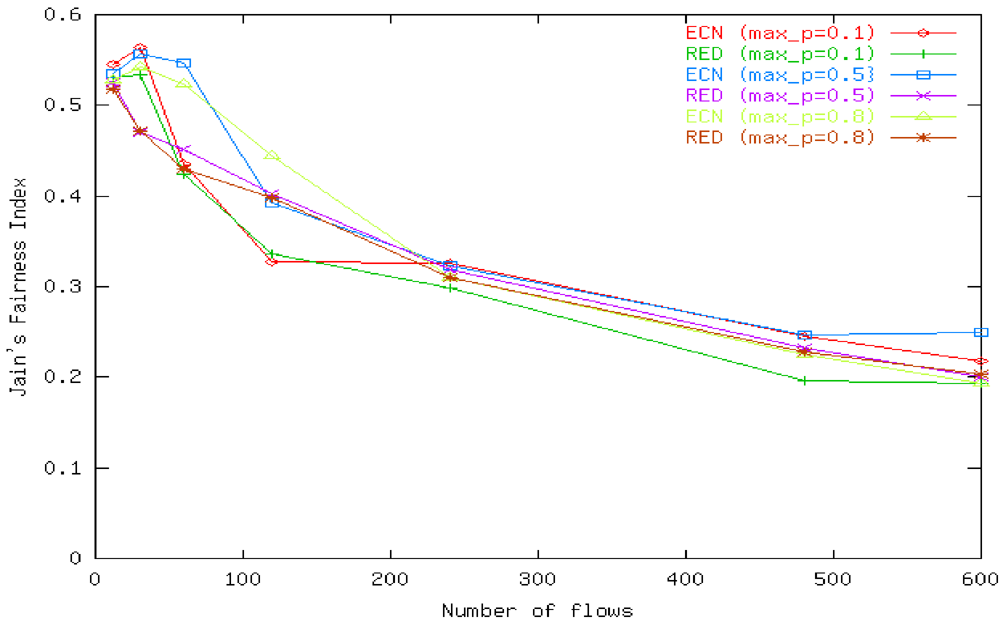

Figure 4.13 Aggregate Jain's Fairness Index with the Number of flows. min_{th} = 10 packets, max_{th} = 30 packets.

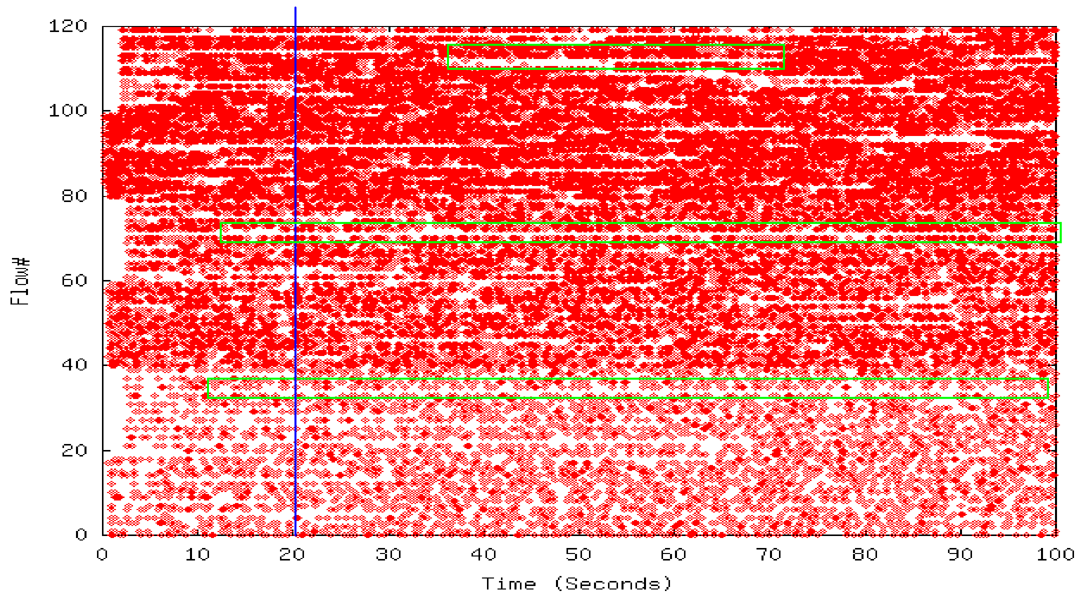

Figure 4.14 ECN Marked packet Statistics, 120 flows, *max* $_p$ =0.8

NOTE*: (1). Lockout regions*: 🟩

(2). Flow No. 0 ~ 39 refers to fragile flows, flow No. 40 ~ 79 for average flows, and flow No. 80 ~ 119 for robust flows.

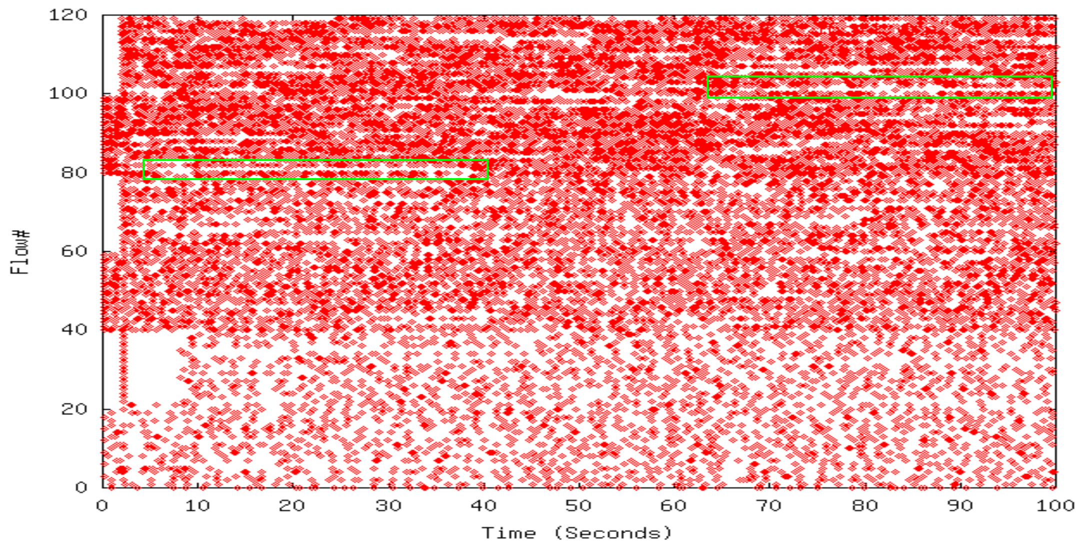

Figure 4.15 RED Dropped packet Statistics, 120 flows, *max* $_p$ =0.8

4.4.4.2 Visual Max-min Fairness

Figure 4.16 through 4.19 provide a visual sense of max-min fairness via the gap between the averaged goodputs for the three flow groups. In all these graphs, ECN provides better overall goodput than RED, but the difference is most pronounced in Figure 4.16 where the traffic is generated by 60 flows. Figure 4.16 and 4.17 differ only in an increase of max_p from 0.2 to 0.5. The more aggressive ECN marking in Figure 4.17 provides better goodput for robust flows than RED. However, this change doesn't reduce the goodput gap between robust and fragile flows. Figure 4.18 keeps max_p =0.5 but simulates 60 flows.

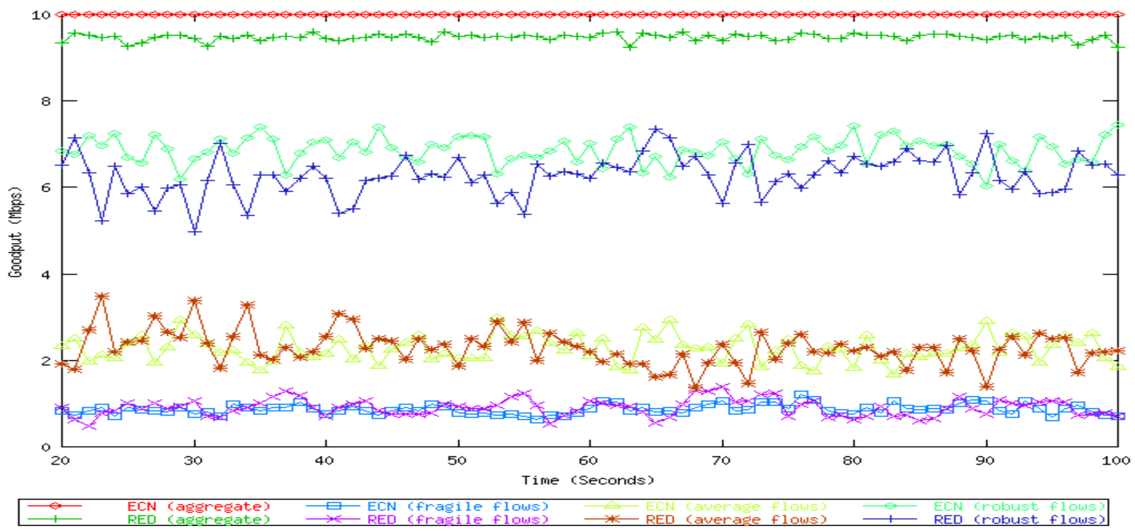

Figure 4.16 Goodput Distribution among Each Flow Group with Time.

Number of flows = 30, max_p = *0.2*, min_{th} = 10 packets, max_{th} = 30 packets.

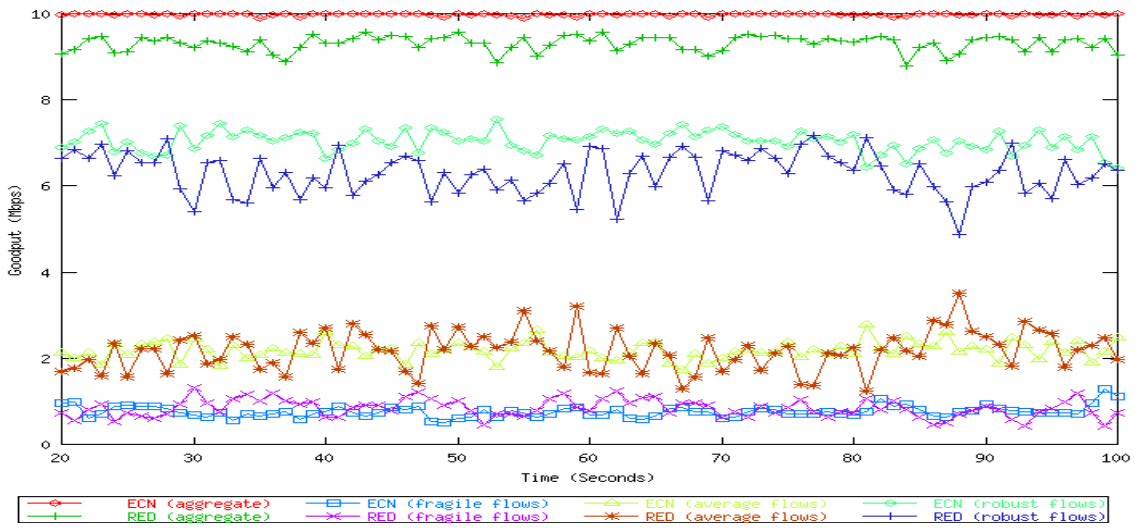

Figure 4.17 Goodput Distribution among Each Flow Group with Time.

Number of flows = 30, max_p = *0.5*, min_{th} = 10 packets, max_{th} = 30 packets.

Although overall goodput remains relatively unchanged for ECN in Figure 4.18, the goodput for the robust flows goes down while the goodput of the average and fragile flows increase slightly. This implies that varying max_p when there are heterogeneous flows can provide improvement in the visual max-min goodput. RED goodput is adversely affected by more flows. This suggests an adaptive ECN that uses different values of max_p for the different flow groups.

The significance of using goodput instead of throughput as a performance metric can be clearly seen in Figure 4.18 and 4.19. Because goodput excludes retransmissions, RED has about 12% lower goodput than ECN in Figure 4.16. Since RED drops and ECN marks, the RED drops trigger more TCP retransmissions. This effect is completely hidden in Figure 4.19 where aggregate RED throughput is only slightly lower than aggregate ECN throughput.

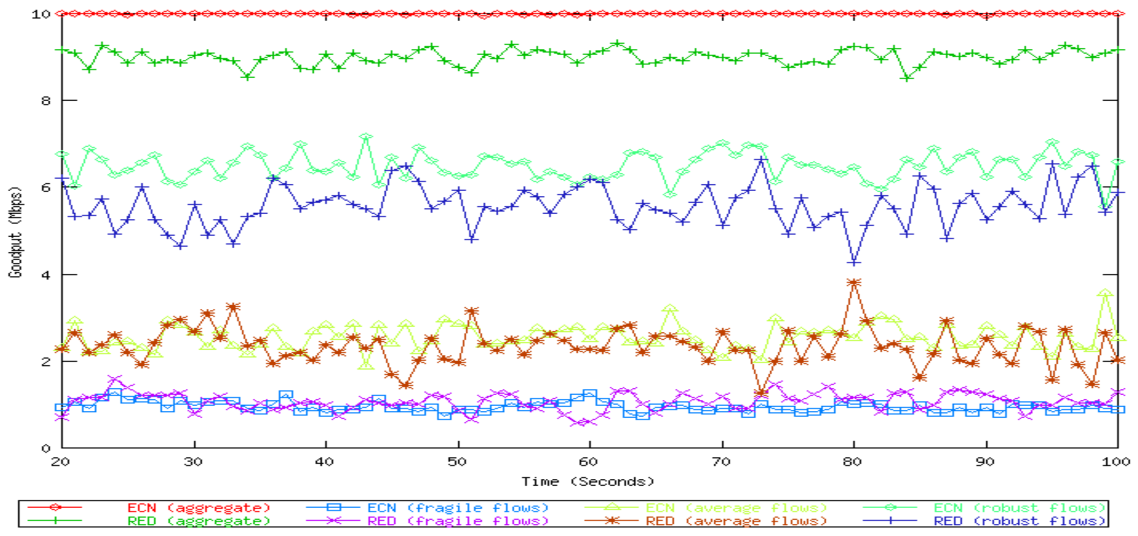

Figure 4.18 Goodput Distribution among Each Flow Group with Time.

Number of flows = 60, $max_p = 0.5$, min_{th} = 10 packets, max_{th} = 30 packets.

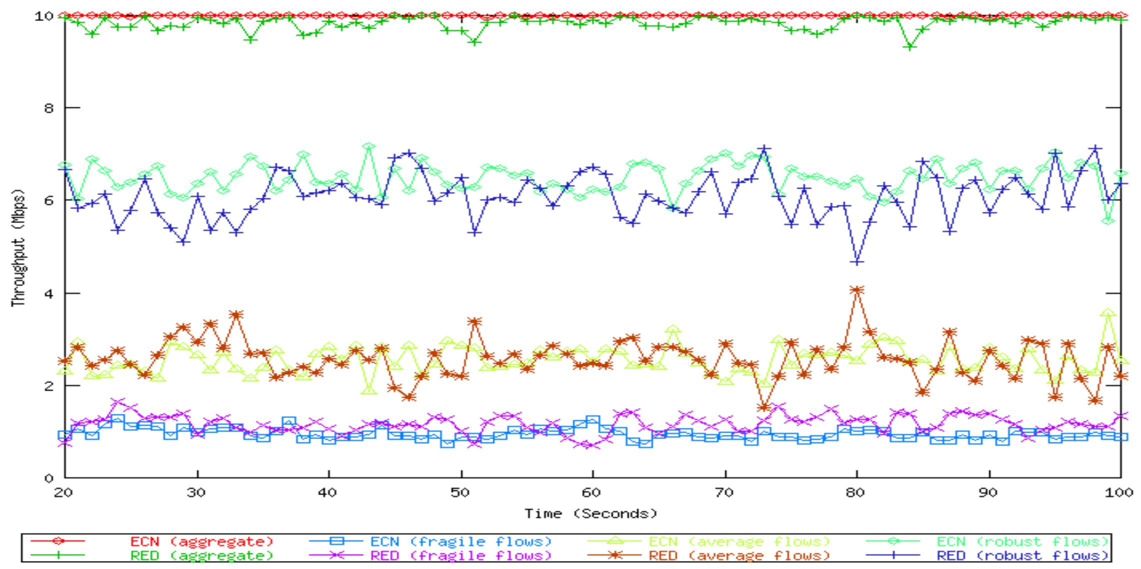

Figure 4.19 Throughput Distribution among Each Flow Group with Time.

Number of flows = 60, $max_p = 0.5$, $\max_{th} = 30$ packets, $min_{th} = 10$ packets

4.4.5 ECNM

Figure 4.20 compares standard ECN with ECNM. Recall ECNM differs from standard ECN in that ECNM marks packets when the average queue size exceeds max_{th} and drops packets only when the router queue overflows. This graph shows that ECN provides better goodput except at small values of max_p than ECNM.

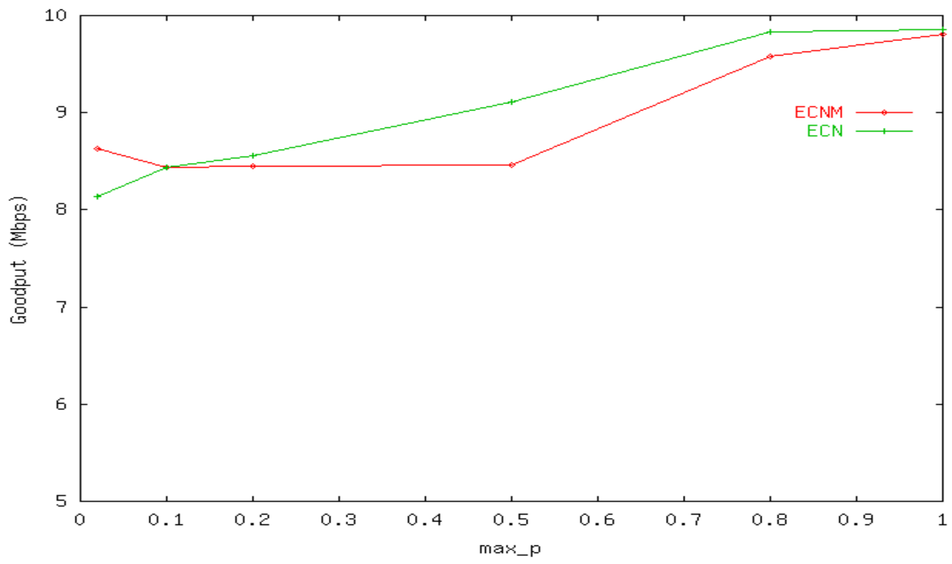

Figure 4.20 Goodput Distribution with $\max_p$,

Number of flows = 120, min_{th} = 10 packets, max_{th} = 30 packets

4.5 Conclusions

This chapter presents a series of ***ns-2*** simulations that evaluate the behavior and performance of ECN by comparing it with RED with heterogeneous flows. Generally ECN provides better goodput and is fairer than RED. However, in some case RED may have better throughput than ECN, especially when the number of flows and max_p are high. The results also show that the performance of both mechanisms be affected by the number of competing flows. However, ECN with an aggressive max_p setting provides significantly higher goodput than RED when there are a large number of heterogeneous flows. ECN also has a higher Jain's fairness Index and visual max-min fairness in the range of flows just below where flow lockout phenomena occur.

In the simulations studied, neither RED nor ECN mechanism is fairer to fragile and average flows. These results suggest that if congestion control is to handle Web traffic consisting of thousands of concurrent flows with some degree of fairness then further enhancements to ECN are needed. Based on these results, an adaptive version of ECN (AECN), which can adjust max_p based on the round-trip time of a flow, is proposed in the next chapter.

