Investigating the Effects of Active Queue Management on the Performance of TCP Applications

by
Nguyen Tuong Long Le

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2005

Approved by:

Kevin Jeffay, Advisor

F. Donelson Smith, Reader

Ketan Mayer-Patel, Reader

Jasleen Kaur, Reader

Vishal Misra, Reader

ABSTRACT
NGUYEN TUONG LONG LE: Investigating the Effects of Active Queue Management on the Performance of TCP Applications.
(Under the direction of Kevin Jeffay.)

Congestion occurs in the Internet when queues at routers fill to capacity and arriving packets are dropped ("lost"). Today, congestion is controlled by an adaptive mechanism built into TCP that regulates the transmission rate of a TCP connection. This mechanism dictates that each connection should detect instances of packet loss, interpret such instances as an indication of congestion, and respond to loss by reducing its transmission rate. After the rate has been reduced, the connection probes for additional bandwidth by slowly increasing its transmission rate.

This adaptive behavior, applied independently on each end system, has been one of the keys to the operational success of the Internet. Nevertheless, as the Internet has grown, networking researchers and the Internet Engineering Task Force (IETF) have expressed concern about the scalability of pure end systems' congestion control. For example, pure end systems' congestion control mechanism only detects and reacts to a congestion event after a router queue has overflowed. In response to these concerns, active queue management (AQM) has been proposed as a router-based mechanism for early detection of congestion inside the network. AQM algorithms execute on network routers and detect incipient congestion by monitoring some function of the instantaneous or average queue size in the router. When an AQM algorithm detects congestion on a link, the router signals end systems and provide an "early warning" of congestion. This signaling is performed either explicitly, by setting a specific bit in the header of a packet, or implicitly by dropping some number of arriving packets.

Many AQM algorithms have been proposed in recent years but none of them have been thoroughly investigated under comparable (or realistic) conditions in a real network. Moreover, existing performance studies have concentrated on network-centric measures of performance and have not considered application-centric performance measures such as response time. In this dissertation, I investigated the effects of a large collection of existing AQM algorithms on the performance of TCP applications under realistic conditions in a real network. At a high-level, the primary results are that many existing AQM algorithms do not perform as well as expected when they are used with packet dropping. Moreover, a detailed investigation of the classical random early detection, or RED algorithm, has uncovered a number of design flaws in the algorithm. I have proposed and investigated a number of modifications to RED and other algorithms and have shown that my variants

significantly outperform existing algorithms.

Overall, this dissertation shows promising results for AQM. When combined with packet marking, AQM algorithms significantly improve network and application performance over conventional drop-tail queues. Moreover, AQM enables network operators to run their networks near saturation levels with only modest increases in average response times. If packet marking is not possible, the dissertation also shows how a form of differential treatment of flows that I invented can achieve a similar positive performance improvement. Further, I also developed a new AQM algorithm that can balance between loss rate and queuing delay to improve the overall system performance.

ACKNOWLEDGMENTS

First, I thank my advisor Kevin Jeffay and my co-advisor Don Smith for their support and for teaching me how to do research. Doing a dissertation is a challenging and formidable task but being able to work with Kevin and Don has made it a rewarding experience for me. I also thank Ketan Mayer-Patel, Jasleen Kaur, and Vishal Misra for taking time from their busy schedules to serve on my dissertation committee.

I would like to thank my former advisors and mentors from Germany: Adam Wolisz, Michael Smirnov, Georg Carle, and Henning Sanneck. They got me interested in systems and networking research and gave me opportunities to work in this area in the first place.

I appreciate all of the advice, support, and friendship of former and current students in the networking and systems group in the Department of Computer Science at the University of North Carolina at Chapel Hill. My thanks also go to the faculty and staff in the Department of Computer Science at the University of North Carolina at Chapel Hill for making Sitterson Hall a friendly and great place to work. I would like to acknowledge the help and friendship of many friends that I met in Germany and in the U.S. It has been an amazing ride for me in the last twelve years.

Finally and most importantly, I thank my family for their love and support. My parents give me love and everything that I could ever ask for. My brother has been my role model and sets high academic standards for me to achieve. My uncles, aunts, and cousins in Germany and in the U.S. welcomed me to their families with open arms. I would not have made it through my journey without their help. My wife Vân Anh gives me love, hope, support, and encouragement. Ich hab' Dich lieb und wir sind cool!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACK	acknowledgment
AIMD	additive-increase / multiplicative-decrease
ALTQ	Alternate Queueing
AQM	Active Queue Management
ARED	Adaptive RED
AVQ	Adaptive Virtual Queue
BDP	bandwidth-delay product
CDF	cumulative distribution function
CCDF	complementary cumulative distribution function
cwnd	congestion window
ECN	Explicit Congestion Notification
FIFO	first-in, first-out
QoS	quality of service
PI	Proportional Integral
RED	Random Early Detection
REM	Random Exponential Marking
RIO	RED with In and Out
RIO-PS	RED with In and Out with Preferential Treatment to Short Flows
RTO	retransmission timeout
RTT	round-trip time
SACK	selective acknowledgment
TCP	Transmission Control Protocol
ssthresh	slow start threshold

Chapter 1

Introduction

Computers and their applications are now an integral part of daily life. Their widespread usage ranges from entertainment, to processing and storage of data, to control of critical infrastructures such as power grids and industrial plants. In many cases, computer applications need to communicate and share data with each other to realize their objectives. Examples of such applications are web browsing, electronic mail, text messaging, file sharing, audio streaming, video streaming, and electronic commerce.

One of the key enabling technologies for today's computer technology is the Internet that is used by millions of computers to communicate and share data with each other. At a high level, the Internet is composed of many specialized computers called *routers* that forward data between *end systems*. The routers are connected to each other by various communication *links*. End systems are typically the ultimate sources and sinks of data, i.e., end system typically do not forward data that they do not transmit or receive.

Since a computer network is a shared infrastructure, several end systems can be using the network at the same time. Overload or *network congestion* occurs when end systems simultaneously transmit more data than routers can forward. End systems can ameliorate network congestion by employing *congestion control* algorithms that detect and react to congestion in the network. End systems' congestion control allows computers to use a shared network infrastructure and enjoy good performance. It significantly contributed to the growth and success of the Internet in the nineteen eighties and nineties. However, as the Internet has grown and the number of end systems has increased, concerns have been raised about the scalability and effectiveness of the pure congestion control implemented solely on end systems.

Active queue management (AQM) in routers has been proposed as a measure to preserve and improve Internet performance [BCC+98, Flo00a]. AQM algorithms detect incipient congestion by typically monitoring a router's queue size. In its simplest form, when impending congestion is detected, AQM algorithms notify the end systems to reduce their transmission rates by proactively dropping packets before congestion actually occurs. In this manner,

AQM algorithms can help end systems' congestion control mechanisms detect and react to congestion faster and more effectively.

A large number of AQM algorithms have been proposed in research literature. However, many of them have been evaluated only by simulations under rather simple and unrealistic traffic conditions. Moreover, evaluations are often incomparable as they use different traffic models, network topologies, etc. Thus, there is currently a lack of understanding of the performance of AQM algorithms in complex and realistic network environments. This dissertation is concerned with evaluation of AQM algorithms in a real and complex laboratory network under realistic conditions.

In the rest of this Chapter, I will give a short introduction to computer networks, end systems' loss recovery and congestion control, and AQM algorithms. The loss recovery and congestion control algorithms described in this Chapter are used by the *Transmission Control Protocol* (TCP), a protocol that is currently implemented and used widely by the end systems in the Internet. Finally, I will conclude this Chapter with motivation for my dissertation, my thesis statement, and a summary of major results and contributions.

1.1 Introduction to Computer Networks

When an end system has data to send, it fragments data into units called *packets*. The end system also packages some control information into a *packet header* and places it at the beginning of each packet. In an IP network, the packet header contains the source IP address to specify the sender of the packet. The packet header also contains the destination IP address that tells whom the packet is destined for. The destination address is used by the routers to locate the receiver of the packet and to subsequently forward the packet toward the receiver.

In many cases, the sender and the receiver are not directly connected. In these cases, the sender sends the packet to a router that it is directly connected to. When the packet arrives at a router, the router uses the packet's destination address as a search key and searches its routing information database for a forwarding decision. If the router and the receiver are directly connected, the router forwards the packet to the receiver. If not directly connected, the router forwards the packet to another router that is closer to the receiver. This forwarding process is repeated until the packet finally reaches a router that is directly connected to the receiver. That router can then send the packet directly to the receiver.