Chapter 5

Adaptive ECN (AECN) for Heterogeneous TCP Flows

5.1 Introduction

In the last chapter, the simulation results by comparing the simulated performance of RED routers and ECN routers shows that ECN does provide better goodput and fairness than RED for heterogeneous flows in most cases. When the demand is held constant, the number of flows generating the demand has a negative effect on performance. Meanwhile, the simulations with many flows demonstrate that the bottleneck router's marking probability must be aggressively increased to provide good ECN performance when the number of flows increases.

Based on these simulation results, this chapter presents an adaptive version of ECN (AECN) that can further improve the performance of ECN on on the goodput or throughput and fairness by properly adjusting the relevant ECN parameters. Rather than treat all flows with a same max_p in ECN, AECN divides all flows competing for a bottleneck into three flow groups, and deploys a different max_p for each flow group so that a fragile flow can have higher chance to get a proper share of bandwidth when competing with a robust flow. Meanwhile AECN also adjusts min_{th} for each robust flow group and max_{th} to get higher performance when the total number of flows changes. Furthermore, AECN uses a mark-front strategy, instead of mark-tail strategy in standard ECN, to mark the first unmarked packet in the front of a corresponding flow queue so that the notification of congestion can be speeded up to a sender.

5.2 The Basic Algorithm of AECN

5.2.1 Assumptions

Just like standard ECN, AECN is used together with TCP congestion control mechanisms like slow-start and congestion avoidance. When an acknowledgement is not marked, the source follows existing TCP algorithms to send data and increase the congestion window. Upon the receipt of an ECN-Echo packet, the source halves its congestion window and reduces the *ssthresh*. In the case of a packet loss, the source follows the TCP algorithm to reduce the window and retransmit the lost packet.

AECN delivers congestion signals by sending CE packet, which sets the CE bit as 1, but determining when to set the bit depends on the average queue size. Like standard ECN, AECN uses the average queue length as in the proposal in [RAM99] and [RAM01]. Hence, when the average queue size of an AECN router is smaller than min_{th}, no marking action occurs when a new packet comes in the router. When the average queue size is between min_{th} and max_{th}, a marking action to a packet (could be the packet at the front or the tail of the router queue) will happen with a probability. Once the average queue size is above max_{th}, all the incoming packets will be dropped as standard ECN does.

This study has a few assumptions as follows: (1) Receiver windows are large enough so that the bottleneck is in the network. (2) A sender always has packets to send and will send as many packets as its window allows. (3) Receivers acknowledge every received packet and there are no delayed ACKs. (4) Router queue length is measured in packets and all packets have the same size. (5) The TCP header has an enough extra space to contain the round-trip time information in each packet, and the round-trip time has small enough time granularity.

5.2.2 Terminologies

5.2.2.1 Flow Queue

A flow queue is a virtual queue, which refers to a queue storing the address of each packet in the router queue for a specific flow group.

AECN divides all flows into three flow groups, i.e., fragile, average and robust flow group, depending on the round-trip time of each flow. Accordingly, AECN has three flow queues, which are called fragile, average and robust flow queues respectively. Each flow queue maintains a range of round-trip times to be compared with the round-trip time in a new packet for deciding which flow queue this new packet belongs to, and a maximum marking probability max_p for deploying different marking probabilities.

5.2.2.2 Base max_p

Just as standard ECN maintains a maximum marking probability, max_p, AECN maintains a **base max_p** for the average flow group. While the marking probability for the fragile flow group is the most conservative to permit fragile flows to get more bandwidth when competing with the other two flow groups, and the marking probability for robust flows is the most aggressive. Hence, AECN sets the maximum marking probability to (**base max_p / α**) for fragile flow group, and to (**base max_p * β**) for the robust flow group. α and β both are constants, and their concrete values depend on the average round-trip time of each flow group.

5.2.2.2 Unlockout Range

As was shown in last chapter, ECN is better than RED in some specific ranges of the number of flows. These ranges, called the unlockout range, are determined by whether the lockout phenomenon is heavy or not. Once the lockout phenomenon occurs seriously due to the high number of flows, ECN and RED, like other TCP congestion control mechanisms, both don't help much to control the congestion effectively. Therefore, it doesn't make sense to compare the performance of AECN and ECN, and RED, beyond the unlockout range.

5.2.3 Strategies

In most ECN implementations, when congestion happens, the congested router marks the incoming packets that just enter the router queue. When the buffer is full or when a packet needs to be dropped as in RED, some implementations, e.g. ***ns-2*** simulator, uses the "drop from front" option as suggested in [YIN90]. A brief discussion of drop from front in RED can be found in [FLO98b]. However, for packet marking, ***ns-2*** still pick the just-incoming packet to mark, rather than the front packet.

5.2.3.1 Round Trip Time Strategy

As mentioned in section 5.2.1, AECN assumes that the TCP header has enough reserved space to contain the round-trip time of each flow. The decision which flow group a new packet belongs to is dependent on the round-trip time of the packet. Before a source sends out a new packet, the round-trip time of the last ACKed outgoing packet of this flow is added into the TCP header of this new packet. The computation of round-trip time in ***ns-2*** simulation uses the round-trip time mechanism of TCP Vegas [AHN95] [BRA95].

5.2.3.2 Marking Front Strategy

One of the weaknesses of mark-tail strategy is its discrimination against new flows [LIU01]. Consider the time when a new flow joins the network, but the buffer of the congested router is occupied by packets of old flows. In the mark-tail strategy, the packet that just arrived will be marked, but the packets already in the buffer will be sent without being marked. The ACK of the sent packets will increase the window size of the old flows. Therefore, the old flows which already have large share of the bandwidth will get more bandwidth. However, the new flow with small or no share of the resources has to backoff., since its window size will be reduced by the marked packets. Contrary to the mark-tail strategy, when a packet needs to be picked for marking, the mark-front strategy will pick the first unmarked packet in the front of the queue and mark it. Connections with large buffer occupancy will have more packets than connections with small buffer occupancy. Compared to the mark-tail strategy that let the packets in the buffer escape the marking, mark-front strategy helps to alleviate the lockout phenomenon [LIU01]. Therefore, we can expect that mark-front strategy would be fairer than marking-tail strategy. It’s well-known that TCP’s discrimination against fragile flows that have large RTT or small *cwnd* [QIU99]. The cause of the discrimination is similar to the discrimination against new flows. If fragile flows and robust flows start at the same time, robust flows will receive their ACKs faster and therefore grow faster, and then get more bottleneck bandwidth. When congestion happens to the bottleneck, there are more packets from robust flows than those from fragile flows. With the mark-tail strategy, packets already in the router queue will not be marked but only newly arrived packets will be marked. This may cause robust flows to grow ever larger than fragile ones. Mark-front strategy alleviates this discrimination by treating all packets in the buffer equally. Packets already in the buffer may also be marked. In this way, fragile flows can get larger bandwidth, which would make AECN fairer to all flows. Meanwhile, mark-front strategy can hasten the transmission of congestion notification to the sender since the marked packet doesn't need to wait in the router queue. In this way, the sender can get congestion notification earlier to execute congestion action.

5.2.4 Basic Algorithm

The basic algorithm of AECN consists of the following three steps (See Algorithm 5.1). The relationship between the AECN router queue and the three flow queues is shown in Figure 5.1.

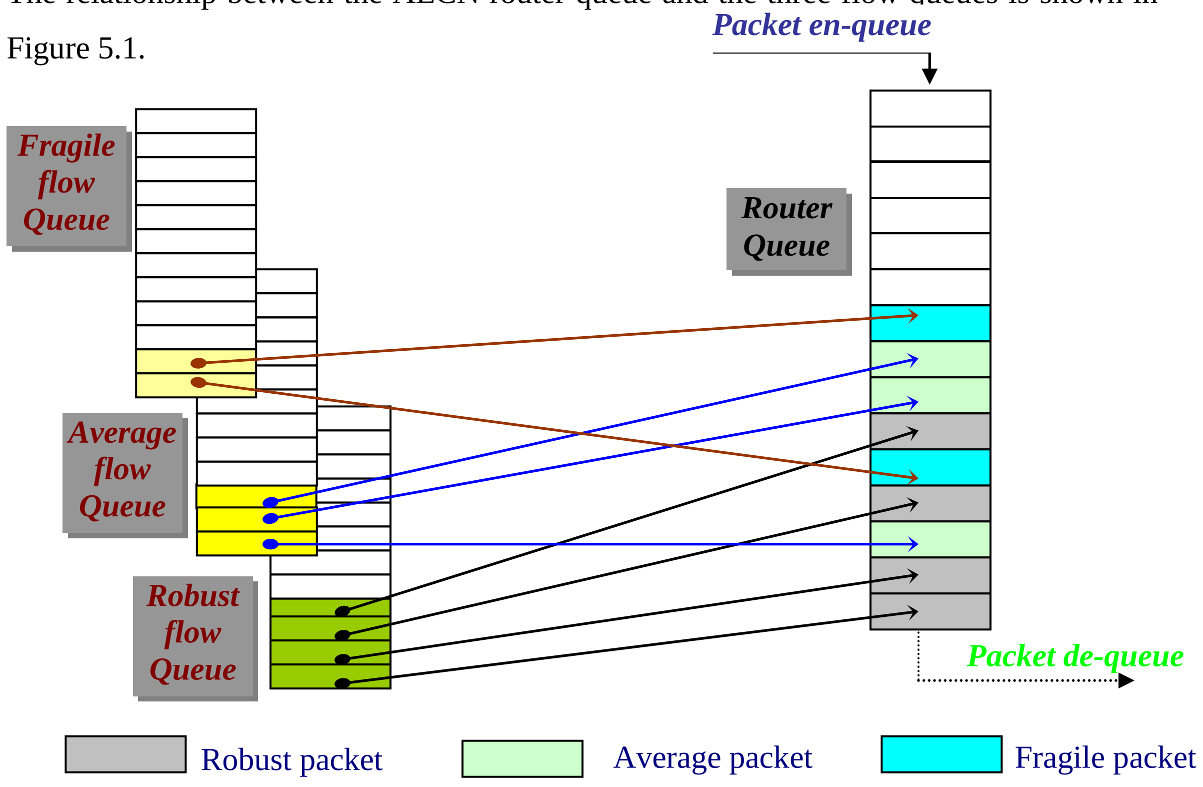

Figure 5.1 The Relationship between Three Flow Queues and Router Queue

1. Initialization:

Initially, all queues are set empty, including the router queue and the three flow queues.

2. En-queue:

When a new packet comes into the router, AECN will check:

a). If avg >= max_{th}, AECN will drop this incoming packet just as standard ECN does.

b). If avg is below max_{th},

1). Add this packet into the router queue.

2). Deploy ***AECN RTT strategy*** for deciding which flow queue this packet belongs to.

AECN RTT strategy:

1). Get RTT contained in the incoming packet, (in milliseconds)

2). Decide which flow queue this packet belongs to:

```
If RTT is in the RTT range of robust flow queue
    status = ROBUST_FLOW_QUEUE;
else if RTT is in the RTT range of average flow queue
    status = AVERAGE_FLOW_QUEUE;
else if RTT is in the RTT range of fragile flow queue
    status = FRAGILE_FLOW_QUEUE;
```

3). If avg is between min_{th} and max_{th}, deploy ***AECN marking-front strategy*** to mark the first unmarked packet in the corresponding flow queue.

AECN marking-front strategy:

1). With the value of status, find the first unmarked packet recorded in the corresponding flow queue.

2). Select a maximum marking probability: max_p,

```
if (status == ROBUST_FLOW_QUEUE)
    $max_p$ = min{ (base-$max_p$ * $\beta$) , 1};
else if (status == AVERAGE_FLOW_QUEUE)
```

```
                max_p = base-max_p ;
            else if (status == FRAGILE_FLOW_QUEUE)
                max_p = base max_p / α;
        3). Update the marking probability with the new max_p
        4). Mark the selected packet with the new marking probability.
3.  De-queue
Once an outgoing packet leaves the AECN router, AECN will check:
    a). If (status == ROBUST_FLOW_QUEUE)
            Remove the first node in the robust flow queue.
        Else if (status == AVERAGE_FLOW_QUEUE)
            Remove the first node in the average flow queue;
        Else if (status == FRAGILE_FLOW_QUEUE)
            Remove the first node in the fragile flow queue;
    b). Remove the packet from the router queue.
```

Algorithm 5.1 *the basic algorithm of AECN*

5.3 Implementation in *ns-2*

To implement AECN, some code needed to be added or modified in ***ns-2.*** In this study, all flows use the TCP variant: TCP Reno. The implementation of AECN includes two aspects:

1. The implementation of AECN RTT strategy in TCP Reno:

   To obtain the RTT of each flow, one variable, r_rtt_, is added into the packet header of TCP Reno in ***ns-2***. Each time a TCP Reno source receives an ACK for a specific flow, the real round-trip time of this flow is updated with the code shown in **Appendix A**. Before a source sends out a new packet, the updated round-trip time of this flow is put into this new outgoing packet.

   For the first packet sent out from a source for connection setup, i.e. handshaking, its RTT would be 0 (ms). The AECN router takes the first packet of each flow as being from an average flow.

2. The implementation of AECN based on standard ECN

The second part of AECN implementation includes implementing the three flow queues (See **Appendix B**) and modifying the code based on standard ECN.

The address of each packet in the router queue is kept by the corresponding flow queue, which maintains the current number of the packets of the same flow group. Once a new packet arrives at the router queue and the average queue size is between min_{th} and max_{th}, the address of this packet in the router queue is pushed into a corresponding flow queue, and a different maximum marking probability is deployed (See REDQueue::enque(), and REDQueue::drop_early() in Appendix B).

When a packet leaves the router queue, its address information in a flow queue will be removed from the corresponding flow queue (See REDQueue:deque() in Appendix B).

5.4 Simulation Scenarios

To compare the performance of standard ECN and AECN, a series of simulations with the ***ns-2*** simulator were run. As mentioned earlier, the algorithm of standard ECN in ***ns-2*** simulator is changed to implement the basic algorithm of AECN. The basic network simulation topology (Figure 5.2) used in the experiment is the same as that shown in last chapter (See Figure 4.1), but with some different parameter settings to make it closer to a real network configuration.

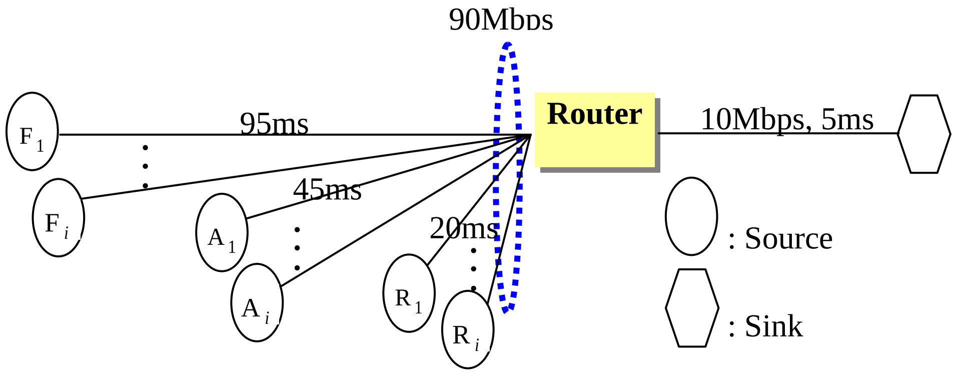

Figure 5.2 Simulation Topology

In [CHR00], Christiansen says that a typical round-trip time of a flow would be in the range of 7 ms to 137 ms. Accordingly, in the simulation configuration, AECN supposes that the round-trip time of a robust flow, a fragile flow and an average flow are in the ranges of [0.5 ms- 75 ms), [150 ms, +), and [75 ms, 150 ms). Meanwhile, for a packet with a RTT of 0 ms sent out by any flow, AECN takes it as an average flow.

With the basic configuration shown in Figure 5.2, the link delays between a source and the router are set 95 ms, 45 ms and 20 ms for fragile, average and robust flows. Thus, the fixed round-trip times for fragile flows, average flows and robust flows, without taking into account the router queue delay, are 200 ms, 100 ms and 50ms. A FTP application runs on each source using TCP Reno. Each source has a window size of 64 packets. The data packet size, including all headers, is 1000 bytes, and the acknowledgement packet size is 40 bytes.

The total capacity of the bandwidths from all sources is fixed at 90 Mbps. The router has a fixed physical size of 120 packets, and min_{th} and max_{th} (if not explained particularly) are 10 and 30 packets respectively. The bottleneck link has a bandwidth of 10 Mbps with a link delay of 5ms. Half of the number of flows in each flow group start at time 0, the second half start at time 2 seconds. That is, if there are 60flows. 10 fragile, 10 average and 10 robust flows start to run at second 0, and the rest 30 flows at the 2nd second. All simulations were run for 100 seconds.

5.5 Simulation Preliminaries

The main purpose of the simulations is to compare the performance of AECN and standard ECN. But, before running simulations for the performance comparison between AECN and standard ECN, some preliminary simulations were run for confirming the behavior of ECN and RED in the simulation configuration in Figure 5.2. One difference between the simulation configurations shown in Figure 5.2 and Figure 4.1 is that in Figure 5.2 the aggregate capacity of the bandwidth between all sources and the bottleneck router is 90 Mbps while it’s 300 Mbps in Figure 4.1. It’s believed that the change of the aggregate capacity may change the unlockout range. As shown in the last chapter, ECN is better than RED in the unlockout range. Therefore, this study concentrates on comparing the performance of AECN and standard ECN in the unlockout range even though the simulation results beyond the range are also presented in this chapter.

Figures 5.3 through 5.12 present the comparison of standard the ECN and RED on each metric. Figures 5.3 and 5.4 show the Jain's fairness index and goodput of standard ECN goes down dramatically at 120 and 240 flows for max_p being 0.5 and 0.8 respectively. Figure 5.5 presents that even though ECN has better goodput than RED with 60 flows, the visual max-min fairness observed from this figure for both algorithms is relatively low since the gap between the goodput of robust flow group and fragile flow group is so obvious.