Figure 1.1 shows the basic architecture of a router. A router usually has multiple network interfaces attached to different links that connect the router to other routers or subnetworks. A subnetwork is usually composed of one or multiple end systems. When a packet arrives at a router, the router uses the destination address and its local routing information database to determine the output link for that packet. In most cases, the output

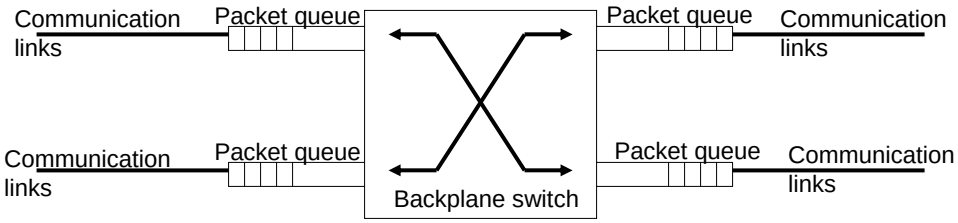

Figure 1.1: Basic router architecture.

link is not the input link (the link on which the packet arrives at the router) and packet has to be transferred from the input link to the output link via an internal switch. If multiple packets that are destined for the same output link arrive at the router via different input links simultaneously, only one packet can be forwarded onto the link and other packets are stored temporarily in a queue at the output link.

When packets arrive at a router for a given destination link at a rate faster than the link's transmission rate, the packets are buffered in a queue at the destination link. This queue is usually maintained in a "first-in, first-out" (*FIFO*) manner. With FIFO queue, arriving packets are enqueued at the end of the queue. When the router is done with the transmission of a packet, it dequeues the packet at the head of the queue and transmits that packet onto the link. The transmission time for a packet on a link is equal to the product of the packet size and the link's transmission rate. If the router's forwarding rate is greater than the packets' arrival rate (the average transmission time is smaller than the interarrival time of packets), the queue will shrink over a time interval until it becomes empty. On the other hand, if the router's forwarding rate is less than the packets' arrival rate (the average transmission time is greater than the interarrival time of packets), the queue will grow over a time interval until the queue becomes full. When a packet arrives at a full queue, i.e., the router has exhausted its storage resources and cannot enqueue another packet, that packet is dropped. When this occurs, we say that the router's queue *overflows*. Further, the default behavior of routers that only drop arriving packets when the queue is full is called *drop-tail* queuing. Drop-tail queuing is also called *drop-tail FIFO* because packets are forwarded in a FIFO (first-in first-out) manner.

1.2 TCP Loss Recovery

Since packets can be lost in transit between the sender and the receiver due to queue overflows in a computer network, a network service for reliable data exchange between applications requires end systems to implement a mechanism to detect the loss of packets and retransmit the lost packets. TCP detects the loss of packets by associating each data byte with a uniquely identified *sequence number*.

The receiver acknowledges the receipt of a data packet by sending the sender an *acknowledgment packet* (ACK packet). The ACK packet carries a sequence number indicating the sequence number of the next in-order byte expected by the receiver (i.e., the sequence number of the last byte in the last in-order data packet received at the receiver plus one). Thus, an ACK packet acknowledges to the sender that all data bytes that have a sequence number smaller than that of the ACK packet have been received by the receiver. For each data packet that arrives at the receiver out of order, the receiver retransmits an ACK packet that contains the sequence number of the last data byte that has been received in order. The ACK packets triggered by the arrival of out-of-order data packets are called *duplicate ACKs* because they carry the same sequence number of the last in-order data byte received by the receiver.

In the case that the sender can send multiple packets in a sequence, a fast way to detect loss of data packets is via a direct use of duplicate ACKs. As mentioned above, the receiver sends duplicate ACKs to inform the sender that data packets arrived at the receiver out of order. Duplicate ACKs can be caused by different network problems. First, as discussed above, duplicate ACKs can be caused by the loss of some data packets. In this case, the receiver generates a duplicate ACK for each data packet arriving at the receiver after the lost packets. Second, duplicate ACKs can be caused by replication of data or ACK packets in the network. Third, duplicate ACKs can be triggered by re-ordering of data packets in the network [Pax99] because each data packet arriving at the receiver out of order causes the receiver to generate a duplicate ACK. Of these three causes, loss is by far the most common.

From the sender's perspective, the last two network problems that cause duplicate ACKs are undesirable but they are rather "harmless" because they do not cause loss of data. However, loss of data packets that causes duplicate ACKs requires the sender to retransmit the lost data packets. Since packet replication and packet re-ordering are rare events, when the TCP sender receives three duplicate ACKs in a row, it assumes that the data packet with the sequence number carried by the duplicate ACKs has been lost and retransmits that packet.

Duplicate ACKs help sender detect and retransmit lost data packets. However, duplicate ACKs are only triggered when a lost data packet is followed by other data packets that arrive at the receiver. If the sender transmits a sequence of data packets and the last data packet in the sequence is lost, the receiver cannot detect the loss of the data packet. In this case, the receiver does not generate duplicate ACKs to inform the sender about the loss of the data packet. Furthermore, duplicate ACKs transmitted by the receiver can also be lost on their way back to the sender. Because of these problems, the sender also needs to rely on other local mechanisms to detect the loss of its data packets. The sender initializes a timer for each data packet that it transmits (more details about the timer's interval will be

provided in section 1.3.3). When a timer for a data packet expires and the acknowledgment for that data packet has not been received, the sender assumes that the data packet has been lost and retransmits it.

1.3 TCP Congestion Control

Congestion at a router causes the router's queue to build up. Since packets in a queue are usually processed in a "first in, first out" manner, a packet arriving at a router must wait in the queue until all packets that had previously entered the queue have been transmitted. Queuing delay increases end-to-end latency and has adverse effects on interactive applications such as web browsing or real-time conferencing. Under severe congestion, the router's queue can grow to its maximum length and the router cannot enqueue any arriving packets. In this case, arriving packets cannot enter the queue and are dropped.

As discussed above, queuing delay increases end-to-end latency and has ill effects on the performance of interactive applications. On the other hand, packet losses can also degrade application performance significantly. For example, a sender needs at least one or multiple round-trip times to detect and retransmit a lost packet via duplicate ACKs. In the worse case, a retransmission is triggered by a timeout which can take one or multiple seconds. Further, a sender's throughput is dramatically reduced during the recovery of lost packets. For this reason, congestion (the root of queuing delay and packet loss) should be prevented from happening. *TCP congestion control* algorithm, implemented in end systems, uses packet loss as an indication of network congestion. TCP reacts to the indication of congestion by reducing its transmission rate. (Besides packet losses, other congestion control algorithms such as TCP Vegas [BOP94], FAST [JWL04], and Sync-TCP [Wei03] also use packet delay or variations in packet delay as an indication of congestion in the network.) While many congestion control algorithms exist in the research literature, I will only provide a short introduction to the basic congestion control algorithms that are currently used by the standard TCP protocol (known as TCP Reno) in this Chapter.

The TCP congestion control algorithms follow the principle of "packet conservation" which states that packets should not enter the network at a rate faster than they leave the network in an equilibrium state [JK88]. The basic mechanisms of TCP congestion control algorithms limit the number of packets of an end system that are inside the network at any point in time. Since it is hard for a sender to know at any point in time which packets are inside the network and which packets already left the network, TCP congestion control algorithms makes a conservative assumption that all unacknowledged packets are still inside the network. (Obviously all packets that are acknowledged by the receiver have already arrived at the receiver and left the network.)

Using the conservative assumption above, TCP congestion control algorithms limit the

maximum number of unacknowledged packets that a sender is allowed to have. This maximum number of unacknowledged packets varies over time and depends on the state of the network. This maximum number of packets is also referred to as the TCP *congestion window* (cwnd). TCP congestion control is divided into two main parts. The first part controls the size of *cwnd* while an end system is trying to find an equilibrium state of the network. This part is called *slow start*. The second part controls the size of *cwnd* after an end system has reached an equilibrium state of the network. This part helps an end system stay in an equilibrium state of the network and is called *congestion avoidance*.

1.3.1 TCP Slow Start

The TCP slow start algorithm is used by an end system to seek and enter an equilibrium state of the network. TCP slow start allows a sender to increase its transmission rate rather fast by growing its *cwnd* exponentially as follows. The *cwnd* is initialized to a small number of packets (typically one or two) by the sender after a connection has been established. TCP also reinitializes *cwnd* to this value and reenters TCP slow start after a long idle period between the sender and the receiver or after the sender has experienced severe losses such that it has to seek for a new equilibrium state of the network. While in slow start phase, the sender increases its congestion window by one packet for each ACK packet that it receives. If the receiver acknowledges each data packet with an ACK packet, the sender can double its congestion window within a round-trip time (RTT). Thus, when there is no packet loss, the sender can grow its congestion window at an exponential rate. Obviously, slow start is not slow at all despite its name.

Since the sequence number in ACK packets are cumulative, a receiver can increase efficiency in network usage and reduce the number of ACK packets by acknowledging every other data packet that it receives with an ACK packet. In this case, the sender's congestion window grows more slowly than in the previous case but the growth rate is still exponential.

Slow start allows a TCP sender to probe for available bandwidth very fast because the sender can grow its congestion window exponentially in this phase. As the sender approaches the limit of capacity in the network, it exits slow start and enters congestion avoidance. The TCP sender infers that it has reached the limit of the network's capacity after it experiences the first packet loss. When this happens, TCP exits its slow start phase and enters the congestion avoidance phase. Alternatively, a TCP sender also exits the slow start phase and enters the congestion avoidance phase after its congestion window becomes larger than a variable called *slow start threshold* or *ssthresh*. This variable is a conservative estimate of available bandwidth in the network and is dynamically updated as explain in section 1.3.2.

1.3.2 TCP Congestion Avoidance

Unlike in the slow start phase, a TCP sender increases its *cwnd* more moderately in the congestion avoidance phase. The rationale for this is that the TCP sender has already reached the equilibrium state of the network and should avoid any drastic change in its transmission rate that may destroy the equilibrium state. In the congestion avoidance phase, the TCP sender increases its *cwnd* by $1/cwnd$ for each ACK packet that it receives. This means that the TCP sender can increase its *cwnd* by 1 packet after successfully sending *cwnd* packets. Since the TCP sender can have at most *cwnd* outstanding packets, i.e., at most *cwnd* packets that have not been acknowledged by the receiver, and it takes a round-trip time for an ACK of a data packet to arrive at the sender, the sender can increase its *cwnd* by 1 packet per round-trip time. This gives the TCP sender an *additive increase* of its *cwnd* in the congestion avoidance phase because *cwnd* grows linearly as a function of time (as opposed to *cwnd* growing exponentially in the slow start phase).

When a TCP sender detects loss of packets, it infers that packets are lost due to congestion in the network. More specifically, a TCP sender assumes that loss is an indication that the current transmission rates of the end systems have exceeded the available capacity of the link. In this case, the TCP sender, together with other TCP senders sharing a bottleneck link, is sending packets at a faster rate than this link can forward the packets. Depending on how packet loss is detected, the TCP sender can take one of the two following alternate approaches.

If packet loss is detected via triple duplicate ACKs, the sender knows that at least some of the packets arrived at the receiver. In this case, the sender remains in the congestion avoidance phase but reduces its *cwnd* by half. The rationale for this is that the TCP sender was previously able to transmit packets without loss at that transmission rate and hence it is safe to start probing for available bandwidth again from that transmission rate. This gives the TCP sender a *multiplicative decrease* of its *cwnd*.

If packet loss is detected via a timer's expiration, the sender infers that a large fraction of its packets were lost (because the sender has not received any ACKs) and that there is severe congestion in the network. In this case, the TCP sender reduces its *cwnd* to one packet and switches to the slow start phase to rediscover the equilibrium state of the network.

After a loss event, a TCP sender infers that the available share of bandwidth is less than its current *cwnd*. Hence, the TCP sender sets its *ssthresh* to half of its current *cwnd* to prevent *cwnd* from overshooting the available bandwidth when it enters the slow start phase next time.

1.3.3 Round-trip Time Estimation

Estimation of round-trip time (RTT) plays an important role in the TCP recovery algorithm. Round-trip times experienced by a connection usually consists of a propagation delay and queuing delay. While propagation delays usually are constant, queuing delays can vary because queues at routers along the path of a connection can grow or shrink over time.

Due to the variable queuing delays, it is not easy for the end systems to estimate the round-trip time accurately. However, an accurate estimation of round-trip times allows TCP to set its timers efficiently. An underestimation of round-trip times can lead to timer intervals that are too short. Since timers are used to detect loss of data packets at the sender upon their expiration, short timers will cause unnecessary retransmissions of data packets and waste of bandwidth. On the other hand, an overestimation of round-trip times can cause the sender to wait too long to detect a packet loss and increase experienced delay for applications running on top of TCP. Since the RTT of a connection can change over time, TCP needs to dynamically adjust its RTT estimate. TCP adjusts its RTT estimate to the trend of RTT measurements but uses a low pass filter to smooth out fluctuations in RTT measurements. For a RTT measurement m_i at time i, the RTT estimate a_i at time i is updated as follows:

$$a_i = (1 - g)a_{i-1} + gm_i \tag{1.1}$$

where g is a constant ($0 < g < 1$). TCP also estimates and updates the variation v_i of round-trip times for measurement at time i as follows:

$$v_i = v_{i-1} + g(|m_i - a_i|) \tag{1.2}$$

The retransmission timeout (RTO) has to be on the order of the round-trip time of a connection to allow for data packets to arrive at the receiver and for their corresponding ACK packets to come back to the sender. Van Jacobson proposed that the retransmission timeout has to be computed from both the mean and variance of RTT measurements to reflect wide fluctuations in RTTs [JK88]. The retransmission timeout RTO_i at time i is computed as follows:

$$RTO_i = a_i + 4v_i \tag{1.3}$$

1.4 Active Queue Management

Although its service is critical to the performance of many applications, the Internet has mainly relied on the end systems' cooperative behavior to deal with congestion when it

occurs. Specifically, the end systems' TCP transport protocol is the sole agent responsible for reducing the end systems' sending rate when congestion is present in the Internet. When the TCP congestion control algorithm detects packet losses, it assumes that these losses are caused by queue overflows in routers due to congestion. The TCP congestion control algorithm helps relieve congestion in the Internet by reducing the end systems' sending rate until the end systems experience no further packet losses. This cooperative behavior, performed independently by all active end systems simultaneously, and TCP's conservative probing algorithm for available bandwidth have been the key to the operational success of the Internet.

Nevertheless, as the Internet grew, networking researchers and the Internet Engineering Task Force (IETF) were concerned about the scalability of the pure end systems' congestion control. For example, end systems' congestion control algorithms only detect and react to a congestion event *after* a router's queue overflows. Furthermore, since each individual connection's view of the network is limited, and each connection usually has a only small proportion of the bandwidth on a high-speed link, inferring congestion at the end systems is difficult and imprecise. Thus, congestion control implemented solely at the end systems is inefficient.

Beyond efficacy, networking researchers were also concerned about other disadvantages of drop-tail FIFO. First, drop-tail FIFO only limits the aggregate transmission rate of all flows at a bottleneck link by dropping arriving packets when this aggregate rate exceeds the link capacity and the queue is full. However, drop-tail FIFO does not control and limit the transmission rate of each flow. Because of this, drop-tail FIFO tolerates unfairness among flows and potentially allows high-bandwidth flows to dominate low-bandwidth flows. In extreme situations when one or a few high-bandwidth flows do not react to packet drops as an indication of network congestion, these few flows can consume the entire bandwidth of a link while all other flows reacting to congestion indications reduce their *cwnd* to 0. Flows that react to congestion indications are called *responsive flows*. On the other hand, flows that ignore congestion indications are called *unresponsive flows*.

Second, drop-tail FIFO can cause synchronization among flows because when the queue is full, all arriving packets from different flows are dropped at the same time. Flow experiencing packet drops reduce their transmission rates at the same time and then start increasing their transmission rates at the same time after congestion has abated. Eventually, these flows reach the full queue and reduce their transmission rates at the same time again. As this process perpetuates, these flows effectively synchronize their transmission rates. Synchronizations among flows can cause unnecessarily bursty traffic which leads to undesired effects such as high packet loss rates and low link utilizations.

Because of the problems discussed above, researchers and the IETF proposed *active queue management* (AQM) as a mechanism for detecting congestion inside the network.

Further, they have strongly recommended the deployment of AQM in routers as a measure to preserve and improve Internet performance [BCC+98, Flo00a]. AQM algorithms run on routers and detect incipient congestion by typically monitoring the instantaneous or average queue size. When the average queue size exceeds a certain threshold but is still less than the capacity of the queue, AQM algorithms infer congestion on the link and notify the end systems to back off by proactively dropping some of the packets arriving at a router. Alternately, instead of dropping a packet, AQM algorithms can also set a specific bit in the header of that packet and forward that packet toward the receiver after congestion has been inferred. Upon receiving that packet, the receiver in turns sets another bit in its next ACK. When the sender receives this ACK, it reduces it transmission rate as if its packet were lost. The process of setting a specific bit in the packet header by AQM algorithms and forwarding the packet is also called *marking*. A packet that has this specific bit turned on is called a *marked packet*.

End systems that experience the marked or dropped packets reduce their transmission rates to relieve congestion and prevent the queue from overflowing. Some AQM algorithms also attempt to detect and control unresponsive flows. With AQM, congestion is prevented before it actually occurs. Thus, the deployment of AQM could lead to a high throughput, low loss, and low queuing delay network. This would particularly improve performance of interactive applications such as Web browsing and real-time conferencing [BCC+98].

Most AQM algorithms avoid global synchronization among flows by introducing randomness in the decision of marking or dropping arriving packets. When congestion is suspected on a link, most AQM algorithms do not mark or drop arriving packets deterministically but randomly. The marking or dropping probability for an arriving packet usually depends on the estimated degree of congestion on the link.

1.5 Evaluation of AQM Algorithms

Although many AQM algorithms have been proposed in recent years, none of them have actually been widely deployed. While popular AQM algorithms such as Random Early Detection (*RED*) [FJ93] or its variants are implemented by most router vendors and shipped in virtually every router, AQM is rarely turned on. The reason is that AQM algorithms are usually complex and their effects on network and application performance are not well understood. For example, AQM algorithms are typically characterized by several parameters and little is known about how to set these parameters to achieve good performance for TCP applications.