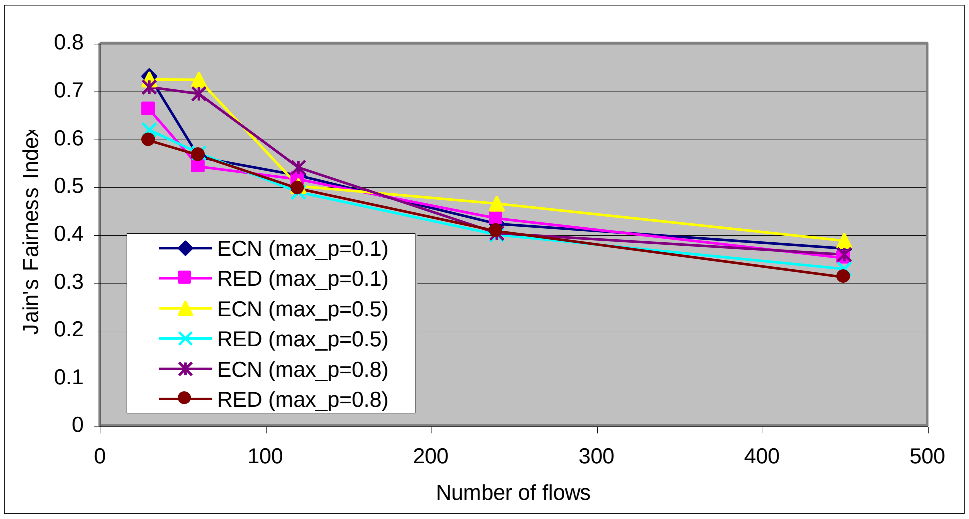

Figure 5.3 Jain's Fairness Index with the Number of Flows

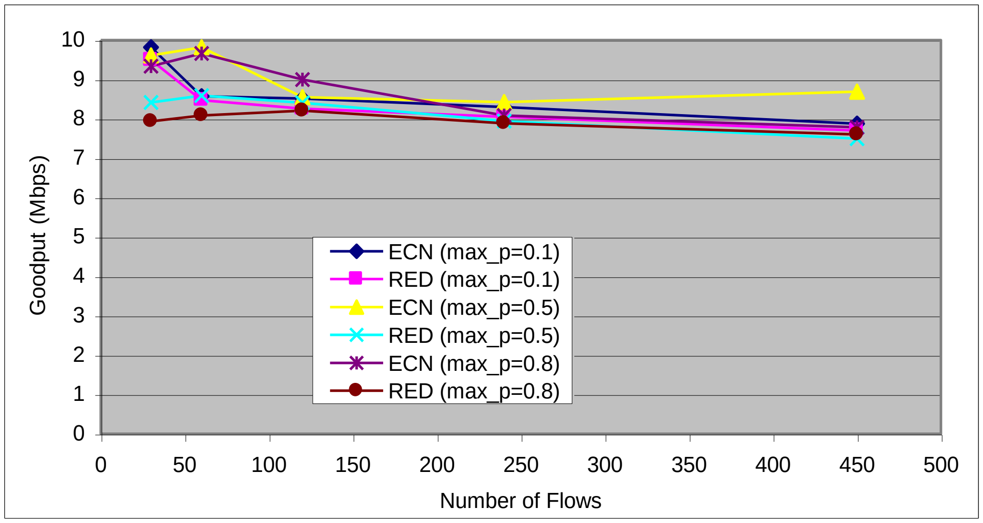

Figure 5.4 Goodput with the Number of Flows

Figures 5.6 through 5.8 show there are some flows gets locked out in some particular periods (The ▭ regions list some of the lockout occurrences). In these figures, y-coordinate presents the flow id of each packet. The flows with flow No. 0-19 refer to the 20 fragile flows, those with flow No. 20-39 are the 20 average flows, and the others with flow No. 40-59 are the robust flows. As shown in Figure 5.6, the robust flow group gets the most packets marked, while the fragile flow group gets the least. Figure 5.7 shows that there are packet drops around at second 50, 67 and 78. While compared Figure 5.8 with Figure 5.7, it's easy to find that RED has many more packet drops than ECN since RED drops an incoming packet probabilistically when the average queue size is between min_{th} and max_{th} .

Figures 5.9 through 5.11 present the statistics of dropped or marked packets of RED and ECN with 120 flows. The flows with flow No. 0-39 refer to the 40 fragile flows, those with flow No. 40-79 are the 40 average flows, and the others with flow No. 80-119 are the 40 robust flows. It's obvious that the lockout phenomenon become much heavier for 120 flows than that for 60 flows, and more packets get dropped or marked.

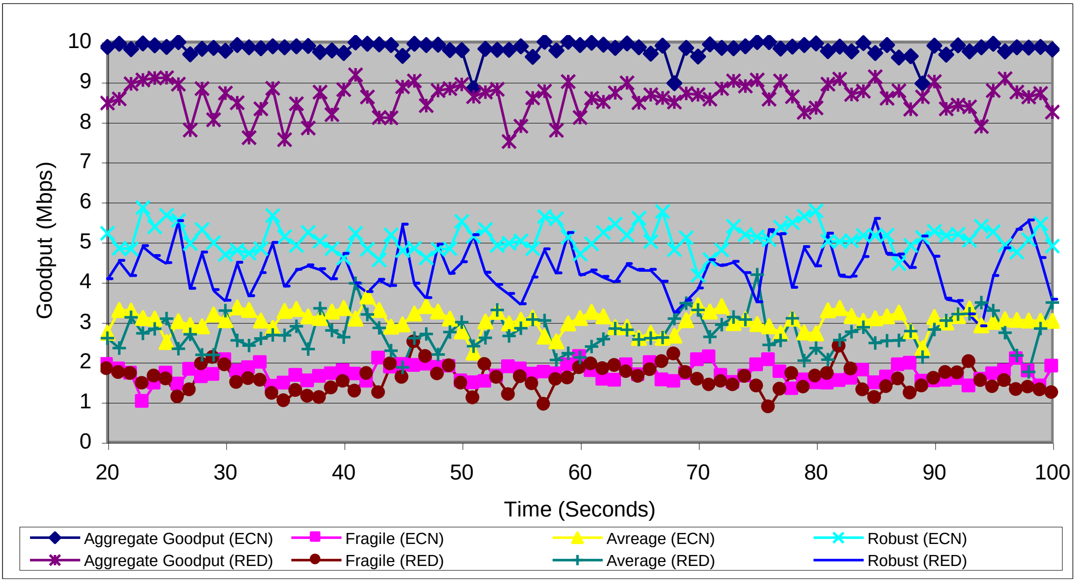

Figure 5.5 Goodput Distribution between ECN and RED,

60 flows, $\boldsymbol{max}_p$ =0.5.

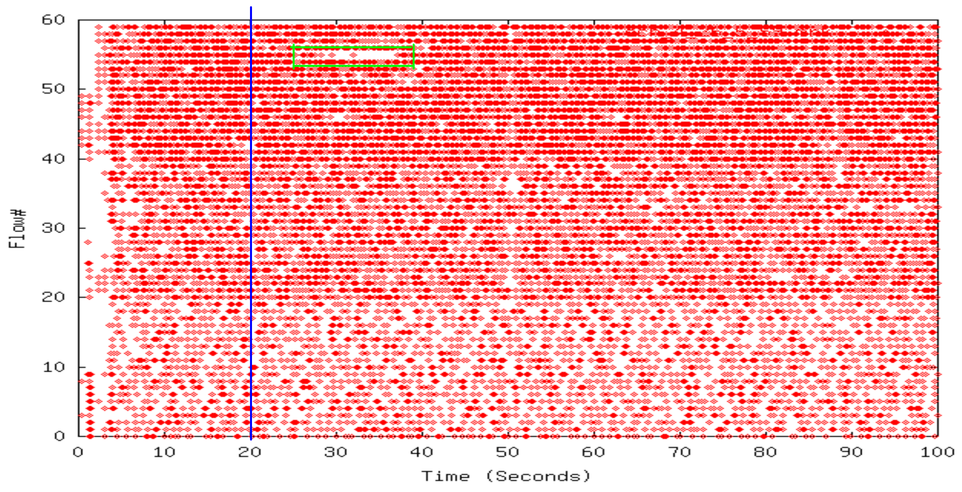

Figure 5.6 ECN Marked packet Statistics, 60 flows, max_p =0.5.

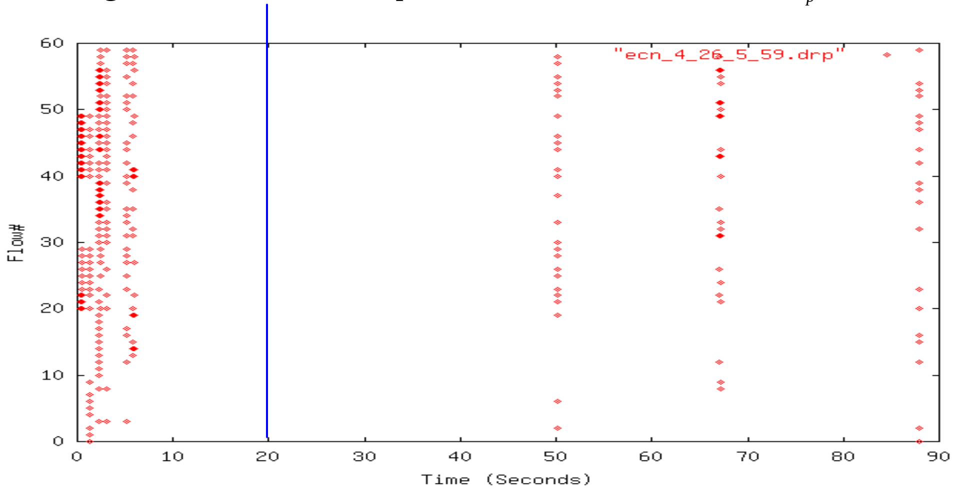

Figure 5.7 ECN Dropped packet Statistics, 60 flows, max_p =0.5.

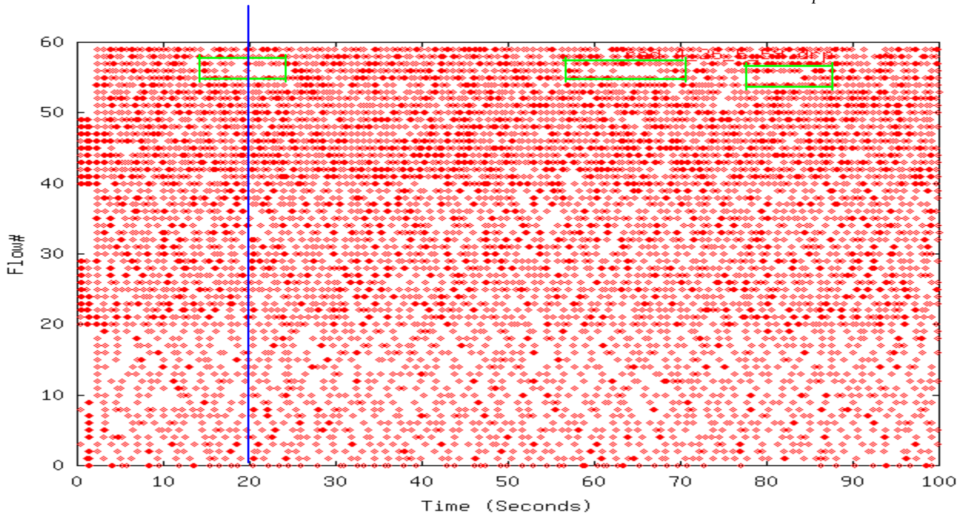

Figure 5.8 RED Dropped packet Statistics, 60 flows, max_p =0.5.

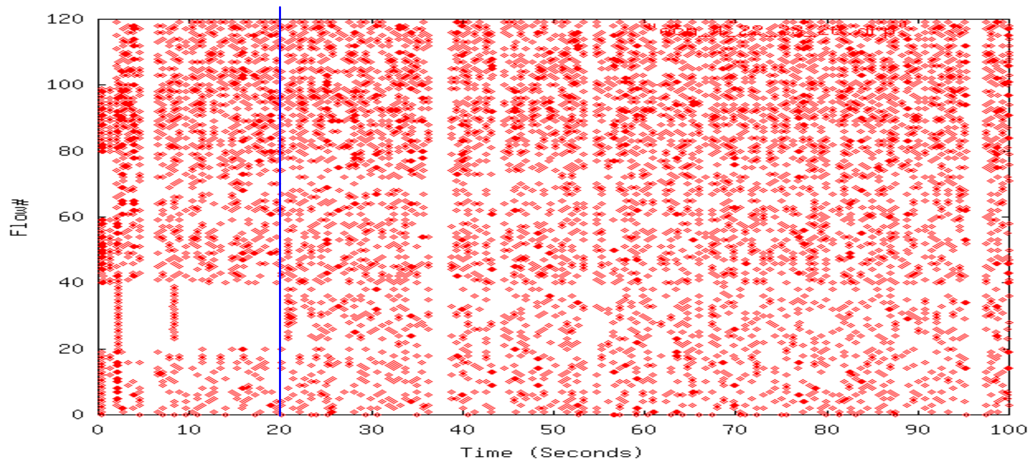

Figure 5.9 ECN dropped packet Statistics, 120 flows, *max* $_p$ =0.5.

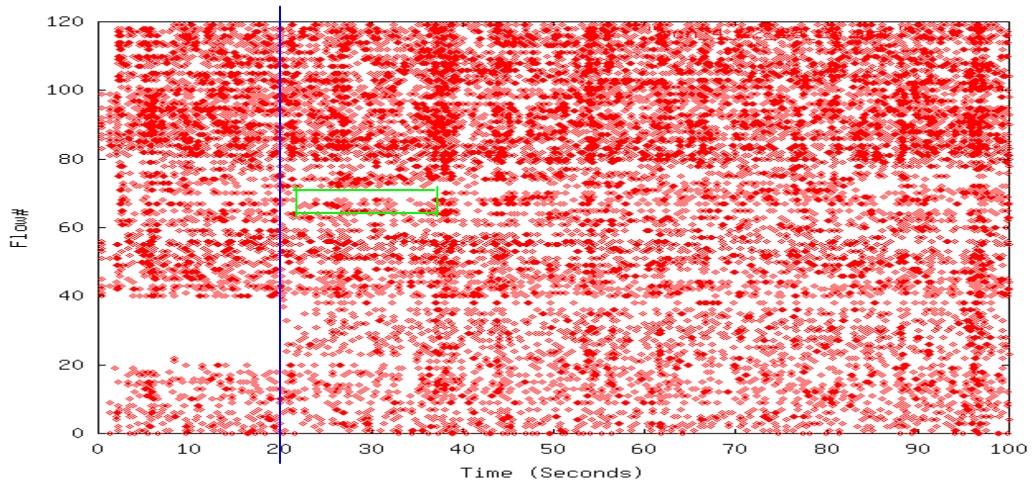

Figure 5.10 ECN marked packet statistics, 120 flows, *max* $_p$ =0.5.

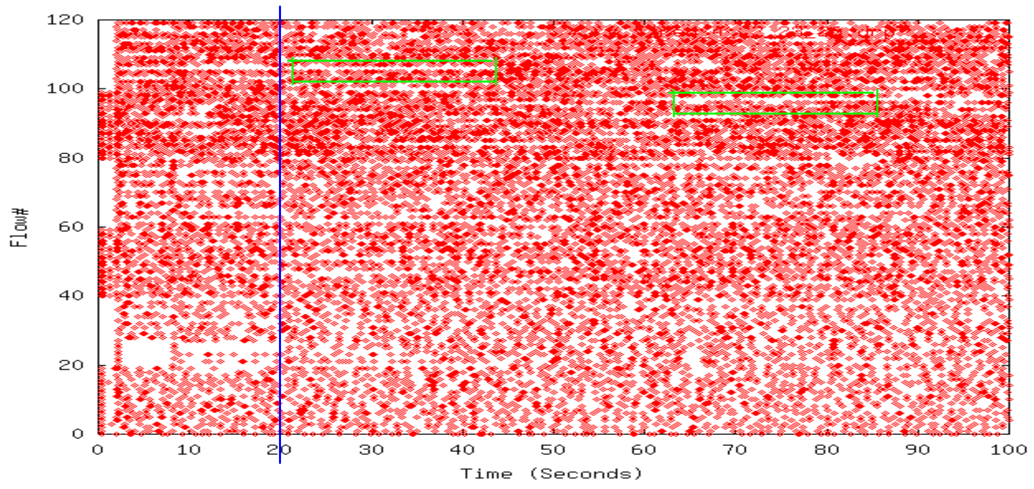

Figure 5.11 RED dropped packet statistics, 120 flows, *max* $_p$ =0.5.

The lockout phenomenon is definitely more serious after 240 flows. Therefore, this study sets the unlockout range at 240 flows and below with the simulation topology shown in Figure 5.2, and focuses on evaluating the performance of AECN and ECN within this range.

Figures 5.12 through 5.15 show the similar results as those in last chapter. Both ECN and RED can get high aggregate throughput (Figure 5.12). However, when $\boldsymbol{max}_p$ is equal to 0.8, ECN may get less aggregate throughput than RED in some range of the number of flows. Nevertheless, ECN still has better goodput than RED in this range. The possible reasons for that include two aspects: (1). $\boldsymbol{max}_p$ =0.8 is so high that a high number of packets get marked which cause the senders to slow down frequently; (2). Since ECN marks the incoming packets, instead of dropping, the ECN router will have more packets enter the router queue than RED when the number of flows is high. This causes ECN to have higher queue delay (See Figure 5.15) and higher chance than RED to hit the max_{th} , which causes ECN to drop the packets. Figures 5.13-5.15 show that ECN gets a little higher delay than RED in the unlockout range since ECN router has higher average queue size. But, once out of the unlockout range, there's no difference on delay for ECN and RED since both have an average queue size stably around max_{th} when the number of flows is really high and the congestion is heavy.

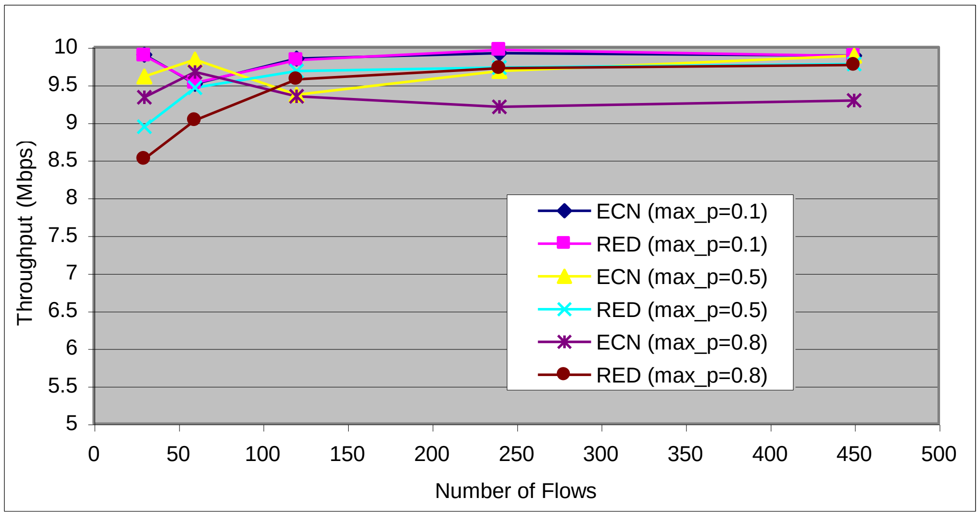

Figure 5.12 Throughput with the Number of Flows

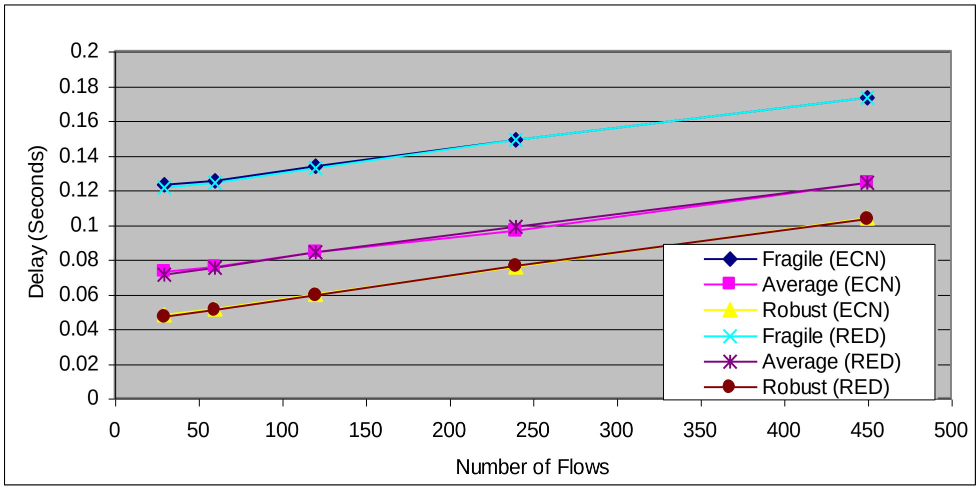

Figure 5.13 One-way Delay with the Number of Flows (max_p =0.1).