Although guidelines and recommended parameter settings are provided for most AQM algorithms (e.g., [Flo00b]), most of these guidelines and recommended parameter settings have been shown to be suboptimal in practice or even harmful in some cases [MBDL99,

CJOS01, LAJS03]. For example, in one study, good parameter settings for the prominent AQM algorithm RED that improved network and application performance were only found by exhaustive search [CJOS01]. Furthermore, these good parameter settings were significantly different from the parameter settings recommended by the algorithm designers. It is believed that network operators do not have confidence in AQM and the recommended parameter settings for AQM algorithms because none of the proposed AQM algorithms have been thoroughly evaluated under realistic conditions in real networks.

A reason for the lack of understanding of the effects of AQM algorithms is that most existing evaluations were done only by simulation and only demonstrated how AQM algorithms work in simplistic environments with a small number of flows. Further, traffic models used in existing evaluation studies usually do not capture complex characteristics of real Internet traffic. Thus, it is unclear whether and how AQM algorithms work under realistic conditions such as high-speed links and in the presence of a large aggregation of flows. Furthermore, there has been no systematic or comparison study of AQM algorithms in a complex and realistic environment. Thus, although many researchers agree that the deployment of AQM is necessary for stability and performance of the Internet, there have been many debates about *which* AQM algorithm is the right one for deployment.

Since none of AQM algorithms have been evaluated in *real networks*, there is currently still a lack of fundamental understanding of the interactions between AQM and end systems in a complex and realistic environment. For example, the models for traffic and propagation delays used in this dissertation were carefully derived from large-scale measurements studies. These models contain characteristics of real Internet traffic and allow a realistic evaluation of network protocols and mechanisms in a laboratory network. Further, while many studies have focused on whether AQM algorithms can control and stabilize router queues, the effects of AQM algorithms on the performance of TCP applications such as response times or connection durations have been largely ignored. Nonetheless, these effects of AQM algorithms on TCP applications can be more important to Internet users than stabilization of router queues.

In this dissertation, I thoroughly investigate the effects of a large collection of prominent AQM algorithms on the performance of TCP applications in a *complex* and *realistic* environment with a rich traffic mix. The AQM algorithms that I investigate are:

- Approximate Fairness through Differential Dropping (AFD) [PBPS03],
- Adaptive Random Early Detection (ARED) [FGS01],
- Adaptive Virtual Queue (AVQ) [KS01],
- BLUE [FKSS02],
- Proportional Integral controller [HMTG01],

- Random Exponential Marking (REM) [ALLY01],
- and RED with In and Out with Preferential treatment to Short flows (RIO-PS) [GM01].

The outcome of my study sheds light on the effects of these algorithms on network and application performance. For example, I investigate in this dissertation why ARED, a contemporary redesign of the prominent RED algorithm, consistently gave poor performance. Based on the findings in my investigation, I propose a number of modifications for ARED that achieve significant performance improvement. Further, I design two new AQM algorithms and demonstrate that they outperform all existing AQM algorithms.

1.6 Thesis Statement

Active queue management has been proposed by networking researchers and the Internet Engineering Task Force as a mechanism to preserve and improve the performance for Internet applications. Many AQM algorithms have been proposed in recent years but none of them have been thoroughly investigated under realistic conditions in a real network. For this reason, there is currently a lack of understanding of the effects of AQM algorithms on network and application performance.

In this dissertation, I carry out a performance analysis for a large collection of AQM algorithms and thoroughly investigate the effects of these algorithms on the performance of TCP applications under realistic conditions in a real network. I show that existing AQM algorithms only obtain significant performance improvement for TCP applications when they are used in combination with the ECN signaling protocol. Without ECN, existing AQM algorithms only give a small performance improvement over conventional drop-tail queues. I then show that a new approach in AQM designs is necessary to deliver good performance without relying on ECN and demonstrate how that is done.

1.7 Summary of Results and Contributions

Experimental results in this dissertation supports the following findings. These findings are based on the premise that response times and connection durations are the primary performance metric for TCP applications and that other performance metrics such as loss rates and link utilization are secondary.

- At 80% load or below, drop-tail obtained performance that was competitive to that of all AQM algorithms. Further, since drop-tail closely approximated the performance of the uncongested network at 80% load, there appears to be no need for AQM algorithms at 80% load or lower.

- At 90% load or higher, PI gave a small performance improvement over drop-tail and other AQM algorithms. However, when PI and REM were operated in the *marking mode*, they gained significant performance improvement and significantly outperformed drop-tail.
- ARED, the arguably most prominent AQM algorithm, obtained very poor performance and consistently gave poorer performance than drop-tail at all loads.
- Two modification of the ARED algorithms proposed in this dissertation, ARED "byte mode" and ARED/ECN "new gentle", outperformed drop-tail and gave significant performance improvement over the original ARED algorithm.
- Differential treatment of flows improved performance for a majority of flows significantly. For example, the Differential Congestion Notification (DCN) algorithm proposed in this dissertation approximated the performance of the uncongested network even at very high loads. On contrary to most existing AQM algorithms, DCN did not rely on marking packets in order to deliver good performance for TCP applications.

1.8 Organization of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 provides a survey of AQM algorithms and reviews relevant related work. Chapter 3 explains the experimental methodology that was used to investigate the effects of AQM algorithms on the performance of TCP applications. Chapter 4 and 5 show the effects of AQM algorithms on the performance of Web application, a specific TCP application but arguably the most important one. Chapter 6 demonstrates the effects of AQM algorithms on the performance of TCP applications. Chapter 7 investigates the interaction between AQM algorithms and link-level buffering. Chapter 8 summarizes and concludes the dissertation.

Chapter 2

Background and Related Work.

AQM has been a very active research area in the last decade and a large number of AQM algorithms have been proposed for different purposes. At a high level, existing AQM algorithms were designed for the following purposes: stabilizing router queues, approximating fairness among flows, controlling unresponsive high-bandwidth flows, and improving performance for short connections. In this Chapter I will review the most prominent AQM algorithms in these categories[1]. As briefly discussed in section 1.4, AQM algorithms can notify the end systems of impeding congestion by dropping arriving packets as an implicit congestion indication or by setting a specific bit in the packet headers as an explicit congestion indication. This explicit mechanism for delivering congestion indications to end systems will be reviewed in section 2.5. I also review existing evaluation studies for AQM algorithms in section 2.6.

2.1 Stabilizing Router Queues

The primary goal of AQM is to replace drop-tail and prevent queue overflows (as discussed in section 1.4, drop-tail allows queue overflows to occur and can cause negative effects on network and application performance). But beyond this goal, many AQM algorithms also attempt to stabilize router queues to limit queuing delay for packets. The incentives of stabilizing router queues is the ability to provide quality of service (QoS) [BCS94, BBC+98]. A challenge for AQM algorithms is that stabilization of router queues should not be achieved at the expenses of decreased link throughput or increased packet loss rates. In this section, I will review how prominent AQM algorithms in research literature attempt to accomplish this challenge.

[1]Note that the categorization presented here is rather simplistic and some AQM algorithms can have components that fall into different categories. In these cases, I will review the components of these AQM algorithms in different categories.

2.1.1 Random Early Detection (RED)

The Random Early Detection (RED) algorithm was introduced in 1993 and was the first AQM algorithm. Its purpose was to provide congestion avoidance and maintain the network in a region of high throughput and low delay [FJ93]. RED is designed to work with end systems' congestion control algorithms such as TCP and uses packet marking or dropping as a mechanism to provide feedback of congestion to the end systems. Furthermore, RED attempts to keep the average queue size low but absorbs bursty traffic and transient congestion by allowing some fluctuations in the actual queue size. RED also attempts to avoid global synchronization by marking or dropping packets randomly. As explained in section 1.4, global synchronization can occur when a queue overflows and all TCP connections sharing a common link experience packet losses simultaneously. In this event, all TCP connections reduce their window to one, go through slow start at the same time, and stimulate global synchronization.

RED uses a weighted average queue size to detect congestion in the network. This weighted average queue size is computed by running the instantaneous queue size through a low-pass filter. The motivation for using the weighted average queue size rather than the instantaneous queue size is to detect persistent congestion at a router but tolerate transient congestion by allowing the instantaneous queue size to grow temporarily. With this design guideline, RED avoids bias against bursty traffic (unlike drop-tail FIFO). When a packet arrives at the router and the instantaneous queue size is not empty, the average queue size is updated as follows:

$$avg \leftarrow (1 - w_q)avg + w_q q \tag{2.1}$$

where q, avg, and w_q are the instantaneous queue size, the average queue size and the coefficient for the low-pass filter.

If the instantaneous queue size is empty (the link has been idle for a period), the average queue size is updated as follows:

$$avg \leftarrow (1 - w_q)^m avg \tag{2.2}$$

where m is the number of packets of average size that could have arrived to an empty queue during the idle period.

Initially, the recommended value for the coefficient of the low-pass filter was 0.002. However, revised guidelines for setting RED parameters recommend a value for the coefficient of the low-pass filter such that the weighted average queue size reaches 63% of the instantaneous queue size within ten round-trip times [FGS01].

After computing the average queue size by using either equation 2.1 or 2.2, the RED algorithm executes the pseudo code shown in figure 2.1 to determine the dropping or marking

if $q < min_{th}$ **then**
  $p \leftarrow 0$
else if $min_{th} \leq q \leq max_{th}$ **then**
  $p \leftarrow max_p(q - min_{th})/(max_{th} - minth)$
else
  $p \leftarrow 1$
end if

Figure 2.1: Pseudo code for RED

Parameters	Description
w_q	Coefficient of a low-pass filter for computing the average queue size
max_p	Maximum mark or drop probability when average queue size varies between min_{th} and max_{th}
min_{th}	Low queue threshold for computing drop or mark probability for arriving packets
max_{th}	High queue threshold for computing drop or mark probability for arriving packets

Table 2.1: RED Parameters

probability for arriving packets. The parameters for RED are summarized in table 2.1.

When the weighted average queue size is smaller than a minimum threshold (min_{th}), RED infers that there is no congestion in the network. In this case, no arriving packets are marked or dropped. When the weighted average queue size is between the minimum threshold (min_{th}) and the maximum threshold (max_{th}), RED infers an incipient congestion in the network and marks or drops arriving packets randomly. The probability of marking or dropping packets is proportional to the weighted average queue size and varies linearly between 0 and a maximum drop probability (max_p, typically 0.10). If the weighted average queue size exceeds max_{th}, RED infers a severe congestion in the network and drops all arriving packets. Figure 2.2 demonstrates different operational regions of the RED algorithm when the average queue length is smaller than min_{th}, between min_{th} and max_{th}, and larger than max_{th}. Obviously, the actual size of the queue must be greater than max_{th} to absorb transient bursts of packet arrivals.

2.1.2 Random Early Detection with "Gentle Mode"

Firoiu and Borden used the TCP throughput equation developed by Padhye et al. to estimate the average queue size of a router carrying n TCP flows as a function of the loss rate p: $q = G(p)$ [FB00]. The throughput of a TCP flow is expressed as a function of round-trip time R and packet loss rate p [PFTK98].

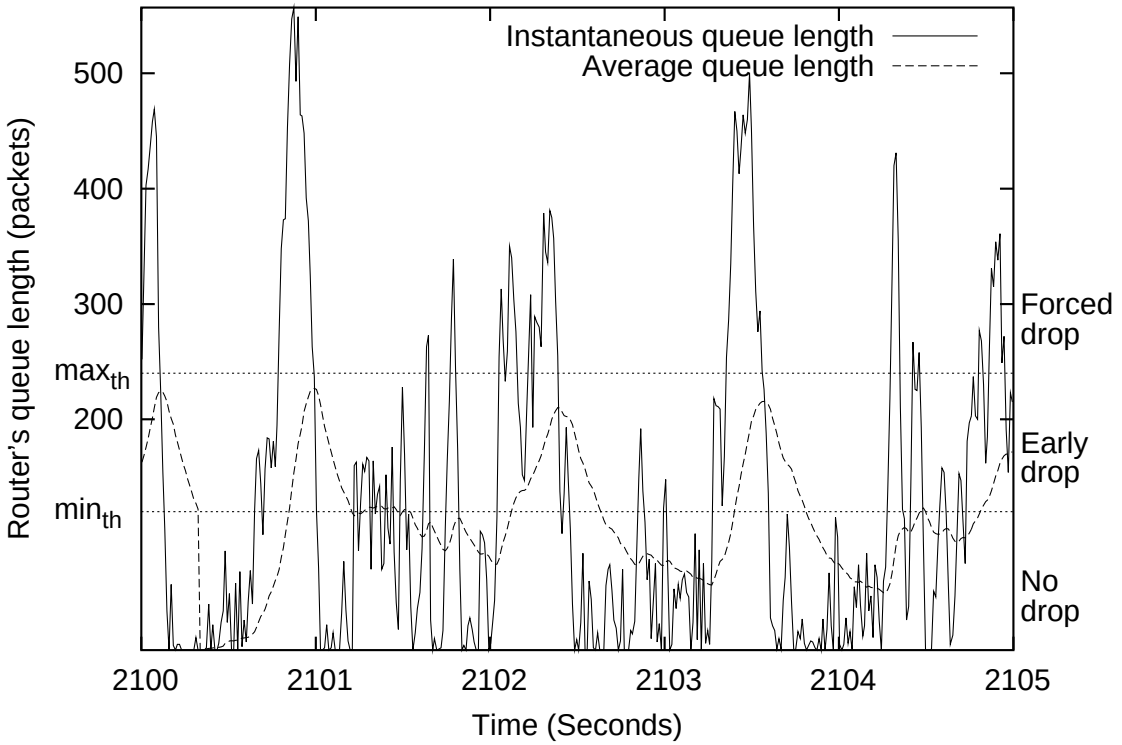

Figure 2.2: Operational regions of RED

$$T(p,R) = M\frac{\frac{1-p}{p}+\frac{W(p)}{2}+Q(p,W(p))}{R(\frac{b}{2}W(p)+1)+\frac{Q(p,W(p))F(p)T_0}{1-p}} \quad \text{if } W(p)<W_{max} \tag{2.3}$$

$$T(p,R) = M\frac{\frac{1-p}{p}+\frac{W_{max}}{2}+Q(p,W_{max})}{R(\frac{b}{8}W_{max}+\frac{1-p}{pW_{max}}+2)+\frac{Q(p,W_{max})F(p)T_0}{1-p}} \quad otherwise \tag{2.4}$$

where M is the average packet size, T_0 is the initial TCP time-out, b is the number of packets acknowledged by an ACK (usually 2), and W_{max} is the maximum window of a TCP receiver. W, Q, and F also have the following expressions.

$$W(p) = \frac{2+b}{3b}+\sqrt{\frac{8(1-p)}{3bp}+(\frac{2+b}{3b})^2} \tag{2.5}$$

$$Q(p,w) = min\left(1,\frac{(1-(1-p)^3)(1+(1-p)^3)(1-(1-p)^{(w-3)})}{1-(1-p)^w}\right) \tag{2.6}$$

$$F(p) = 1+p+2p^2+4p^3+8p^4+16p^5+32p^6 \tag{2.7}$$

Further, Firoiu and Borden modeled the loss rate p of a router running the RED algo-

if $q < min_{th}$ **then**
  $p \leftarrow 0$
else if $min_{th} \leq q \leq max_{th}$ **then**
  $p \leftarrow max_p(q - min_{th})/(max_{th} - minth)$
else if $max_{th} \leq q \leq 2max_{th}$ **then**
  $p \leftarrow max_p + (1 - max_p)(q - max_{th})/max_{th}$
else
  $p \leftarrow 1$
end if

Figure 2.3: Pseudo code for GRED

rithm as a function of the average queue size q: $p = H(q)$. They pointed out that a system consisting of n TCP flows and a router running the RED algorithm eventually converges to an equilibrium point (p_s, q_s) that satisfies the two equations $q_s = G(p_s)$ and $p_s = H(q_s)$ (p_s and q_s are the packet loss rate and average queue size at the equilibrium point).

However, if $p_s > max_p$, the equilibrium point of a system cannot be reached due to the discontinuity in the drop probability of the RED algorithm (increasing from max_p to 1.0 immediately when max_{th} is reached). In this case, the router queue oscillates between being empty and being full and can cause undesirable or harmful effects [FB00].

To fix this problem, Floyd proposed a modification to the original RED algorithm that introduced a "gentle mode" in which the mark or drop probability increases linearly between max_p and 1.0 as the average queue length varies between max_{th} and $2max_{th}$ [Flo00b]. The new algorithm is expressed in pseudo code in figure 2.3.

The new RED algorithm with the "gentle mode" is called "Gentle RED" or GRED. Figure 2.4 shows the drop probability function of RED and GRED. As can be seen in figure 2.4, the GRED algorithm does not have any discontinuity in its drop probability function. This can potentially fix the queue instability problem that was pointed out by Firoiu and Borden.

2.1.3 Adaptive Random Early Detection (ARED)

Feng et al. argued that a weakness of RED is that it does not take into consideration the number of flows sharing a bottleneck link [FKSS99]. As discussed in Chapter 1, a TCP flow in congestion avoidance reduces its transmission rate by half when it experiences a packet mark or drop. If the bandwidth of a bottleneck link is shared equally by n flows (each flow receives $1/n$ bandwidth of the link), a single packet mark or drop causes one flow to reduce its transmission rate to $0.5n^{-1}$ and reduces the offered load by a factor of $(1 - 0.5n^{-1})$. It is obvious that as n increases, the effect of a packet mark or drop on reducing the aggregate transmission rates of n TCP flows decreases. Thus, when n is large, RED either has to incur a high packet loss rate or is not effective in reducing load on a