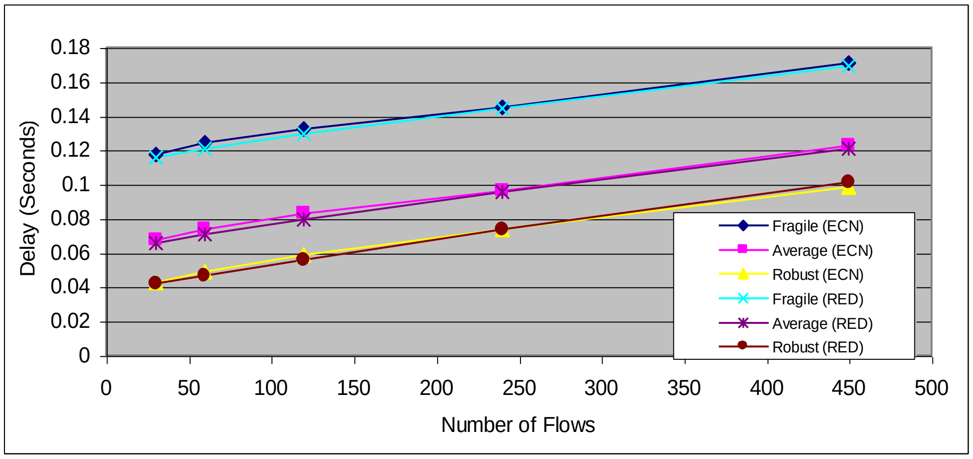

Figure 5.14 One-way Delay with the Number of Flows (max_p =0.5).

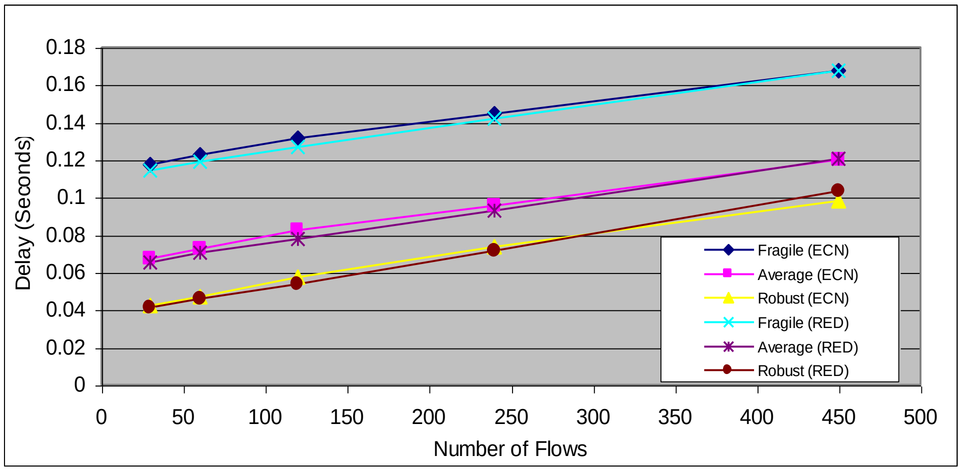

Figure 5.15 One-way Delay with the Number of Flows (max_p =0.8).

5.6 Performance Comparison Between AECN and ECN

This section concentrates on presenting the simulation results for AECN, and comparing it with ECN.

5.6.1 The Selection of α and β

AECN uses two parameters **α, β** to implement two separate maximum marking probabilities for fragile flow group and robust flow group. Therefore, simulations were run to check the influence of different **α, β** on AECN performance**.** For simplicity, AECN first supposes that **α** is equal to **β.** Figure 5.16 through 5.19 shows the selection of **α** on the performance of AECN with 60 flows and 120 flows on goodput and Jain's fairness index.

In the study, simulations with **α** of 1.0, 1.5, 2, 2.5, 2.6, 2.8 and 3.0 were run. Figures 5.16 and 5.18 present the change of Jain's fairness index with **α.** As shown in figure 5.17 and 5.19, when **α** increases from 1.0 to 3.0, the marking probability for the fragile flow group becomes less and less aggressive, so they get more bandwidth and then their goodput increases. While the marking probability for the robust flow group becomes more and more aggressive, and gets less bandwidth. For 60 flows (Figure 5.16), when **α** is equal to 2.5, AECN gets the highest Jain's fairness index. Meanwhile, even though the selection of **α** in this case seems that it doesn't change much the aggregate goodput share of the average flow group, when **α** increases from 1.0 to 3.0, the goodput of the robust flow group comes to be less while the fragile flow group gets more and more goodput and the average flow group tends to remain stable on goodput. Especially, when **α** is below 2.6, AECN comes to behave better on the visual max-min fairness with **α** increasing, and reaches the best when **α** is equal to 2.6. Considering the fact that AECN has the highest Jain's fairness index at the point of **α** of 2.5 and tends to decrease after this point, it's believed that **α** at the point of 2.5 is preferable for AECN when the number of flows is 60.

Figure 5.18 and 5.19 further shows the results of the selection of α on the performance of AECN with 120 flows. When α is equal to 2.5, AECN still gets the highest Jain's fairness index, and when α is above 2.5, the goodput and Jain's fairness index of AECN tends to decrease. Therefore, for 120 flows, α at the point of 2.5 is also preferable for AECN even though the visual max-min fairness at the point of 2.6 is the best.

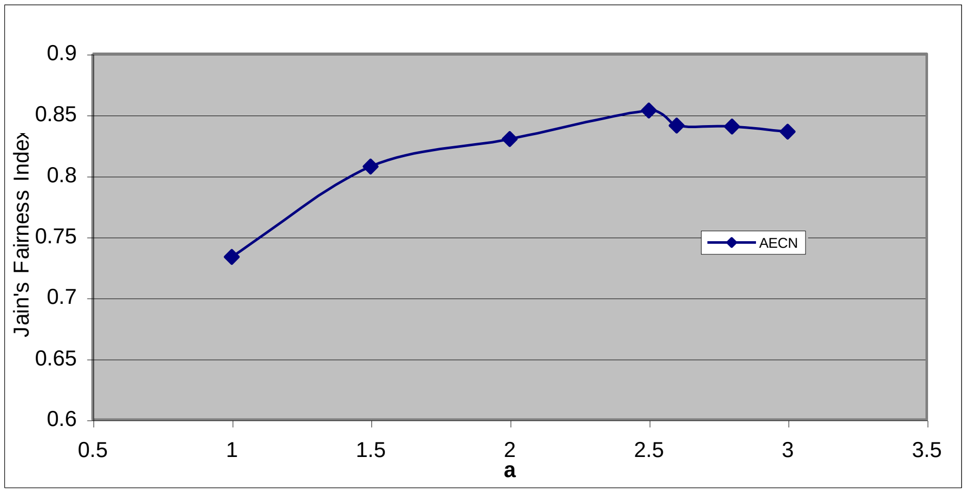

Figure 5.16.Jain's Fairness Index with α, ($\alpha = \beta$),

60flows, **base-*max*** $_p$ =0.5.

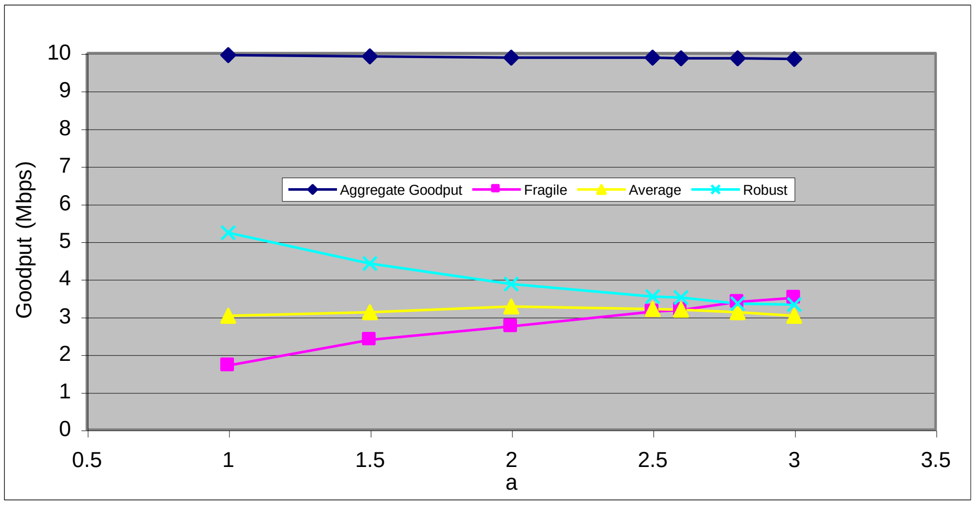

Figure 5.17. Goodput with α, ($\alpha = \beta$),

60flows, **base-*max*** $_p$ =0.5.

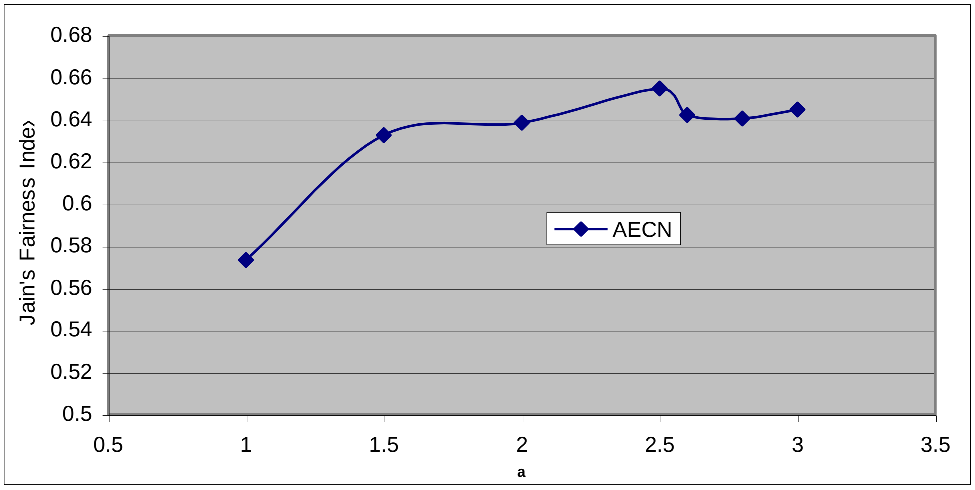

Figure 5.18. Jain's Fairness Index with α, (α = β),

120flows, **base max_p**=0.5.

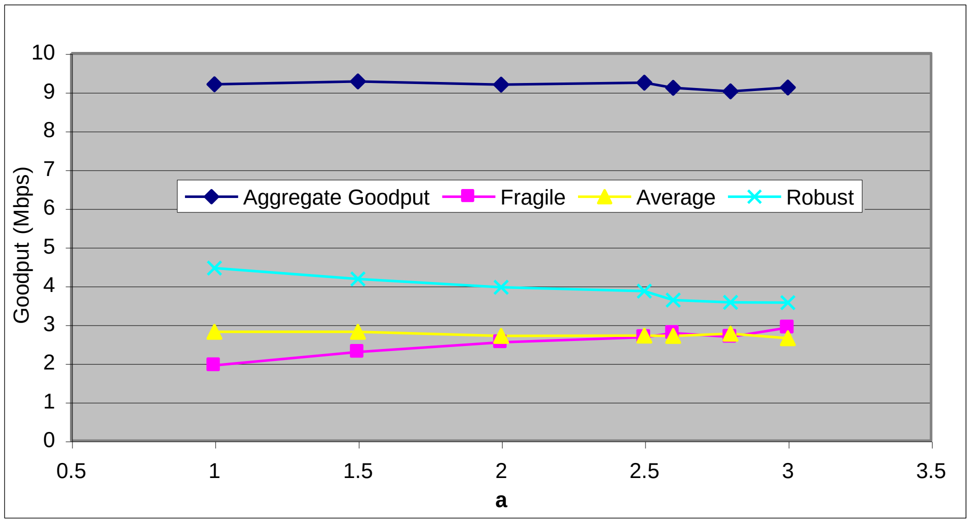

Figure 5.19. Goodput with α, (α = β),

120flows, **base-*max*** $_p$ =0.5.

From the above observations, **α** at the point of 2.5 is selected for AECN to compare it with standard ECN before further refining AECN.

One more interesting thing, observed from figures 5.3, 5.4 and Figure 5.16 through 5.19, is that when **α** is 1.0, which makes AECN use marking-front strategy with a same

max $_p$ [1] for the three flow groups, AECN with marking-front strategy has a little higher goodput and Jain’s fairness index (See the statistics in table 5.1) than standard ECN, which uses marking-tail strategy. For example, when the number of flows is 60, the mean Jain’s fairness index and aggregate goodput of AECN with α equal to 1 are 0.733493 and 9.9499Mbps, while the mean Jain’s fairness index and aggregate goodput of standard ECN are 0.722963 and 9.8198 Mbps. When the number of flows is 120, the mean Jain’s fairness index and aggregate goodput of AECN with α equal to 1 are 0.573331 and 9.200395Mbps, while the mean Jain’s fairness index and aggregate goodput of standard ECN are 0.500629 and 8.552889Mbps

	Jain’s Fairness Index		Goodput (Mbps)	
	AECN α=1.0	**ECN**	**AECN α=1.0**	**ECN**
60 flows	0.733493	0.722963	9.9499	9.8198
120 flows	0.573331	0.500629	9.200395	8.552889

Table 5.1 Performance statistics between AECN and ECN,
AECN: **base-*max*** $_p$ =0.5, α = 1.0**; ECN: *max*** $_p$ ***=0.5;***

5.6.2 Performance Evaluation of AECN with α = 2.5

This section presents the simulation results of AECN with α of 2.5 ($\alpha = \beta$) with different number of flows, and makes comparisons of AECN with standard ECN. All the other parameter settings for the simulations in this section are the same described in section 5.5.

5.6.2.1 Goodput

Figures 5.20 and 5.21 present that AECN keeps getting higher goodput than ECN when the number of flows increases. This shows that by restricting robust flows with an

[1] That is, the maximum marking probabilities for all flows of the three flow group are equal to **base *max*** $_p$

aggressive $\boldsymbol{max}_p$ and encouraging fragile flows with a conservative $\boldsymbol{max}_p$**,** AECN gets fewer retransmissions than standard ECN. The main contribution is that AECN marks many more packets than ECN, which reduces the chance that the average queue size hits max_{th} .

Observing both Figures 5.20 and 5.21, when **base-*max*** $_p$ = 0.5, AECN has better aggregate goodput and visual max-min fairness for 30 and 60 flows; but the aggregate goodput and visual max-min fairness goes down for 120 flows. When **base-*max*** $_p$ = 0.8, AECN gets better aggregate goodput and visual max-min fairness for 120 flows, but the advantage decreases when the number of flows is 240 flows.

From these observations, we can conjecture that the selection of **base-*max*** $_p$ for AECN should be adaptive to the number of flows.

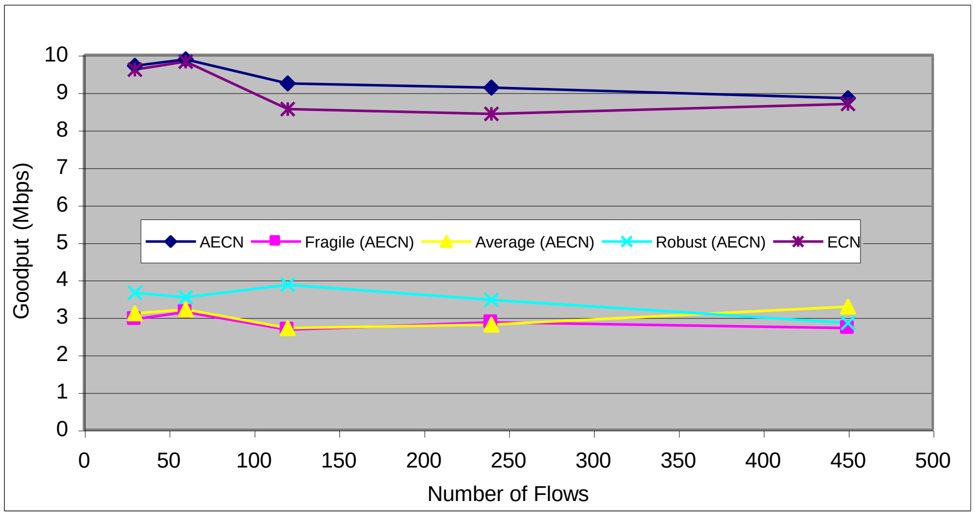

Figure 5.20 Goodput with the number of flows, base-*max* $_p$ = 0.5.

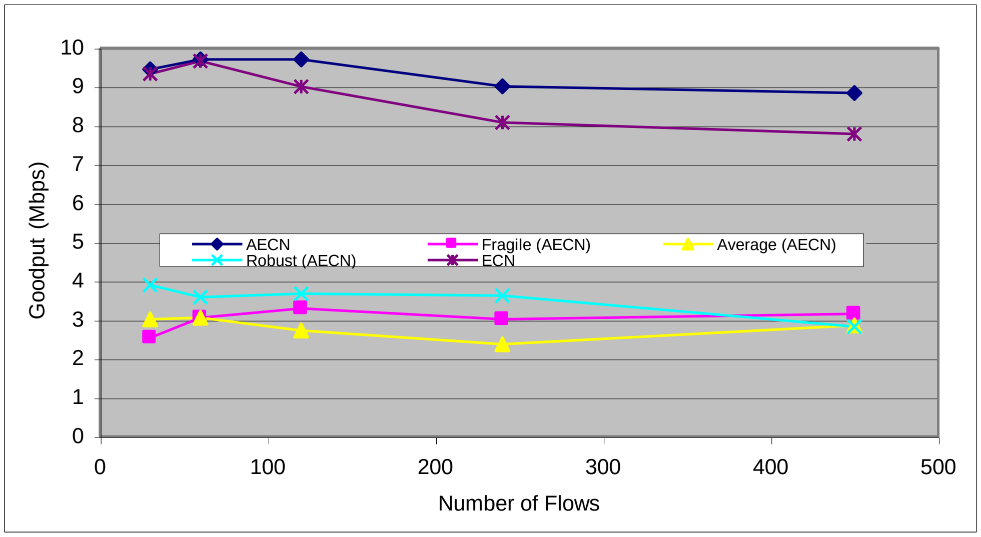

Figure 5. 21 Goodput with the number of flows, base-$max_p = 0.8$.

5.6.2.2 Throughput

Figures 5.22 and 5.23 show that AECN can also keep getting higher throughput than standard ECN. Combined with Figure 5.20 and 5.21, AECN is believed to have a smaller loss rate than standard ECN. That's mainly due to the fact that marking the robust flows more aggressively can reduce the chance that the average queue size hits max_{th}, which cause fewer packet drops. This result can be further demonstrated by comparing the statistics of dropped and marked packets shown in Figures 5.24 and 5.27 with Figures 5.6 through 5.11. These figures show that AECN has fewer packet-drops than ECN, but more packet-marks than ECN. As shown in Figure 5.25, when the number of flows is 60, AECN has no drop after the 20th second, while ECN has drop occurrence in several short periods (See Figure 5.7). Furthermore, the distribution of marked packets of AECN with 60 flows is more stable than ECN. In the case of 120 flows (See Figure 5.26 and 5.27), it's more obvious that AECN gets much fewer packet drops than ECN (See Figures 5.9 and 5.10), and gets more marks. This results shows that AECN can, to some extent, further alleviate the occurrence of lockout phenomenon and create short periods where there are no drops (See Figures 5.26 and 5.27).

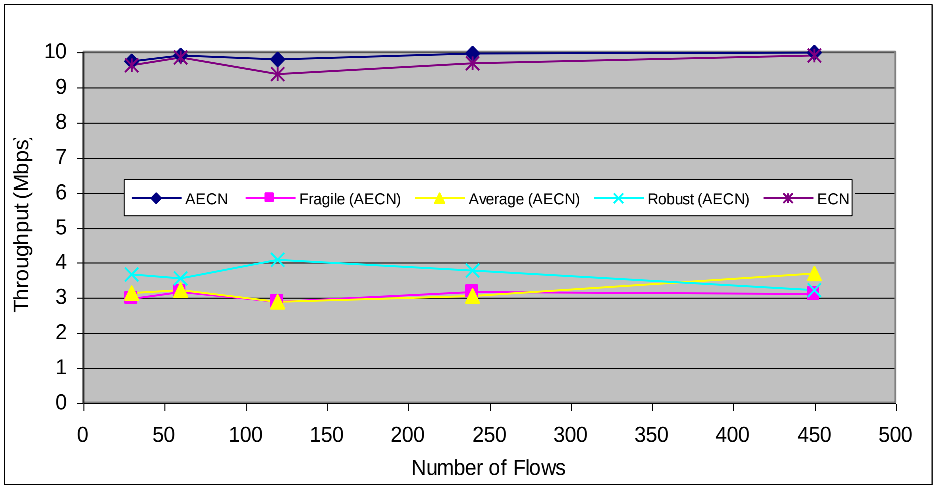

Figure 5.22 Throughput with the number of flows, base-max_p = 0.5.

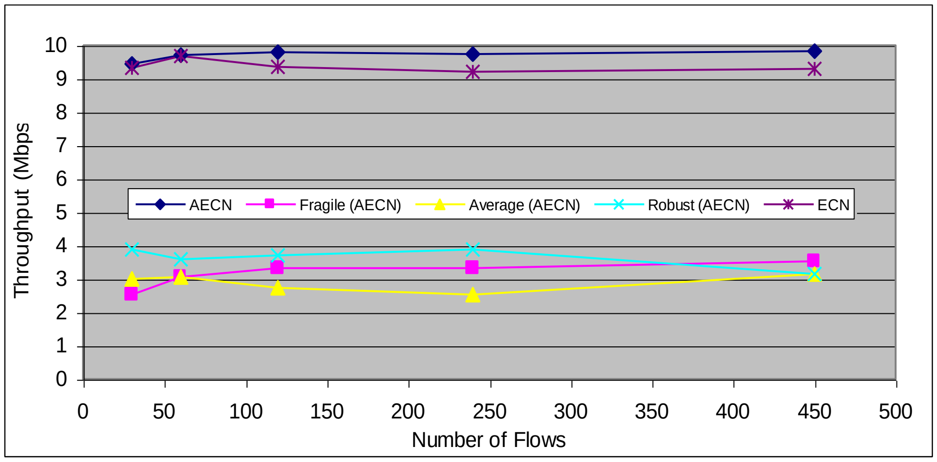

Figure 5.23 Throughput with the number of flows, base-max_p = 0.8.

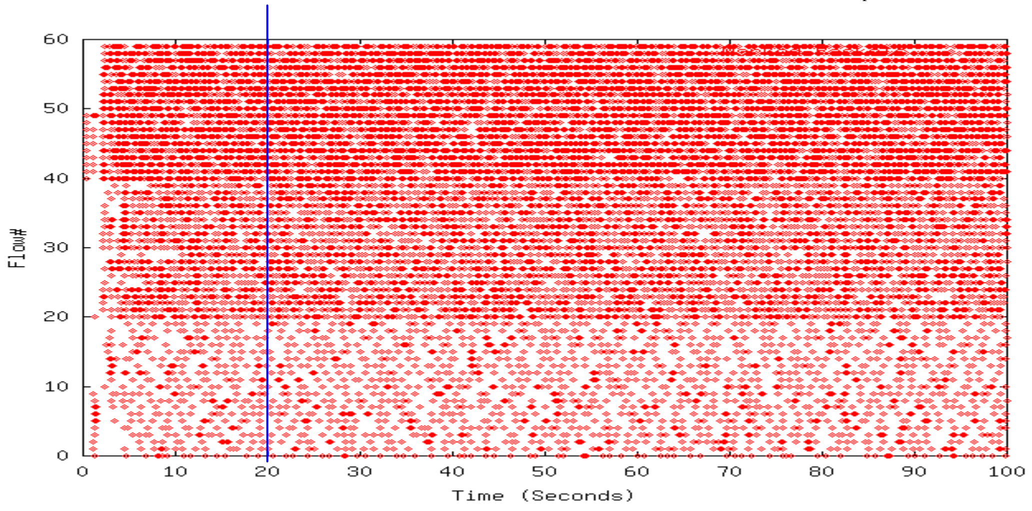

Figure 5.24 AECN marked packet Statistics, 60 flows, max_p =0.5.

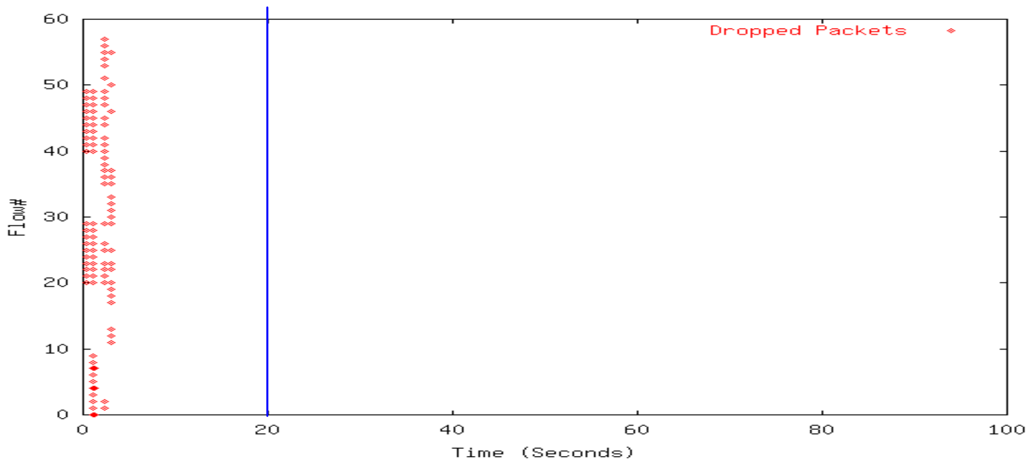

Figure 5.25 AECN dropped packet Statistics, 60 flows, max_p =0.5.

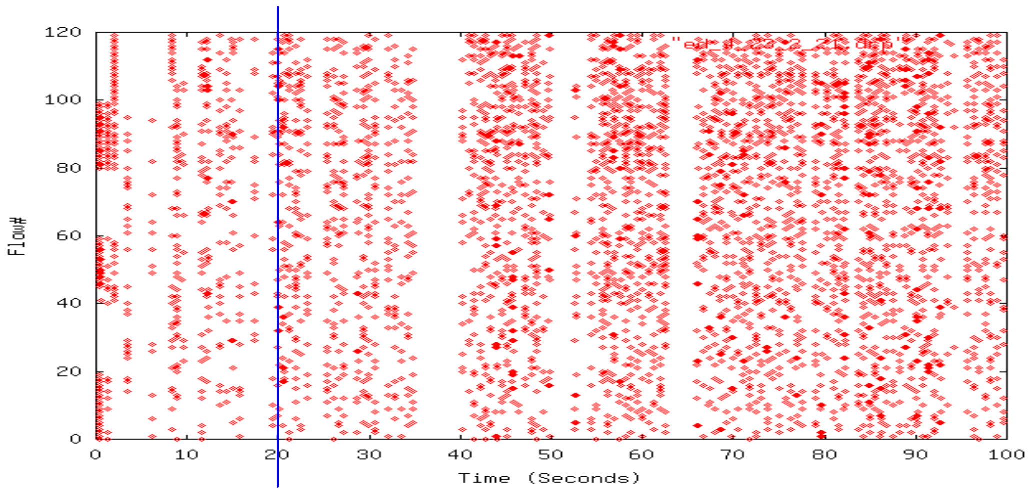

Figure 5.26 AECN dropped packet Statistics, 120 flows, max_p =0.5.

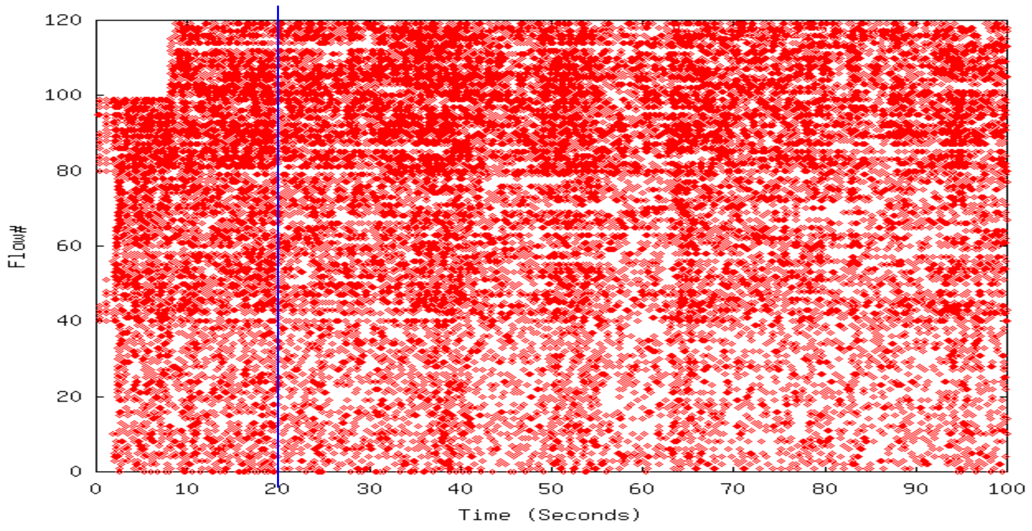

Figure 5.27 AECN marked packet Statistics, 120 flows, max_p =0.5.

One phenomenon shown in Figure 5.26 and 5.27 is that during some periods AECN gets no drops but marks. Figure 5.28 shows the reason why this phenomenon may happen. As shown in Figure 5.28, in those periods the average queue size almost seldom hits max_{th} , but stay quite stably close to max_{th} .

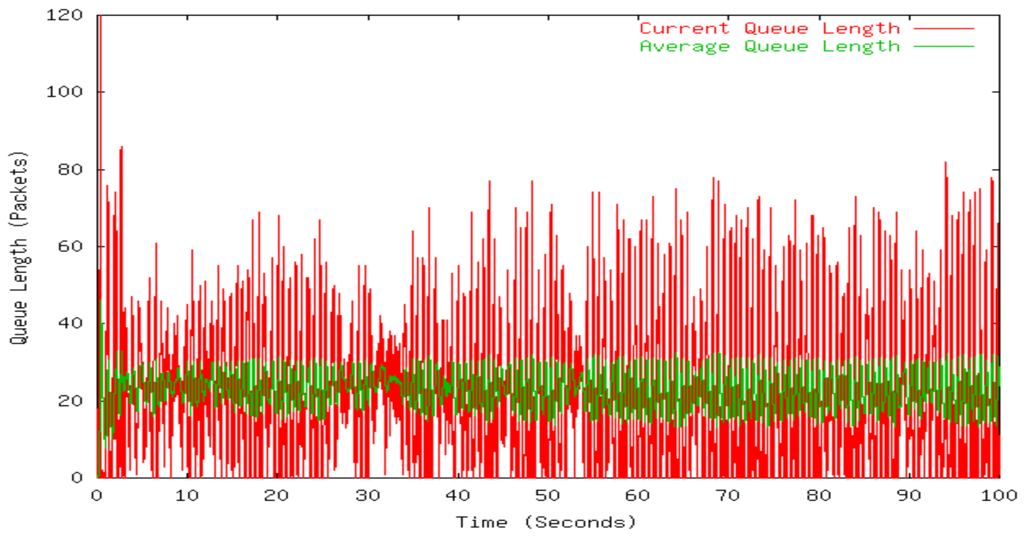

Figure 5.28 AECN Queue Length Change with Time, 120 flows, max_p =0.5.

5.6.2.3 Fairness

By restricting the robust flows with an aggressive max_p and encouraging the fragile flows with a conservative $\boldsymbol{max_p}$**,** AECN lets the fragile flow group get a higher share of the bandwidth at the bottleneck link than ECN**.** On the other hand AECN reduces the bandwidth share of robust flow group. Accordingly, Jain's fairness index of AECN (See Figure 5.29 and 5.30) increases in this way. As shown in both figures, within the unlockout range, AECN has higher than 10% improvement on Jain's fairness index. Meanwhile, observing the goodput and throughput distribution among each flow group in Figure 5.5 and Figures 5.20-23, we've already found that AECN has better visual max-min fairness than ECN since the gap between the robust flow group and the fragile flow group shrink for AECN, that is, the share of goodput or throughput for the fragile flow group increases while the share for the robust flow group goes down.

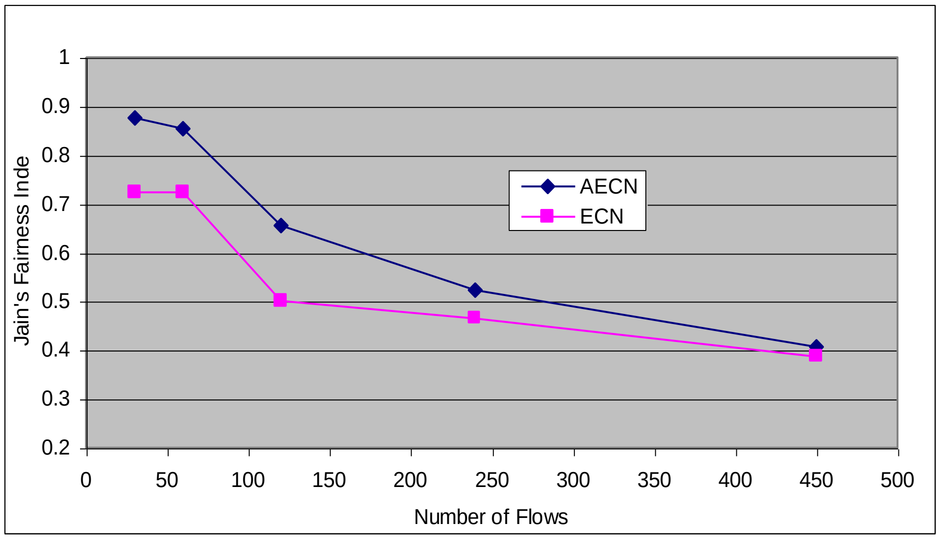

Figure 5.29 Jain’s Fairness Index with the number of flows, base-*max* $_p$ = 0.5

.

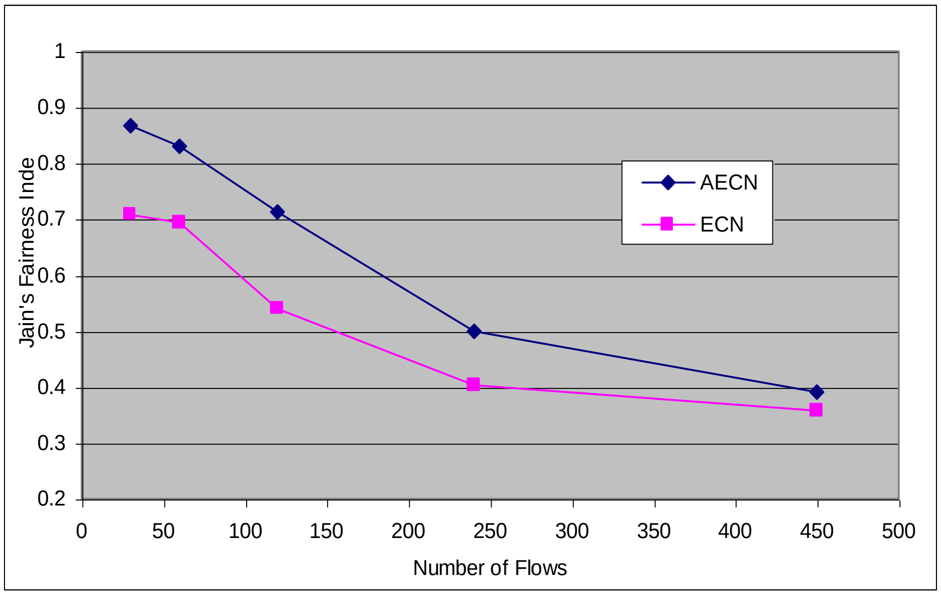

Figure 5.30 Jain’s Fairness Index with the number of flows, base-*max* $_p$ = 0.8.

5.6.2.4 Delay

Figure 5.31 presents the one-way delay distribution with the number of flows. As shown in this figure, increasing the number of flows causes the increment of delay, which is mainly due to higher queue delay in the router. This figure also shows that when the number of flows is low, like below 60 flows, AECN has lower delay than standard ECN since AECN uses mark-front strategy. But when the number of flows is really high and the congestion is heavy, AECN may have slightly higher queue delay.

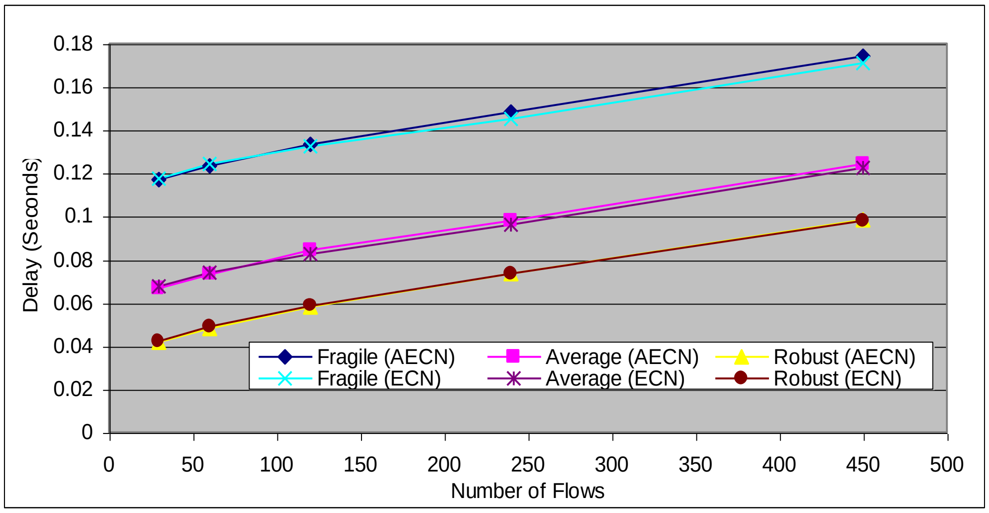

Figure 5.31 One-way delay with the number of flows, base-$max_p = 0.5$.

5.7 AECN Refinements

The section presents the methods taken to further refine AECN to keep better goodput and fairness when the number of flows is high, and the simulation results with the refined AECN. As shown in Figures 5.20 and 5.21, the robust flow group still gets a little higher share of the aggregate goodput than the other two groups when α and β both are equal to 2.5. To make it fairer to each flow group, we investigate the following three methods as AECN refinements. To distinguish with the version of AECN described in the previous section, the following versions of AECN are identified with AECN2, AECN3 and AECN4 respectively.

5.7.1 Population-balanced AECN (AECN2)

This method is to use the number of packets in each flow queue, that is, the current length of each flow queue, to decide the value of α and β. The basic idea of this method is that AECN2 supposes that robust flow group always occupies the most room in the router queue while fragile flow group gets the least, so AECN2 can use the ratio between the length of a flow queue and that of the router queue to decide different maximum marking probability. In this way, the robust flow group will get the most marks and the fragile flow group will get the least.

Algorithm 5.2 lists the basic algorithm of this population-balanced method in AECN2 to calculate α and β to update max_p when the average queue size is between max_{th} *and* min_{th}.

```
avg_fq_len = the length of the average flow queue;
if (avg_fq_len==0)
        avg_fq_len = the length of the router queue;
if ( status == FRAGILE_FLOW_QUEUE ) {
        $\alpha$ = avg_fq_len / the length of the fragile flow queue;
        $max_p$ = base_$max_p$ / $\alpha$;
} else if ( status == AVERAGE_FLOW_QUEUE ) {
        $max_p$ = base_$max_p$ ;
} else if ( status == ROBUST_FLOW_QUEUE ) {
        $\beta$ = the length of the robust flow queue / avg_fq_len;
        $max_p$ = min { 1, base_$max_p$ * $\beta$ };
}
```

Algorithm 5.2 *the basic algorithm of* AECN2

Based on this algorithm and implemented in ***ns-2***, more simulations were run to check the performance of AECN2 with the same simulation topology and configuration settings as shown in Figure 5.2.