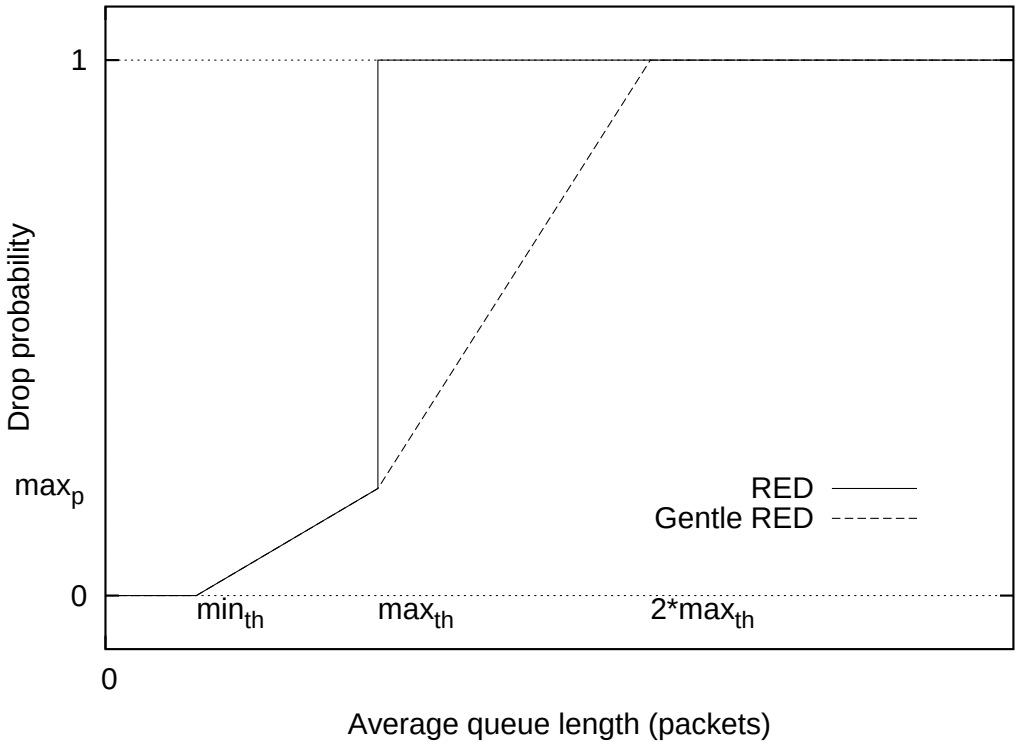

Figure 2.4: Drop probability function of RED and Gentle RED

congested link and in controlling the queue length. On the other hand, when n is small, RED can be too aggressive, i.e., it drops too many packets, and can cause underutilization of an Internet link.

Feng et al. concluded that RED needs to be dynamically tuned as the of the traffic on a link changes, i.e., the number of flows on an Internet link is not known a priori [FKSS99]. They proposed a self-configuring algorithm for RED by adjusting max_p in 1999. In their algorithm, max_p is adjusted every time the average queue length falls out of the target range between min_{th} and max_{th} [FKSS99]. When the average queue length is smaller than min_{th}, max_p is decreased multiplicatively to reduce RED's aggressiveness in marking or dropping packets; when the queue length is larger than max_{th}, max_p is increased multiplicatively. This algorithm is expressed in pseudo code in figure 2.5. α and β in the pseudo code are constants that have to be chosen by network operators.

Floyd et al. improved upon Feng's original adaptive RED algorithm by replacing the MIMD (multiplicative increase multiplicative decrease) approach with an AIMD (additive increase multiplicative decrease) approach for adapting max_p slowly in 2001 [FGS01]. The pseudo code for updating max_p proposed by Floyd et al. is shown in figure 2.6. They also provided guidelines for choosing min_{th}, max_{th}, and the coefficient of the low-pass filter for computing the weighted average queue size. The Adaptive RED version proposed by Floyd

On every update for queue average q:
if $min_{th} < q < max_{th}$ **then**
  $status \leftarrow Between$
  $max_p \leftarrow 0$
end if
if $q < min_{th}$ && $status != Below$ **then**
  $status \leftarrow Below$
  $max_p \leftarrow max_p/\alpha$
end if
if $q > max_{th}$ && $status != Above$ **then**
  $status \leftarrow Above$
  $max_p \leftarrow max_p\beta$
end if

Figure 2.5: Pseudo code for updating max_p by Feng et al.

On every update interval (0.5 seconds) for max_p:
if $q > min_{th} + 0.6(max_{th} - min_{th})$ && $max_p \leq 0.5$ **then**
  $max_p \leftarrow max_p + min(0.01, max_p/4)$
else if $q < min_{th} + 0.4(max_{th} - min_{th})$ && $max_p \geq 0.01$ **then**
  $max_p \leftarrow max_p/\beta$
end if

Figure 2.6: Pseudo code for updating max_p by Floyd et al.

et al. (referred to herein as "ARED") also includes the "gentle mode" that was discussed in 2.1.2. The parameters for the ARED algorithm proposed by Floyd et al. are summarized in table 2.2.

According to Floyd et al., one of the original RED algorithm's main weaknesses is that it cannot control the router's average queue size effectively and predictably. When max_p is high or congestion on the link is light, RED keeps the average queue size near min_{th}. On the other hand, when max_p is low or the link is heavily congested, RED's average queue size grows to $maxth$. Floyd et al. claimed that ARED does not have this problem since it dynamically adjusts max_p. They also demonstrated via simulations that ARED can achieve good and predictable performance without requiring hand-tuning its parameter settings. Further, they claimed that unlike RED, ARED is relatively insensitive to parameter settings [FGS01].

2.1.4 Proportional Integral (PI) controller

Hollot et al. applied control theory to design a Proportional Integral (PI) controller that attempts to avoid queue overflows or an empty queue in 2001. They argued that a queue overflow or an empty queue are undesirable because a queue overflow causes packet losses

Parameters	Description
w_q	Coefficient of a low-pass filter for computing the average queue size
β	Decrease factor for adapting max_p
min_{th}	Low queue threshold for computing drop or mark probability for arriving packets
max_{th}	High queue threshold for computing drop or mark probability for arriving packets

Table 2.2: ARED parameters

and an empty queue results in underutilization of the link. The PI algorithm regulates the queue length around a target value called the "queue reference" (q_{ref}) [HMTG01]. Further, PI also attempts to keep the queue and its variation small to reduce the queuing delay. PI uses instantaneous samples of the queue length taken at a constant sampling frequency as its input. Figure 2.7 demonstrates how PI samples the router's instantaneous queue length at constant intervals.

The mark or drop probability is computed as

$$p(kT) = \alpha(q(kT) - q_{ref}) - \beta(q((k-1)T) - q_{ref}) + p((k-1)T) \tag{2.8}$$

where $p(kT)$ is the mark or drop probability at the k^{th} sampling interval, $q(kT)$ is the instantaneous sample of the queue length and T is reciprocal of the sampling frequency. The final mark or drop probability for a packet is scaled the size of that packet to approximate the theoretical fluid model.

$$p_{arriving\ packet} = p(kT) \times \frac{pktsize_{arriving\ packet}}{pktsize_{average}} \tag{2.9}$$

A close examination of equation 2.8 shows that the drop probability increases in sampling intervals when the queue length is higher than its target value (when $q(kT) > q_{ref}$, $q(kT) - q_{ref} > 0$ and causes $p(kT)$ to increase). Furthermore, the drop probability also increases if the queue has grown since the last sample (reflecting an increase in network traffic). Conversely, the drop probability in a PI controller is reduced when the queue length is lower than its target value or the queue length has decreased since its last sample. The parameters for PI are summarized in table 2.3. The parameters α, β, and T depend on the link capacity, the maximum round-trip time, and the expected number of active flows using the link.

Hollot et al. demonstrated via a number of simulations that PI can stabilize the router queue around a given queue reference. On the other hand, when the same simulations were repeated with RED, RED either showed oscillatory behavior or sluggish behavior (the router queue took a long time to converge to an equilibrium state). In particular, while PI

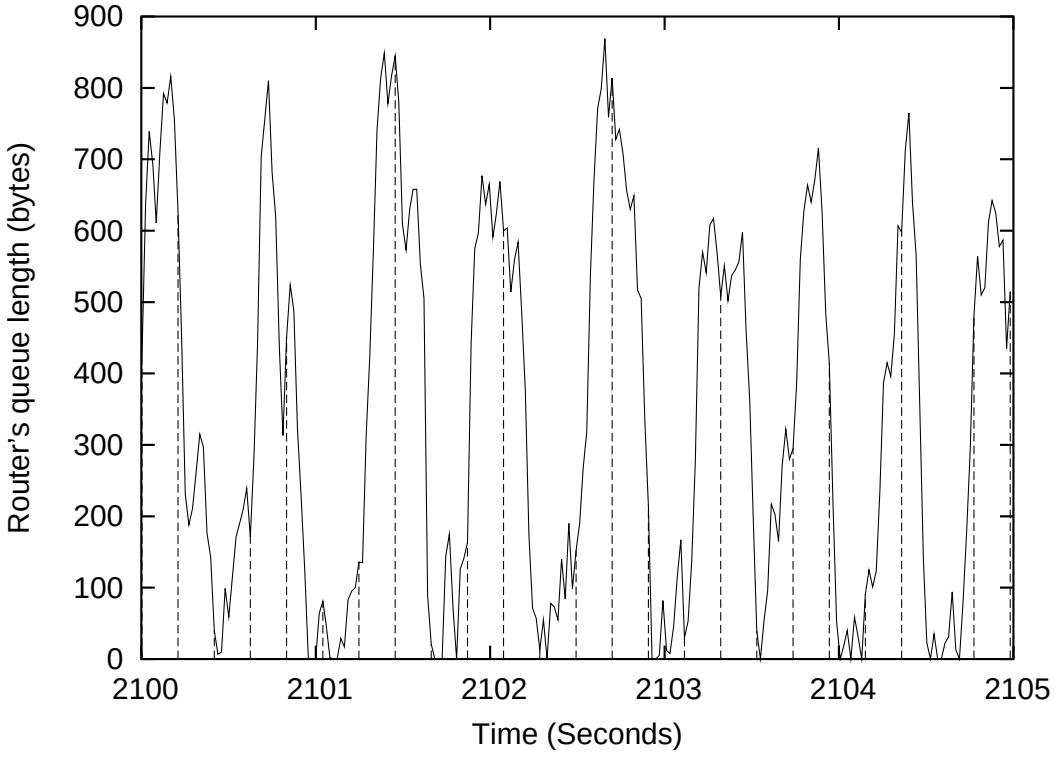

Figure 2.7: Sampling operation of PI

Parameters	**Description**
q_{ref}	Target queue reference for the instantaneous queue
α and β	Coefficients for PI equation
T	Sampling interval for the instantaneous queue