Figures 5.32 and 5.33 show the simulation results on goodput and fairness with different **base_**max_p. Compared these results with that in Figure 5.29, Jain's fairness index in Figure 5.32 is always lower than that at 60 flows in Figure 5.29. Figure 5.33 further demonstrates that the population-balanced method among the three flow queues still helps robust flow group, which gets the most percentage of goodput shown in the figure. At this point, AECN2 seems to not win AECN with different value of **base-**max_p. More efforts need to be put into further investigating and refining this method[1].

[1] Due to the time limit, AECN2 will be further investigated and refined in the future work.

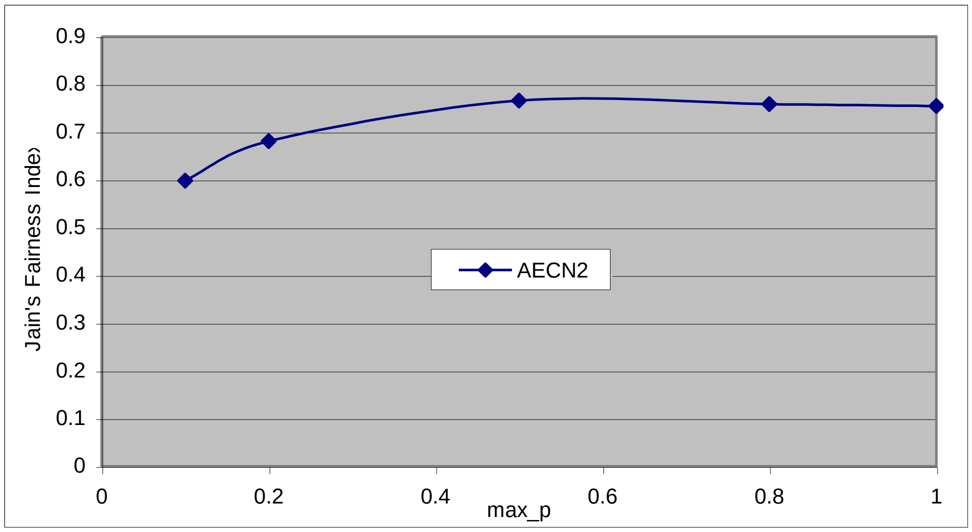

Figure 5.32 Jain's Fairness Index with different base-max_p .

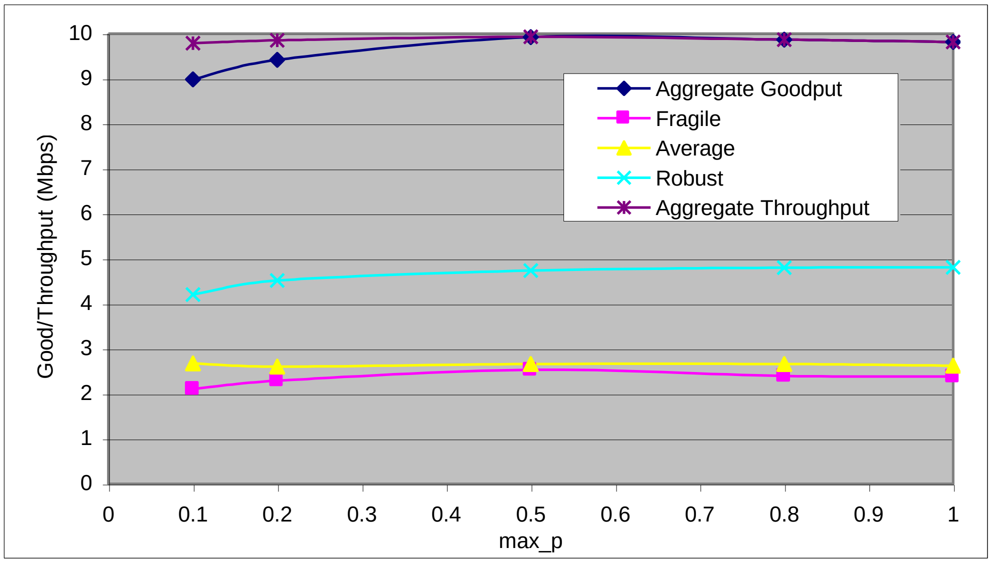

Figure 5.33 Jain's Fairness Index with different base-max_p .

5.7.2. Adjusting Min$_{th}$ (AECN3)

To avoid that the robust flows get too high a percentage of the router queue room, AECN3 tries to to adjust min_{th} for the robust flow group. By observing the change of min_{th} in Figure 5.34, it's not difficult to find that decreasing min_{th} will cause marking earlier and the marking probability will be more aggressive since the value of each updated marking probability is proportional to the average queue size which is mainly decided by min_{th} *and* max_{th} once $\boldsymbol{max}_p$ and w_q both are fixed.

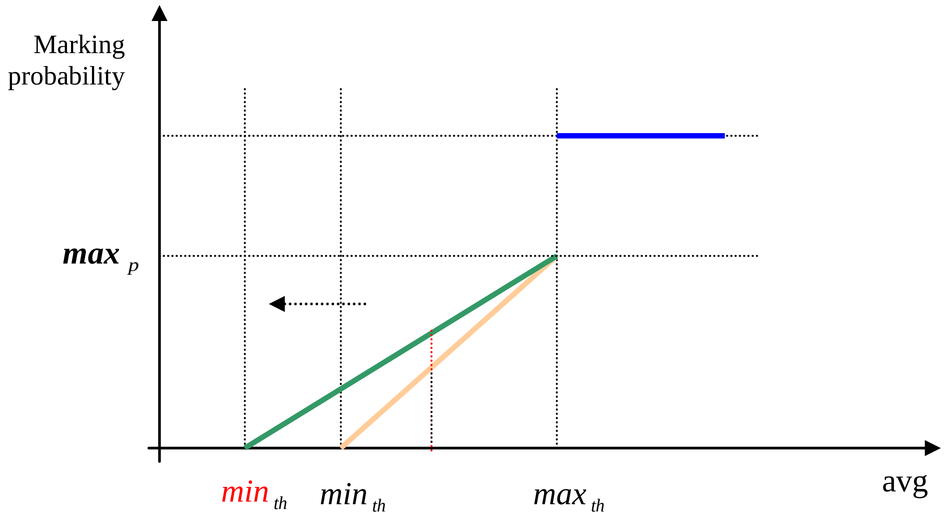

Figure 5.34, the relationship between marking probability and min_{th}

Figures 5.35 through 5.40 show the performance of AECN3 with different min_{th} for robust flow group. Figures 5.35 and 5.37 present Jain's fairness index with different min_{th} for robust flow group. As shown in Figure 5.35 Jain's fairness index for AECN3 with 60 flows tends to increase when min_{th} increases, while Jain's fairness index for AECN3 with 120 flows tends to decrease when min_{th} increases. This result shows the adjustment of min_{th} can be helpful to improve the performance of AECN3 when the number of flow changes. When the number of flows is changed from 60 flows to 120 flows, the congestion will become heavier, in which case in order to get higher performance of AECN3, the robust flow group can use a relatively smaller min_{th} .

Figures 5.36 and 5.38 present the goodput distribution among the three flow groups. In Figure 5.36, increasing min_{th} makes the goodput distribution among the three flow groups come closer, especially at point of min_{th} = 9 packets, where the visual max-min fairness is also the best for 60 flows. Figure 5.38 shows the goodput distribution with 120 flows. The aggregate goodput and visual max-min fairness tends to go down a little when min_{th} increases since the goodput of the robust flow group goes up while the other two groups decreases, as well. The visual max-min fairness is the best when min_{th} lies between points of 2 packets and 3 packets.

Figures 5.39 and 5.40 present the throughput distribution among the three flow groups. Both figures show that the throughput of AECN3 in this case keeps high.

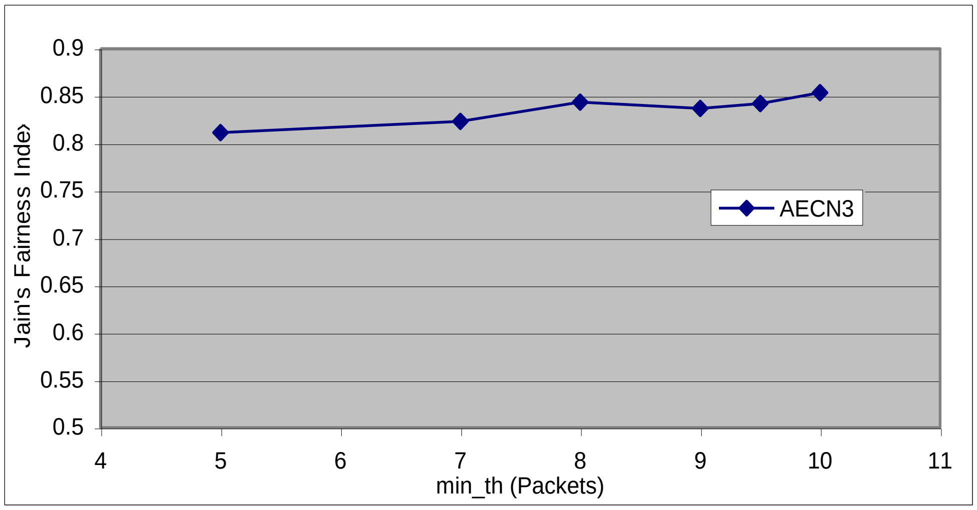

Figure 5.35 The Jain's fairness index with varied min_{th} for robust flows

60flows, **base_*max*** $_p$ = 0.5**.**

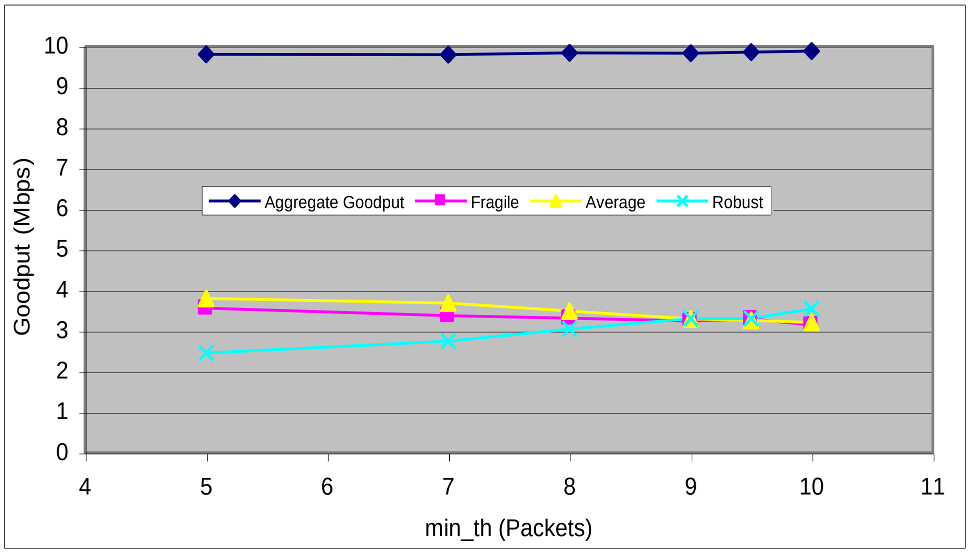

Figure 5.36 Goodput with varied min_{th} for robust flows

60flows, **base_*max*** $_p$ = 0.5**.**

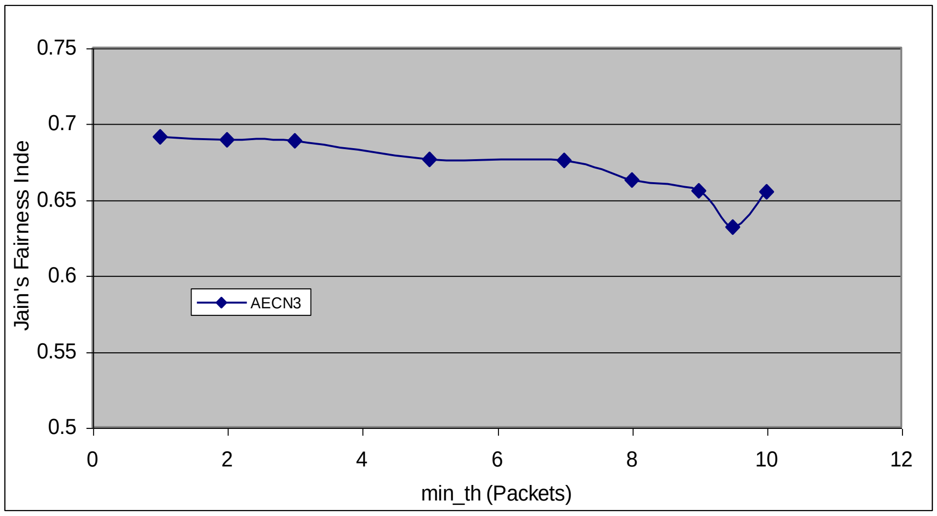

Figure 5.37 The Jain's fairness index with varied min_{th} for robust flows, 120flows, **base_*max*** $_p$ = ***0.5.***

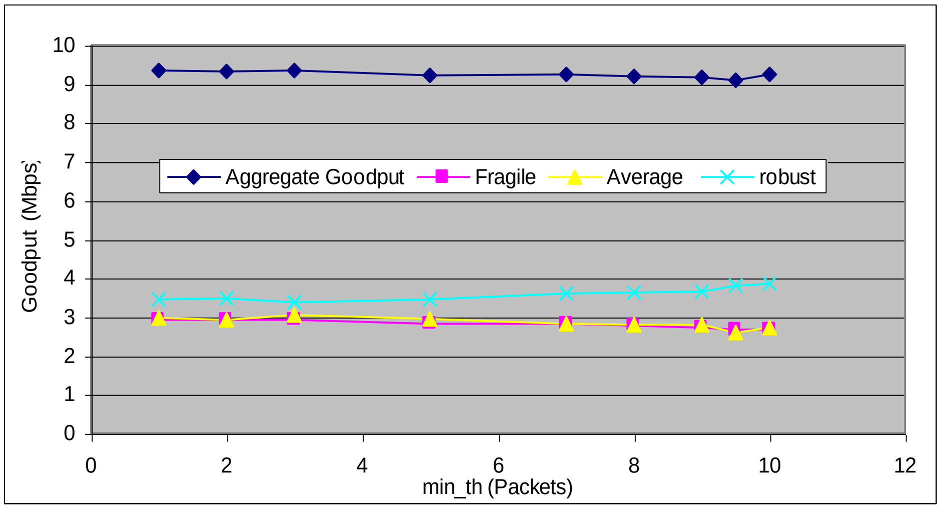

Figure 5.38 Goodput with varied min_{th} for robust flows 120flows, **base_*max*** $_p$ = 0.5**.**

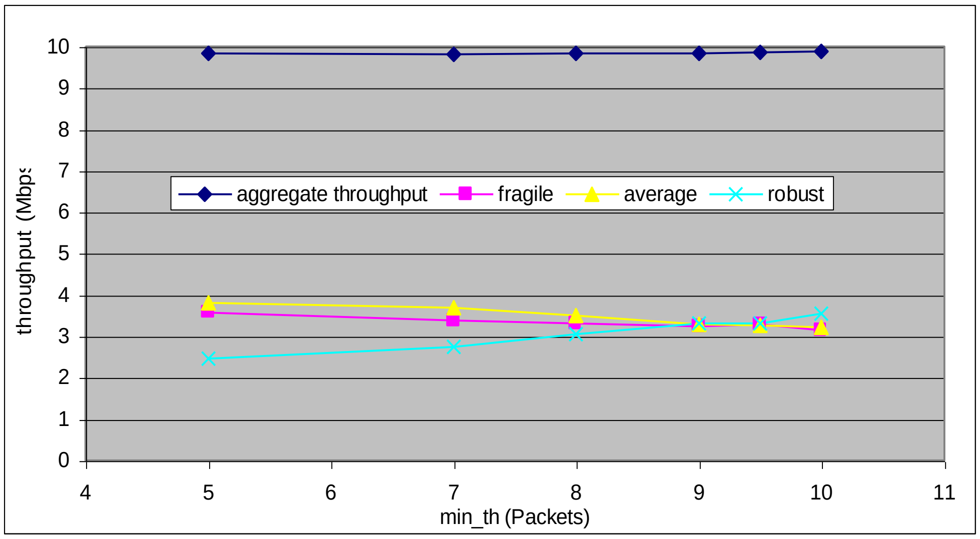

Figure 5.39 Throughput with varied min_{th} for robust flows

60flows, **base_*max*** $_p$ = 0.5**.**

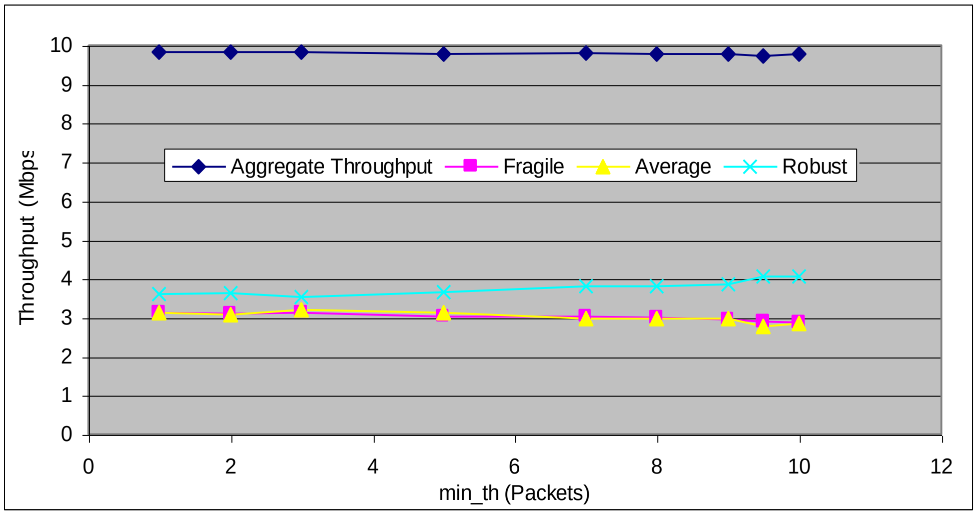

Figure 5.40 Throughput with varied min_{th} for robust flows

120flows, **base_*max*** $_p$ = 0.5**.**

5.7.3 Adjusting max$_{th}$ (AECN4)

Previous results indicate that when there are many flows aggregate goodput may be limited by max_{th}. From this viewpoint, it's expected that adjusting max_{th} may help AECN get better goodput. Therefore, the version of AECN adjusting max_{th} (AECN4) is investigated in this section. By increasing max_{th} with a fixed min_{th}, AECN4 will have more room in the router queue. Thus the chance that the average queue size hits max_{th} will decrease, and more packets will have higher chance to enter the router queue, instead of being dropped.