Table 2.3: PI Parameters

managed to avoid queue overflows, RED was not effective in controlling the router queue and allowed queue overflows to occur frequently in the simulations. This unstable behavior of RED could be observed when the network load suddenly changed. Eventually, RED was able to control the router queue and avoid queue overflows but RED needed a long period to respond to changes in network load. Further, RED also showed oscillatory behavior in some cases even when the network load was constant. Hollot et al. attributed these poor behaviors of RED to its low-pass filter that is used for computing the weighted average queue size.

2.1.5 Random Exponential Marking (REM)

Athuraliya et al. proposed the Random Exponential Marking (REM) algorithm in 2001 [ALLY01]. REM attempts to obtain high utilization, low loss, and low queuing delay. The key insight is that REM uses a congestion measure called "price" that is decoupled from performance measures such as packet loss or queue length. REM periodically samples the router queue and updates the congestion measure to reflect any mismatch between packet arrival and departure rates at the link (i.e., the difference between the demand and the service rate), and any queue size mismatch (i.e., the difference between the actual queue length and its target value).

Given the k^{th} samples of the router queue and the mismatch between packet arrival and departure rates, the congestion measure $p(kT)$ at time kT is computed by:

$$p(kT) = max(0, p((k-1)T) + \gamma(\alpha(q(kT) - q_{ref}) + x(kT) - c)) \tag{2.10}$$

where c is the link capacity (in packet departures per unit time), $q(kT)$ is the queue length, and $x(kT)$ is the packet arrival rate, all determined at time kT. As with ARED and PI, the control target is only expressed by the queue size. The mark or drop probability in REM is defined as

$$prob(kT) = 1 - \phi^{-p(kT)} \tag{2.11}$$

where $\phi > 1$ is a constant. The parameters for REM are summarized in table 2.4.

In overload situations, the packet arrival rate exceeds the link capacity and the router queue grows. Because of this, the congestion price increases and more packets are dropped or marked to signal TCP senders to reduce their transmission rate. Conversely, when congestion abates, the packet arrival rate will eventually become smaller than the link capacity and the router queue shrinks. In this case, the congestion price is reduced and REM drops or marks fewer packets. This allows the senders to potentially increase their transmission rates.

When there is a positive rate mismatch, i.e., the packet arrival rate is higher than the

Parameters	Description
q_{ref}	Target queue reference for the instantaneous queue
α and γ	Constants for computing the "congestion price"
ϕ	Constant for computing the mark or drop probability
T	Sampling interval for the instantaneous queue

Table 2.4: REM Parameters

link capacity, over a time interval, more packets are backlogged at the router and cause the router queue to increase. Conversely, a negative rate mismatch over a time interval will drain the queue. Thus, REM is similar to PI because the rate mismatch can be detected by comparing the instantaneous queue length with its previous sampled value. Furthermore, when drop or mark probability is small, the exponential function can be approximated by a linear function [Ath02].

2.1.6 Adaptive Virtual Queue (AVQ)

Kunniyur and Srikant proposed a rate-based AQM scheme called Adaptive Virtual Queue (AVQ) in 2001 [KS01]. They argued that since PI detects congestion and adapts the mark or drop probability by monitoring the router queue, PI performs poorly when the router queue size is small. In other words, PI delivers poor performance when it attempts to control the router queue around a small target queue threshold. Kunniyur and Srikant proposed the AVQ algorithm that attempts to achieve the same performance goals as PI but can operate with small router buffers.

AVQ attempts to control a router queue by matching the input rate and the link capacity. AVQ decouples its mark or drop probability from the physical router queue by maintaining a virtual queue that is serviced by a virtual link capacity. Because of this feature, Kunniyur and Srikant argued that AVQ can control a router queue even with a small router buffer. The virtual link capacity that services the virtual queue is adjusted based on the packet arrival rate, i.e., the input rate. The virtual capacity is reduced when the input rate is higher than the actual link capacity. When the input rate is lower than the actual link capacity, the virtual link capacity is increased. A packet is marked or dropped when it arrives at a full virtual queue. When congestion occurs, more packets are marked or dropped because the input rate is higher than the actual link capacity causing the virtual link capacity to decrease. This in turn causes the end systems' TCP to back off to relieve congestion at the router.

Let α be a positive constant and let λ, C, and γ be the packet arrival rate, the physical link capacity, and the desired utilization on the link. Further, let $\tilde{C}$ and $\dot{\tilde{C}}$ be the virtual queue capacity and its derivative. The virtual capacity $\tilde{C}$ is updated by AVQ on each packet arrival as follows.

Define:
B = buffer size
s = arrival time of previous packet
t = current time
b = number of bytes in current packet
VQ = number of bytes currently in the virtual queue

On every packet arrival:
$VQ \leftarrow max(VQ - \tilde{C}(t-s), 0)$
if $VQ + b > B$ **then**
  Mark or drop the arriving packet in the physical router queue
else
  Queue the arriving packet in the physical queue
  Update the virtual queue: $VQ \leftarrow VQ + b$
end if
$\tilde{C} \leftarrow max(min(\tilde{C} + \alpha \times \gamma \times C(t-s), C) - \alpha \times b, 0)$
$s \leftarrow t$

Figure 2.8: Pseudo code for AVQ

Parameters	Description
γ	The desired utilization at a link
α	Coefficient for computing the virtual queue capacity

Table 2.5: AVQ Parameters

$$\dot{\tilde{C}} = \alpha(\gamma C - \lambda) \tag{2.12}$$

The pseudo code for equation 2.12 and the parameters for AVQ are shown in figure 2.8 and 2.5.

Kunniyur and Srikant performed simulations in the network simulator ns-2 to compare AVQ against PI, REM, and RED. Their simulation results showed that AVQ responded faster to network suddenly changed than PI, REM, and RED while achieving similar loss rates and link utilization as the other AQM algorithms. Further, when the network load changed gradually, AVQ obtained the lowest packet losses and maintained the smallest router queue while achieving similar link utilization as the other AQM algorithms.

2.1.7 Stabilized Random Early Drop (SRED)

As in the case of RED, the Stabilized Random Early Drop (SRED) algorithm proposed by Ott et al. in 1999 attempts to realize the goals for AQM set forth in IETF's RFC 2309 such as high throughput and low queuing delay by stabilizing the queue length around a queue threshold [OLW99]. SRED stabilizes the router queue by dropping arriving packets

with a drop probability dependent on the network load. When the network load is heavy and more packets arrive than the router can forward, SRED increases the drop probability to stabilize the router queue. On the other hand, SRED reduces the drop probability when the network load is light. SRED measures the network load by monitoring the instantaneous router queue and estimating the number of active connections on a link.

In order to estimate the number of active flows, the SRED algorithm keeps the headers of recent packet arrivals in a cache table called a "zombie list". When a packet arrives, it is compared with a randomly chosen packet from the zombie list. If the two packets are of the same flow, a "hit" is declared. On the t^{th} packet arrival, $Hit(t)$ is 1 if there is a hit and $Hit(t)$ is 0 otherwise. Further, if $Hit(t)$ is 0, the packet header in the zombie list is probabilistically replaced by the header of the new packet.

Let α be a constant ($0 < \alpha < 1$), the hit frequency $P(t)$ at time t is computed as

$$P(t) = (1 - \alpha)P(t - 1) + \alpha Hit(t) \tag{2.13}$$

Ott et al. observed that a large number of active connections results in a low hit frequency and vice versa. Thus, they estimated the number of active connections as the reciprocal of the hit frequency:

$$N = \frac{1}{P(t)} \tag{2.14}$$

Ott et al. also assumed that TCP connections fairly share the buffer at the router. Given a target router buffer Q_0 and the average congestion window of a TCP connection *cwnd*, they had:

$$cwnd = \frac{Q_0}{N} \tag{2.15}$$

Further, they derived from their previous results [OKM96] that *cwnd* is inversely proportional to the square root of the packet loss rate p.

$$cwnd \sim \frac{1}{\sqrt{p}} \tag{2.16}$$

Combining 2.14, 2.15, and 2.16, Ott et al. arrived at the following equation used for determining the drop probability for arriving packets:

$$p_{zap}(q) = \begin{cases} p_{max} \times min\left(1, \frac{1}{(256 \times P(t))^2}\right) & \text{if } \frac{1}{3}B \leq q < B \\ \frac{1}{4}p_{max} \times min\left(1, \frac{1}{(256 \times P(t))^2}\right) & \text{if } \frac{1}{6}B \leq q < \frac{1}{3}B \\ 0 & \text{if } 0 \leq q < \frac{1}{6}B \end{cases} \tag{2.17}$$

where q is the instantaneous router queue, B is the maximum queue size, and p_{max} is a constant ($0 < p_{max} < 1$).