The parameter settings for AECN4 include:

1. **base_*max*** $_p$ is equal to 0.5,
2. min_{th} =9 Packets for robust flows when the number of flows is 60,
3. min_{th} =3 Packets for robust flows when the number of flows is 120,
4. min_{th} =10 Packets for average and fragile flows**,**
5. **α = β = 2.5.**

Figures 5.41 through 5.44 present the simulation results of AECN4 with the above parameter configuration. As shown in Figure 5.41, AECN4 tends to get higher Jain's fairness index by increasing max_{th}. Meanwhile, the visual max-min fairness (See Figures 5.42 and 5.43) is high and keeps stable when max_{th} increases. Figurse 5.42 and 5.43 further show that AECN4 gets better performance on goodput when max_{th} increases. In Figure 5.42, AECN4 with 60 flows gets the goodput almost close to 10 Mbps while max_{th} is above 40 packets. In Figure 5.43, AECN4 with 120 flows gets almost perfect goodput when max_{th} is above 50 packets. This seems to confirm our expectation that goodput may be limited by the available resources. The selection of max_{th} will set a virtual[1] upper limit of the router queue available for each flow. Therefore, increasing max_{th} will provide more queue space for each flow in the router and allow more packets to come in. However, the improvement on fairness and goodput causes the increase of one-way delay. As shown in Figure 5.44, for 120 flows, when max_{th} increases from 30 packets to 50 packets, the goodput and Jain's fairness index both increase about 7%,

[1] The reason why calls it as virtual upper limit for max_{th} is to distinguish with the physical size limit of a router queue.

while the one-way delays of robust, average and fragile flows increase about 17%, 8% and 7%.

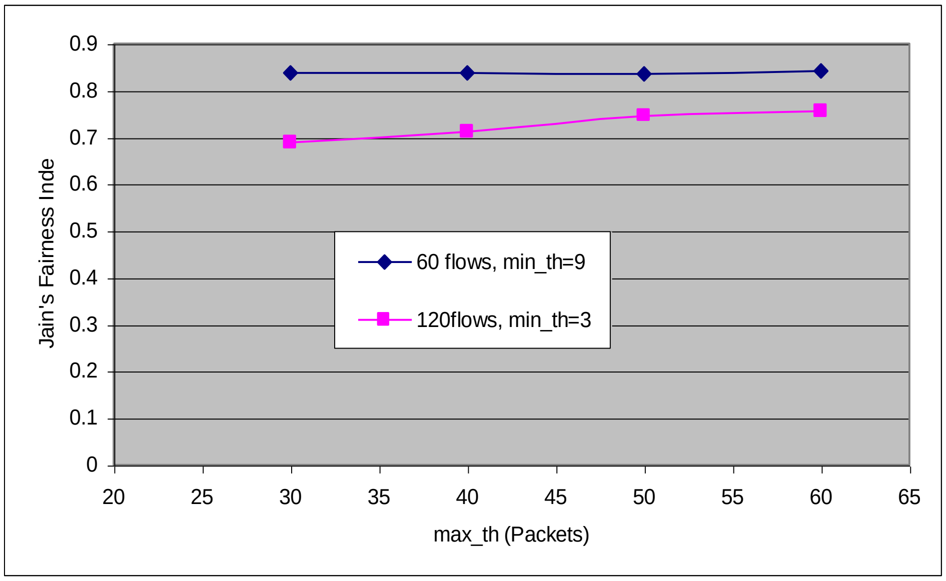

Figure 5.41 Jain's fairness index with different *max* $_{th}$

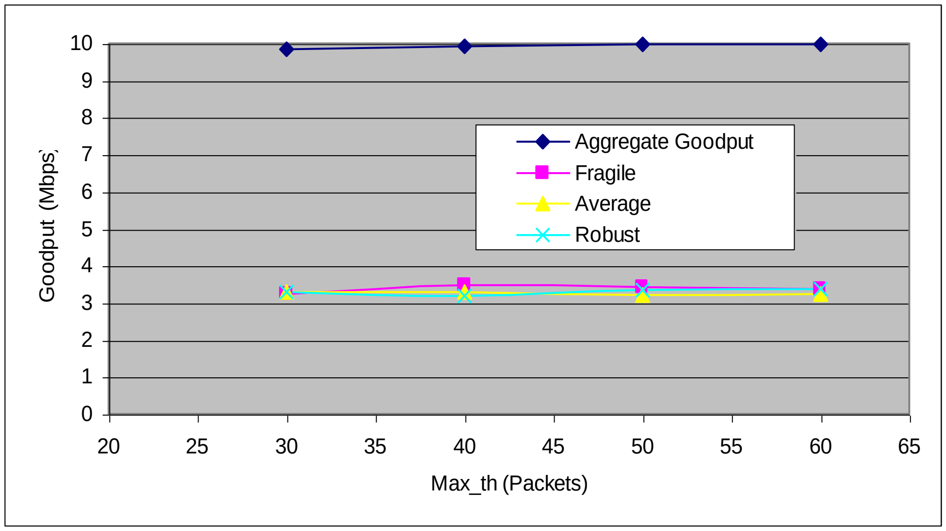

Figure 5.42. Goodput of AECN4 with different *max* $_{th}$, *60 flows*

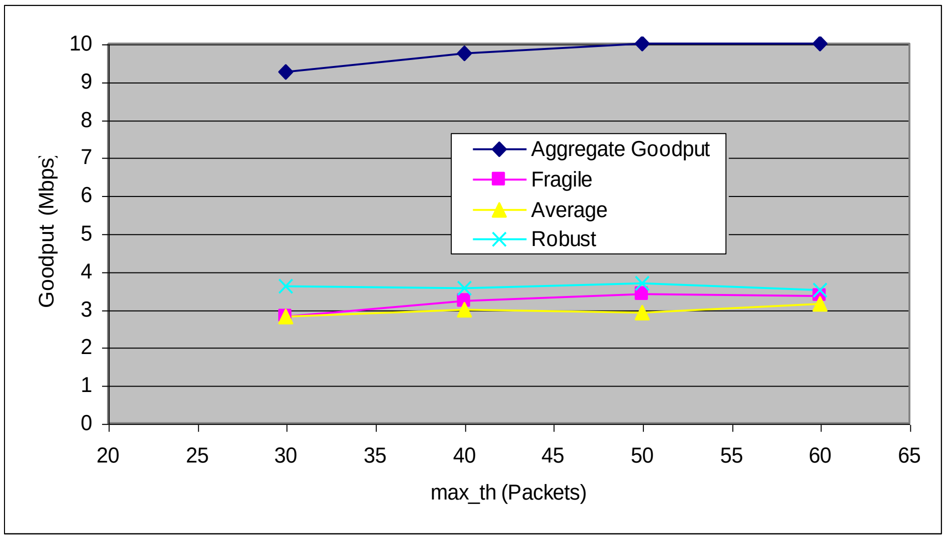

Figure 5.43. Goodput of AECN4 with varied max_{th}, *120 flows*

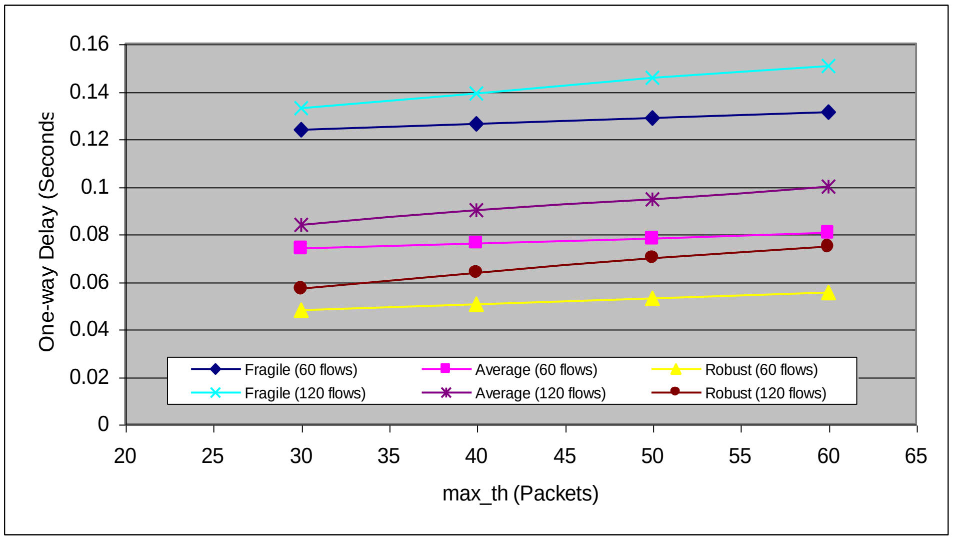

Figure 5.44. One-way delay of AECN4 with varied max_{th}

5.8 Conclusions

This chapter presents the basic algorithm of an adaptive version of ECN (AECN). AECN divides all flows competing a bottleneck into three flow groups, and deploys different max_p for each flow group so that a fragile flow can have higher chance to get a proper share of bandwidth when competing with a robust flow. Meanwhile AECN also adjusts

min_{th} for each robust flow group and max_{th} to get higher performance when the number of flows changes. Furthermore, AECN uses mark-front strategy, instead of mark-tail strategy in standard ECN, to mark an unmarked packet in the front of a corresponding flow queue so that the notification of congestion to a sender can be speeded up.

A number of simulations were run based on the implementation of AECN in ***ns-2***. The simulations show that AECN can treat each flow fairer than standard ECN with the two fairness measurements: Jain's fairness fairness and visual max-min fairness. Meanwhile, AECN has fewer packet drops than standard ECN and can further alleviate the occurrence of lockout phenomenon, and get higher goodput than standard ECN. AECN deploys the mark-front strategy, which can reduce the queue delay.

Based on these simulation results, three methods were investigated to further refine AECN on goodput and fairness. The first refinement is called populate-balanced AECN, which use the ration between the numbers of packets in each flow queue to deploy different max_{p} for each flow queue. The simulation results show that even though this method doesn't mprove the performance of AECN, especially on fairness. The visual max-min fairness isn't so good since the goodput gap between the fragile and robust flow group is still large.

The second refinement is to adjust min_{th} for robust flow group. By decreasing min_{th}, the marking probability will be more aggressive. The simulation results show that this method can further improve the performance of AECN on goodput and fairness. Especially when the number of flows is high, adjusting for the robust flow group can make an enough number of robust flows to slow down earlier so that the congestion at a bottleneck can be alleviated.

The last refinement is to adjust max_{th} for all flows. The previous results indicate that when there are many flows aggregate goodput may be limited by max_{th}. Therefore, it's expected that adjusting max_{th} may help AECN get better goodput. The simulation results confirm this expectation. Adjusting max_{th} can improve the performance of AECN on goodput and fairness. However, this improvement may by offset by high delay. The results show that when increase max_{th}, the one-way delay also increases.

Chapter 6 Conclusions and Future Work

6.1 Conclusions

This study first focused on evaluating the behavior and performance of ECN and RED in heterogeneous flows with a series of ***ns-2*** simulations. Generally ECN provides better goodput and is fairer than RED. However, in some case RED may has better throughput than ECN, especially when the number of flows and max_p are high. The results also show that the number of flows can affect the performance of both mechanisms. However, ECN with an aggressive max_p setting provides significantly higher goodput when there are a large number of heterogeneous flows. ECN also has a higher Jain's fairness Index and visual max-min fairness in the range of flows just below where flow lockout phenomena occur.

In the simulations studied, neither RED nor ECN mechanism is fair to fragile and average flows. These results suggest that if congestion control is to handle Web traffic consisting of thousands of concurrent flows with some degree of fairness then further enhancements to ECN are needed. Based on these results, we propose an adaptive version of ECN (AECN), which can adjust max_p based on the round-trip time of a flow.

AECN divides all flows competing a bottleneck into three flow groups, and deploys different max_p for each flow group so that a fragile flow can have higher chance to get a proper share of bandwidth when competing with a robust flow. Meanwhile, AECN also adjusts min_{th} for each robust flow group and max_{th} to get higher performance when the number of flows changes. Furthermore, AECN uses the mark-front strategy, instead of mark-tail strategy in standard ECN, to mark the first unmarked packet of a corresponding flow group in the router queue so that the notification of congestion can be speeded up to a sender.

A number of simulations were run based on the implemented of AECN in ***ns-2***. The simulations show that AECN can treat each flow fairer than standard ECN with the two fairness measurements: Jain's fairness fairness and visual max-min fairness. Meanwhile, AECN has fewer packet drops than standard ECN and can further alleviate the occurrence of lockout phenomenon, and get higher goodput than standard ECN. AECN, deploying the mark-front strategy, can speed up the transmission of congestion notification to the sender and reduce the queue delay.

Based on these simulation results, three methods were investigate to further refine AECN on goodput and fairness. The first method is called populate-balanced AECN, which use the ratio between the numbers of packets in each flow queue to deploy different max_p for each flow queue. The simulation results show that this method doesn't improve the performance of AECN.

The second method is to adjust min_{th} for robust flow group. By decreasing min_{th}, the marking probability will be more aggressive. The simulation results show that this method can further improve the performance of AECN on goodput and fairness. Especially when the number of flows is high, adjusting for the robust flow group can cause the robust flows to slow down earlier so that the congestion at a bottleneck can be alleviated, which improves the goodput.

The last method is to adjust max_{th} for all flows. The results indicate that when there are many flows aggregate goodput may be limited by max_{th}. Therefore, it's expected that adjusting max_{th} may help AECN get better goodput. The simulation results confirm this expectation. Adjusting max_{th} can improve the performance of AECN on goodput and fairness. However, this improvement may by offset by high delay. The results show that when increase max_{th}, the one-way delay also increases.

In summary, AECN, combined with the second and third methods, can effectively improve the performance itself on goodput and fairness. Especially, when the number of competing flows is high and the congestion at a bottleneck router is heavy, AECN can keep getting better performance than standard by adjusting max_{th} and min_{th}.

6.2 Future Work

The basic algorithm of AECN has been demonstrated in the simulation topology with three different flow groups. Each flow group has the same number of flows. The following is a list of the possible directions of future works:

(1) To further investigate the performance of AECN in more complicated network configurations, such as the number of flows in each flow group can be different, and each flow in a specific flow group can have different round-trip time.

(2) Investigate the performance of AECN with mixed types of traffic, such as Pareto traffic, non-ECN capable and ECN capable traffic, and unresponsive flows like UDP flows.

(3) Further refine AECN deploying the population-balanced method. Even though the simulation results shown in this study with the population-balanced method is not so good, it’s believed that further refinement can be investigated to improve the performance in the network with heterogeneous flows.

(4) Adjust the values of α and β to deploy a more flexible marking probability for each flow group to achieve the optimal performance for a different network scenario, which may have a minimal different round-trip time for each flow.

(5) Optimize the performance of AECN on power [JAI91]. AECN can get high goodput, but it may also have a high delay. One way to optimize AECN is to measure the performance of AECN on power.

(6) To make AECN be a robust algorithm, one of future work is to refine AECN with a more robust model or formula even though it’s impossible to propose a general algorithm for AECN to handle all kinds of network topologies. One concern about the future work is to make AECN more intelligent to accurately estimate the number of flows and adjust ECN parameters adaptively.

(7). Evaluate the performance and behavior of ECN and AECN in a network with multiple congested routers, and look at whether one bit for a congestion experienced packet is enough for ECN.

Appendix A

Code Added for Calculating Round-trip Time

In tcp-reno.cc file,

```
void RenoTcpAgent::recv(Packet *pkt, Handler*)
{
	/* added by Zici Zheng for RTT */
	double currentTime = Scheduler::instance().clock() - firstsent_;
	hdr_flags *flagh = hdr_flags::access(pkt);
	//

	   hdr_tcp *tcph = hdr_tcp::access(pkt);
#ifdef notdef
	   if (pkt->type_ != PT_ACK) {
		      fprintf(stderr,
			         "ns: confiuration error: tcp received non-ack\n");
		      exit(1);
	   }
#endif
	   ++nackpack_;
	/* Added by Zici Zheng for RTT. */
	if (firstrecv_ < 0) { //
		firstrecv_ = currentTime;
		v_baseRTT_ = v_rtt_ = firstrecv_;
		v_sa_ = v_rtt_ * 8.;
		v_sd_ = v_rtt_;
	}
	//

	   ts_peer_ = tcph->ts();

	   if (hdr_flags::access(pkt)->ecnecho() && ecn_)
		      ecn(tcph->seqno());
	   recv_helper(pkt);
	   if (tcph->seqno() > last_ack_) {
		      dupwnd_ = 0;
		      recv_newack_helper(pkt);
		      if (last_ack_ == 0 && delay_growth_) {
			         cwnd_ = initial_window();
		      }
```

```
/* Added by Zici Zheng */
//update path fine-grained rtt and baseRTT
int oldack = last_ack_;
if (tcph->seqno() >= v_begseq_) {
        double rtt;
        if (v_cntRTT_ > 0)
                rtt = v_sumRTT_ / v_cntRTT_;
        else
                rtt = currentTime - v_begtime_;
        v_sumRTT_ = 0.0;
        v_cntRTT_ = 0;

        // calculate # of packets in transit
        int rttLen = t_seqno_ - v_begseq_;

        if (rtt > 0) {
                if (rtt < v_baseRTT_ || rttLen <= 1)
                        v_baseRTT_ = rtt;
        }

        //tag the next packet
        v_begseq_ = t_seqno_;
        v_begtime_ = currentTime;
}

// reset v_sendtime for acked pkts
double sendTime = v_sendtime_[tcph->seqno()%v_maxwnd_];
int transmits = v_transmits_[tcph->seqno()%v_maxwnd_];
int range = tcph->seqno() - oldack;
for (int k=((oldack+1)%v_maxwnd_); \
         k<=(tcph->seqno()%v_maxwnd_) && range>0; \
         k=((++k) % v_maxwnd_), range--) {
        v_sendtime_[k] = -1.0;
        v_transmits_[k] = 0;
}
if ((sendTime != 0.) && (transmits==1)) {
        double rtt, n;
        rtt = currentTime - sendTime;
        v_sumRTT_ += rtt;
        ++v_cntRTT_;
        if (rtt>0) {
                v_rtt_ = rtt;
                if (v_rtt_ < v_baseRTT_)
                        v_baseRTT_ = v_rtt_;
                n = v_rtt_ - v_sa_ / 8;
```

```
                        v_sa_ += n;
                        n = n<0? -n : n;
                        n -= v_sa_ / 4;
                        v_sd_ += n;
                    }
                }
            // end of Added by Zici Zheng

        } else if (tcph->seqno() == last_ack_) {
                if (hdr_flags::access(pkt)->eln_ && eln_) {
                        tcp_eln(pkt);
                        return;
                }
                if (++dupacks_ == numdupacks_) {
                        dupack_action();
                        dupwnd_ = numdupacks_;
                } else if (dupacks_ > numdupacks_) {
                        ++dupwnd_;    // fast recovery
                } else if (dupacks_ < numdupacks_ && singledup_ ) {
                        send_one();
                }
        }
        Packet::free(pkt);
#ifdef notyet
        if (trace_)
                plot();
#endif

        /*
         * Try to send more data
         */

        if (dupacks_ == 0 || dupacks_ > numdupacks_ - 1)
                send_much(0, 0, maxburst_);
}


void RenoTcpAgent::output(int seqno, int reason)
{
    int force_set_rtx_timer = 0;
    Packet* p = allocpkt();
    hdr_tcp *tcph = hdr_tcp::access(p);
    hdr_flags* hf = hdr_flags::access(p);
```

```
        tcph->seqno() = seqno;

        tcph->ts() = Scheduler::instance().clock();

        tcph->ts_echo() = ts_peer_;

        tcph->reason() = reason;

        /* Added by Zici Zheng */

        tcph->r_rtt() = v_baseRTT_;

        //


        if (ecn_) {
                hf->ect() = 1;  // ECN-capable transport
        }
        if (cong_action_) {
                hf->cong_action() = TRUE;  // Congestion action.
                cong_action_ = FALSE;
        }
        /* Check if this is the initial SYN packet. */
        if (seqno == 0) {
                /* added by Zici Zheng */
                v_maxwnd_ = int(wnd_);
                if (v_sendtime_)
                        delete []v_sendtime_;
                if (v_transmits_)
                        delete []v_transmits_;
                v_sendtime_ = new double[v_maxwnd_];
                v_transmits_ = new int[v_maxwnd_];
                for (int i=0; i<v_maxwnd_; i++) {
                        v_sendtime_[i] = -1;
                        v_transmits_[i] = 0;
                }

                int index = seqno % v_maxwnd_;
                v_sendtime_[index] = Scheduler::instance().clock() - firstsent_;
                ++v_transmits_[index];
                //

                if (syn_) {
                        hdr_cmn::access(p)->size() = tcpip_base_hdr_size_;
                }
                if (ecn_) {
                        hf->ecnecho() = 1;
//                      hf->cong_action() = 1;
                        hf->ect() = 0;
                }
        }
        int bytes = hdr_cmn::access(p)->size();

        /* if no outstanding data, be sure to set rtx timer again */
        if (highest_ack_ == maxseq_)
                force_set_rtx_timer = 1;
        /* call helper function to fill in additional fields */
        output_helper(p);

        ++ndatapack_;
        ndatabytes_ += bytes;
        send(p, 0);
        if (seqno == curseq_ && seqno > maxseq_)
                idle();  // Tell application I have sent everything so far
        if (seqno > maxseq_) {
                maxseq_ = seqno;
                if (!rtt_active_) {
                        rtt_active_ = 1;
                        if (seqno > rtt_seq_) {
                                rtt_seq_ = seqno;
                                rtt_ts_ = Scheduler::instance().clock();
                        }

                }
        } else {
                ++nrexmitpack_;
                nrexmitbytes_ += bytes;
        }
        if (!(rtx_timer_.status() == TIMER_PENDING) || force_set_rtx_timer)
                /* No timer pending.  Schedule one. */
                set_rtx_timer();
}
```

Appendix B

Code Modified for Implementing Three Flow Queues

In red.h header file:

```
……

struct fq_node_ {
        Packet*         pkt;
        struct fq_node_* next;
};
typedef struct fq_node_ fq_node;

#define AVG_RTT                 100
#define ROB_RTT                 50
#define FRG_RTT                 200

#define FRAGILE_FLOW    0
#define AVERAGE_FLOW    1
#define ROBUST_FLOW     2


class fq_queue {
private:
        fq_node* head;
        fq_node* tail;
        int      size;

public:
        fq_queue(): head(NULL), tail(NULL), size(0) {}
        ~fq_queue() {}

        void add(fq_node* newpkt) {
                if (!tail)
                                head = tail = newpkt;
                else {
                        tail->next = newpkt;
                        tail = newpkt;
                }
                tail->next = NULL;
                ++size;
        }
```

```
fq_node* popSel(Packet* pkt) {
        fq_node* p=head;
        fq_node* tmp;

        if (!head)
              return NULL;
        if (p->pkt == pkt) {
              if (tail == p)
                    tail=NULL;
              head = head->next;
              --size;
              return p;
        }

        while (p->next) {
              if (p->next->pkt == pkt) {
                    if (p->next == tail)
                          tail = p;
                    tmp = p->next;
                    p->next = tmp->next;
                    --size;
                    return tmp;
              }
              p = p->next;
        }
        return NULL;
}

 fq_node* remove() {
    if (!head) return NULL;
    fq_node* p = head;
    head = p->next;
        if (p == tail) head = tail = NULL;
    --size;
    return p;
 }

 Packet* getFrontUnmarkedPkt() {
        fq_node* p = head;

        if (!p) return NULL;

        hdr_flags* hf = hdr_flags::access(p->pkt);
        if ( (hf->ce() == 0)&&hf->ect() )
              return p->pkt;
```

```
                int i=1;
                while (p->next) {
                        i++;
                        p=p->next;
                        hf = hdr_flags::access(p->pkt);
                        if ( (hf->ce() == 0)&&hf->ect() ) {
                                return p->pkt;
                        }
                }
                //otherwise, something wrong.
        }

        Packet* front() {
                if (head)
                return head->pkt;
                return NULL;
        }

        int length() {
             return size;
        }

};
//
……
```

In red.cc file

```
/*
 * should the packet be dropped/marked due to a probabilistic drop?
 */
int
REDQueue::drop_early(Packet* pkt)
{
        hdr_cmn* ch = hdr_cmn::access(pkt);
        double my_maxp_inv; //  (= 1/my_maxp)

        double alpha, belta;
        double rob_v_a,rob_v_b;

        //int avg_fq_len = avg_fq.length();
        //if (avg_fq_len == 0) {
        //      avg_fq_len = q_->length();
        //}
        switch (status) {
            case FRAGILE_FLOW :
                    alpha = 2.5; //frg_fq.length() / avg_fq_len; //q_->length();
                    my_maxp_inv = edp_.max_p_inv * alpha;
                    break;
            case AVERAGE_FLOW :
                    my_maxp_inv = edp_.max_p_inv;
                    break;
            case ROBUST_FLOW :
                    belta = 2.5; //rob_fq.length() / avg_fq_len;
                    my_maxp_inv = edp_.max_p_inv /  belta;
       break;
    }
    if (my_maxp_inv < 1 )
            my_maxp_inv =1;

    if (status == ROBUST_FLOW) {
            rob_v_a = 1 / (edp_.th_max - edp_.th_min + r_min_th);
            rob_v_b  =  -  (edp_.th_min  –  r_min_th)  /  (edp_.th_max  -  edp_.th_min  +
    r_min_th);
            edv_.v_prob1 = calculate_p(edv_.v_ave, edp_.th_max, edp_.gentle,
                    rob_v_a, rob_v_b, edv_.v_c, edv_.v_d, my_maxp_inv);
            edv_.v_prob = modify_p(edv_.v_prob1, edv_.count, edv_.count_bytes,
                    edp_.bytes, edp_.mean_pktsize, edp_.wait, ch->size());
    } else {
            edv_.v_prob1 = calculate_p(edv_.v_ave, edp_.th_max, edp_.gentle,
                    edv_.v_a, edv_.v_b, edv_.v_c, edv_.v_d, my_maxp_inv);
            edv_.v_prob = modify_p(edv_.v_prob1, edv_.count, edv_.count_bytes,
                     edp_.bytes, edp_.mean_pktsize, edp_.wait, ch->size());
    }

    hdr_flags* hf = hdr_flags::access(pickPacketForECN(pkt));

    double u = Random::uniform();
    if ( u <= my_maxp ) {
      edv_.count = 0;
      edv_.count_bytes = 0;
      if (edp_.setbit /*&& hf->ect()*/ && (edv_.v_ave < edp_.th_max)) {
            hf->ce() = 1;
            return (0);
       } else
            return (1);
    }

    return (0);                              // no DROP/mark
}
```

```
Packet*
REDQueue::pickPacketForECN(Packet* pkt)
{
        /* added by Zici Zheng for AECN */
        switch (status) {
          case FRAGILE_FLOW :
                return frg_fq.getFrontUnmarkedPkt();
          case ROBUST_FLOW :
                return rob_fq.getFrontUnmarkedPkt();
          case AVERAGE_FLOW :
                return avg_fq.getFrontUnmarkedPkt();
        }
        return pkt;
}

void REDQueue::enque(Packet* pkt)
{

……

        /*
         * DROP LOGIC:
         *      q = current q size, ~q = averaged q size
         *      1> if ~q > maxthresh, this is a FORCED drop
         *      2> if minthresh < ~q < maxthresh, this may be an UNFORCED drop
         *      3> if (q+1) > hard q limit, this is a FORCED drop
         */
      register double qavg = edv_.v_ave;
      int droptype = DTYPE_NONE;
      int qlen = qib_ ? bcount_ : q_->length();
      int qlim = qib_ ? (qlim_ * edp_.mean_pktsize) : qlim_;

      curq_ = qlen;   // helps to trace queue during arrival, if enabled

      /* added by Zici Zheng for AECN */
        if ( edp_.setbit && (qavg < edp_.th_max) && (qlen < qlim)) {
           hdr_tcp * tcph = hdr_tcp::access(pkt);
           double f_rtt = tcph->r_rtt() * 1000;
                fq_node* new_node = new fq_node();
                new_node->pkt = pkt;
                new_node->next = NULL;
           if ( (f_rtt >= (FRG_RTT-50)) && (f_rtt < (FRG_RTT+300)) ) {
                status = FRAGILE_FLOW;
                frg_fq.add(new_node);
                } else if ( (f_rtt < (ROB_RTT+25)) && (f_rtt >= 0.5) ) {
                status = ROBUST_FLOW;
```

```
                        rob_fq.add(new_node);
                } else {
                        status = AVERAGE_FLOW;
                        avg_fq.add(new_node);
                }
        }
        //
//printf("enque--%s,  R:%d,  A:%d,  F:%d,  Q:%d\n",  this->name(),  rob_fq.length(),
avg_fq.length(), frg_fq.length(), q_->length());

            double min_th = edp_.th_min;
            if (status == ROBUST_FLOW)
                        min_th = edp_.th_min – r_min_th;

            if (qavg >= min_th && qlen > 1) {
……
            }
            if (qlen >= qlim) {
                        // see if we've exceeded the queue size
                        droptype = DTYPE_FORCED;
            }
            /* pick packet for ECN, which is dropping in this case */
            if (droptype == DTYPE_UNFORCED) {
                Packet *pkt_to_drop = pickPacketForECN(pkt);

                 // this should not happen to AECN.
                fq_node *nd = NULL;
                switch (status) {
                  case FRAGILE_FLOW :
                        nd = frg_fq.popSel(pkt_to_drop);
                        break;
                  case ROBUST_FLOW :
                        nd = rob_fq.popSel(pkt_to_drop);
                        break;
                  case AVERAGE_FLOW :
                        nd = avg_fq.popSel(pkt_to_drop);
                        break;
                  default :
                        ;
                }


                if (pkt_to_drop != pkt) {
                        q_->enque(pkt);
                        bcount_ += ch->size();
                        q_->remove(pkt_to_drop);
                        bcount_ -= hdr_cmn::access(pkt_to_drop)->size();
                        pkt = pkt_to_drop; /* XXX okay because pkt is not needed
 anymore */
                }

                if (nd)
                        delete nd;
…

            return;
}

/*
 * Return the next packet in the queue for transmission.
 */
Packet* REDQueue::deque()
{
            Packet *p;
            fq_node *nd = NULL;

            p = q_->deque();

            if (p != 0) {
                /* Added by Zici Zheng for ECN */
                        if ( rob_fq.front() == p) {
                        nd = rob_fq.remove();
                } else if ( avg_fq.front() == p) {
                        nd = avg_fq.remove();
                } else if ( frg_fq.front() == p) {
                        nd = frg_fq.remove();
                }
                if (nd)
                    delete nd;

                        idle_ = 0;
                        bcount_ -= hdr_cmn::access(p)->size();
            } else {
                        idle_ = 1;
                        ……
            }
            return (p);
}
```

Bibliography

[AHM99]. Ahmed, U., and Salim, J., “Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks”, IETF *Internet Draft, RFC288*4, December 1999.

[AHN95]. Ahn, J, et all, “Evaluation of TCP Vegas: Emulation and Experiment”, ACM SIGCOMM, pp. 185-195, 1995.

[BAL98]. Balakrishnan, H., et all. “TCP Behavior of Busy Internet Server: Analysis and Improvement”, *Proceedings of IEEE Infocom,* Mach 1998. http://www.cs.berkeley.edu/hari/papers/infocom98.ps.gz.

[BAG99]. Bagal, P., Kalyanaraman, S., and Packer, B., “Comparative study of RED, ECN and TCP Rate Control”, Technical Report, March 1999.

[BAJ99]. Bajaj, S., et al, “Improving Simulation for Network Research”, Technical Report 99-702, University of Southern California, March 1999.

[BRA94]. Brakmo, L., Malley, S., and Peterson, L., “ TCP Vegas: New Techniques for Congestion Detection and Avoidance”, ACM SIGCOMM, pp24-35, August 1994.

[BRA95]. Brakmo, L., and Peterson, L., “TCP Vegas: End to End Congestion Avoidance on a Global Internet”, IEEE Journal on Selected Areas in Communications, *vol.* 13, no. 8, October 1995.

[BRA98]. Braden, R., et all, “Recommendations on Queue Management and Congestion Avoidance in the Internet”, *RFC2309*, April 1998.

[BRU98]. Bruyeron, R., Hemon, B., and Zhang, L., “Experimentations with TCP Selective Acknowledgement”, Computer Communication Review, vol. 28, no. 2, pp. 54-77, April 1998.

[CHR00]. Christiansen, M., *et all*, “Tuning RED for Web Traffic”, SIGCOMM’00, pp. 139-150, August 2000.

[CNO00]. Cnodder, S., Pauwels, K., and Elloumi, O, “A Rate Based RED Mechanism”, the 10th International Workshop on Network and Operating System Support for Digital Audio and Video, June 2000.

[FEN97]. Feng, W., et all, “Techniques for Eliminating Packet Loss in Congested TCP/IP Networks”, U. Michigan *CSE-TR-349-97*, November 1997.

[FEN99a]. Feng, W., et all, “A Self-Configuring RED Gateway”, Proceedings of INFOCOM'99, 1999.

[FEN99b]. Feng, W., et all, "Blue: A New Class of Active Queue Management Algorithms", U. Michigan CSE-TR-387-99, April 1999

[FLO91]. Floyd, S., “Connections with Multiple Congested Gateways in Packet-Switched Networks Part1: One-way Traffic”. ACM *Computer Communication Review*, vol. 21, no. 5, pp. 30-47, October 1991.

[FLO93]. Floyd, S., and Jacobson, V., “Random Early Detection Gateways for Congestion Avoidance”, IEEE/ACM Transactions on Networking, vol.1, no. 4, pp. 397-413, August 1993.

[FLO94]. Floyd, S., “TCP and Explicit Congestion Notification”, ACM Computer Communication Review, vol. 24, no. 5, pp. 10-23, October 1994.

[FLO96]. Floyd, S., and Fall, K., "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", *Computer Communication Review*, vol. 26, no. 3, pp. 5-21, July 1996.

[FLO97]. Floyd, S., "RED: Discussions of Setting Parameters", http://www.aciri.org/floyd/REDparameters.txt, November 1997.

[FLO98a]. Floyd, S., "Implementing ECN in TCP", http://www.aciri.org/floyd/ECN-TCP.txt, January 1998.

[FLO98b]. Floyd, S., "RED with drop from front", email discussion on the end2end mailing list, ftp://ftp.ee.lbl.gov/email/sf.98mar11.txt, March 1998.

[FLO99b]. Floyd, S., and Fall, K., "Promoting the Use of End-to-End Congestion Control in the Internet", IEEE/ACM Transactions on Networking, vol. 7, no. 4, pp. 458-472, 1999.

[FLO99c]. Floyd, S., Black, D., and Ramakrishnan, K., "IPSec Interactions with ECN", *Internet Draft draft-ispec-ecn-00.txt*, URL http://www.ietf.cnri.va.us/internet-drafts/draft-ipsec-ecn-00.txt, December 1999.

[FLO00a]. Floyd, S., "Congestion Control Principles", IETF Internet Draft, http://www.aciri.org/floyd/papers/draft-floyd-cong-01.txt, January 2000

[FLO00b]. Floyd, S., Handley, M., and Padhye, J., "A Comparison of Equation-based and AIMD Congestion Control", *Preliminary version*, May 2000.

[FLO00c]. Floyd, S., Rammakrishnan, K., "TCP with ECN: The Treatment of Retransmitted Data Packets", IETF Draft, draft-ietf-tsvwg-tcp-ecn-00.txt, November 2000.

[GER99]. Gerla, M. et all, “Generalized Window Advertising for TCP Congestion Control”, Technical Report No. 990012, Computer Science Department, UCLA, January 1999.

[JAC88]. Jacobson, V., “Congestion Avoidance and Control”. *Proceedings of SIGCOMM ’88,* Palo Alto, CA, August 1988.

[JAC92]. Jacobson, V., Braden, R., and Borman, D., “TCP Extension for High Performance”, IETF, RFC1323, May 1992.

[JAI91]. Jain, R., “The Art of Computer Systems Performance Analysis”, John Wiley and Sons, *QA76.9.E94J32,* 1991.

[JEO00]. Jeonghoon, M., and Jean, W., “Fair End-to-End Window-based Congestion Control”, vol. 8, no. 5, pp. 556-567, 2000.

[KAR91]. Karn, P., and Partridge, C., "Improving Round-Trip-Time Estimates in Reliable Transport Protocol," *ACM Transaction on Computer Systems (TOCS),* vol. 9, no. 4, pp.364-373, November 1991.

[KNI01]. Kinicki, R., and Zheng, Z., “Performance Research of Explicit Congestion Notification (ECN) with Heterogeneous TCP Flows”, International Conference on Networking (ICN), Accepted, Colmar, France, July 2001.

[KUN00]. Kunniyur, S., and Srikant, R., “End-to-End Congestion Control Schemes: Utility Functions, Random Losses and ECN Marks”, IEEE INFOCOM, pp.1323-1332, 2000.

[LIN97]. Lin, D., and Morris, R., “Dynamics of Random Early Dectection”, ACM Computer Communication Review, vol. 27, no. 4, pp. 127-138, October 1997.

[LIU01]. Liu, C., and Jain, R., “Improving Explicit Congestion Notification with the Mark-Front Strategy”, Computer Networks, vol. 35, no. 2-3, pp. 185-201, January 2001.

[MAT97]. Mathis, M., Semke, J., and Mahdavi, J., “The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm”, ACM Computer Communication Review, vol. 27, no. 3, July 1997.

[MAY00]. Mayer, Alain, and Ofek, Yoram, “Local and Congestion-Driven Fairness Algorithm in Arbitrary Topology Networks”. IEEE/ACM Transactions on Networking, vol.8, no.3, pp. 362-372, June 2000.

[MOR97]. Morris, R., “TCP Behavior with Many Flows”, Proc of IEEE ICNP’97, October 1997.

[NS201]. http://www.isi.edu/nsnam/ns/doc/index.html.

[OTT99]. Ott, T., Lakshman, T., and Wong, L., “SRED: Stabilized RED”, Proceedings of IEEE INFOCOM’99, March 1999,

[PAD00]. Padhye, J., et al., “Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation”, IEEE/ACM Transactions on Networking, vol. 8, no. 2, pp. 133-145, April 2000.

[PAX97]. Paxson, V., “End-to-End Internet Packet Dynamics”, ACM SIGCOMM, pp. 139-176, 1997.

[PET00]. Peterson, L.L., and Davie, B.S., “Computer Networks, A Systems Approach”, 2nd Ed., Morgan Kaufmann, San Francisco, 2000.

[QIU99]. Qiu, L., Zhang, Y., and Keshav, S., “On Individual and Aggregate TCP Performance”, Proceeding of IEEE ICNP’99, 1999, [RAG99]. Raghavendra, A. and Kinicki, R., "A Simulation Performance Study of TCP Vegas and Random Early Detection", IEEE, 1999

[RAM99]. Ramakrishnan, K., and Floyd, S., "A Proposal to Add Explicit Congestion Notification (ECN) to IP", ftp://ftp.isi.edu/in-notes/rfc2481.txt, January 1999,

[RAM01]. Rammakrishnan, K., Floyd, S., and Black, D., "The Addition of Explicit Congestion Notification (ECN) to IP", IETF Draft, draft-ietf-tsvwg-ecn-02.txt, February 2001.

[SAV99]. Savage, S., et all, "TCP Congestion Control with a Misbehaving Receiver", ACM Computer Communication Review, vol. 29, no. 5, October 1999.

[STE94]. Stevens, W., "TCP/IP Illustrated, Volume 1: The Protocols", Addison-Wesley, 1994.

[STE97]. Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", *RFC2001*, January 1997,

[THO97]. Thompson, K., Miller, G., and Wilder, R., "Wide-area Internet Traffic Patterns ad Characteristics", IEEE/ACM Transactions on Networking, pp. 10--23, November 1997.

[WRI95]. Wright, G., and Stevens, W., "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1995.

[YIN90]. Yin, N., and Hluchyj, M., "Implication of dropping packets from the front of a queue", 7th ITC, Copenhagen, Denmark, October 1995.