Upon packet loss (or if the router queue exceeds certain threshold) event:
if $now - last\_update > freeze\_time$ **then**
  $p_m \leftarrow p_m + \delta_1$
  $last\_update \leftarrow now$
end if
Upon link idle event
if $now - last\_update > freeze\_time$ **then**
  $p_m \leftarrow p_m + \delta_2$
  $last\_update \leftarrow now$
end if

Figure 2.9: Pseudo code for the BLUE algorithm

Although SRED stands in contrast to the original RED algorithm that does not adjust its drop probability for arriving packets based on the estimated number of active flows, SRED has the same spirit as the ARED algorithms proposed by Feng et al. and by Floyd et al. However, unlike the ARED algorithms, SRED attempts to estimate the number of active flows and uses this estimate to adjust the drop probability for arriving packets.

Ott et al. performed simulations to compare SRED with RED. They observed that as the number of connections varied between 10 and 300 connections, the router queue was independent of the number of connections with SRED. Even when the number of connections reached 1000, the router queue was only slightly increased with SRED. On the other hand, the router queue increased with the number of connections for RED. Further, RED allowed the router queue to overflow or become empty frequently while SRED managed to avoid these undesired behaviors.

2.1.8 BLUE

As in the case of the RED algorithm, the BLUE algorithm invented by Feng et al. in 2001 attempts to achieve low packet loss rates, low queuing delay, and high link utilization [FKSS01a, FKSS02]. BLUE achieves these performance goals by adapting its marking or dropping rate based directly on packet loss and link utilization rather than on the instantaneous or average queue lengths. Thus, BLUE is different from other AQM algorithms which use some information about queue length to control their marking or dropping rate. BLUE uses a single probability p_m for marking or dropping arriving packets. BLUE attempts to adapt p_m to provide appropriate feedback to the end systems. If the queue continually overflows, BLUE increases p_m to mark or drop packets more aggressively. If the queue becomes empty, BLUE decreases p_m. The pseudo code for BLUE is shown in figure 2.9. The main parameters for BLUE are shown in table 2.6. δ_1 and δ_2 are constants ($0 < \delta_1, \delta_2 < 1$ and $\delta_1 \gg \delta_2$).

Feng et al. performed simulations in ns-2 to compare BLUE and RED. Their simulation

Parameters	Description
δ_1	Incremental adjustment for p_m
δ_2	Decremental adjustment for p_m
$free\_time$	Minimum interval between two successive updates of p_m

Table 2.6: BLUE Parameters

results showed that BLUE stabilized the router queue and achieved a high link utilization and a low packet loss rate. On the other hand, RED was not effective in controlling the router queue and allowed overflows to occur frequently. Because of this, RED gave a high packet loss rate. Further, RED also gave lower link utilization than BLUE because periods of packet loss were often followed by periods of underutilization as TCP flows reduced their transmission rates.

Feng et al. also performed experiments with long-lived TCP flows on a congested 100-Mbps link in a small testbed consisting of 6 computers. They obtained experimental results that were similar to their simulation results and concluded that BLUE outperformed RED.

2.2 Approximating Fairness among Flows

Enforcing fairness among flows and providing quality of services (QoS) have been a challenging goal in networking research in the last 20 years. Early approaches to achieving this goal such as Weighted Fair Queuing (WFQ) [DKS89] or Generalized Processor Sharing (GPS) [PG93] used separate packet queues and state for each flow and serviced these queues in a round-robin manner[2]. When a queue is serviced, packets from that queue are dequeued and scheduled for transmission. Because of this, these algorithms are called *scheduling algorithms*.

An obstable for the deployment of scheduling algorithms is their implementation overhead. As links peed and the number of flows continue to grow, it becomes impractical to maintain separate packet queues and state for each flow. Newer scheduling algorithms such as Deficit Round Robin (DRR) [SV95], Core-Stateless Fair Queueing (CSFQ) [SSZ98], and Core-Stateless Guaranteed Throughput (CSGT) [KV03] achieved lower complexity by approximating the fair allocation of bandwidth for flows but still required a certain amount of implementation overhead. For this reason, a number of AQM algorithms have been proposed to approximate fairness among flows with a small overhead. In this section, I will review the prominent AQM algorithms in this category and discuss their trade-offs between complexity and accuracy.

[2]It is also possible to service queues in a weighted round-robin manner to provide certain flows with a larger allocation of bandwidth than the fair share.

Parameters	Description
w_q	Coefficient of a low-pass filter for computing the average queue size
max_p	Maximum mark or drop probability when average queue size varies between min_{th} and max_{th}
min_{th}	Low queue threshold for computing drop or mark probability for arriving packets
max_{th}	High queue threshold for computing drop or mark probability for arriving packets
min_q	Minimum number of packets each flow is allowed to have in the router queue

Table 2.7: FRED Parameters

2.2.1 Flow Random Early Drop (FRED)

Lin and Morris evaluated the effects of RED on different traffic types such as unresponsive, fragile (responsive and short-lived) and robust (responsive and long-lived) flows [LM97]. They pointed out that RED allows unfair allocation of bandwidth because it imposes the same loss rate on all flows regardless of their bandwidths and their responses to losses. They proposed the Flow Random Early Drop (FRED) [LM97] algorithm, a modified version of RED, in 1997 that provides better protection than RED for responsive (fragile and robust) flows. Further, FRED was also able to isolate and control unresponsive greedy flows better than RED.

FRED operates like RED in that it calculates the drop probability as a function of the weighted average queue. However, FRED has the following additions. The algorithm has two additional parameters min_q and max_q. These parameters specify the minimum and maximum number of packets that each flow is allowed to have in the router queue. Like RED, FRED uses a low-pass filter to calculate a weighted average queue avg. Further, FRED maintains a variable $avgcq$ as an estimate of average per-flow packets in the router queue (flows with fewer packets than $avgcq$ in the router queue are favored over flows with more). The algorithm also maintains per-flow packet counts $qlen_i$ for each flow that currently has packets in the router queue. Finally, FRED maintains a variable $strike_i$ for each flow to count the number of times a flow has failed to respond to congestion notification. The pseudo code and parameters for FRED are shown in figure 2.10 and table 2.7.

Lin and Morris performed simulations to compare FRED with RED. The first simulation was conducted with 5 TCP flows and RED on a 45 Mbps bottleneck link in a heterogeneous environment. Results of this simulation showed that long RTT flows received less than their fair shares of bandwidth under RED. Another simulation was performed with a TCP flow and a 8-Mbps constant bit rate UDP flow on a 10-Mbps congested link. Under RED, the TCP flow was not able to obtain its fair share of bandwidth (5 Mbps in this case). When

For each arriving packet P:
Calculate average queue length
Obtain connection ID of the arriving packet: $flow_i \leftarrow connectionID(P)$
if $flow_i$ has no state table **then**
  $qlen_i \leftarrow 0$
  $strike_i \leftarrow 0$
end if
Compute the drop probability like RED: $p \leftarrow max_p \frac{max_{th} - avg}{max_{th} - min_{th}}$
$max_q \leftarrow min_{th}$
if $(avg \geq max_{th})$ **then**
  $max_q \leftarrow 2$
end if
if $(qlen_i \geq max_q || (avg \geq max_{th} \&\& qlen_i > 2 * avgcq) || (qlen_i \geq avgcq \&\& strike_i > 1))$ **then**
  $strike_i \leftarrow strike_i + 1$
  Drop arriving packet and return
end if
if $(min_{th} \leq avg < max_{th})$ **then**
  if $(qlen_i \geq max(min_q, avgcq))$ **then**
    Drop packet P with a probability p like RED
  end if
else if $(avg < min_{th})$ **then**
  return
else
  Drop packet P
  return
end if
if $(qlen_i == 0)$ **then**
  $Nactive \leftarrow Nactive + 1$
end if
Enqueue packet P

For each departing packet P:
Calculate average queue length
if $(qlen_i == 0)$ **then**
  $Nactive \leftarrow Nactive - 1$
  Delete state table for flow i
end if
if $(Nactive)$ **then**
  $avgcq \leftarrow avg/Nactive$
else
  $avgcq \leftarrow avg$
end if

Figure 2.10: Pseudo code for the FRED algorithm

Parameters	Description
L	Number of bin levels
N	Number of accounting bins per level
Δ	Incremental or decremental adjustment for p_m
$free\_time$	Minimum interval between two successive updates of p_m

Table 2.8: SFB Parameters

FRED was used, the problems found in the two aforementioned simulations were corrected and approximate fair shares of bandwidths were allocated to all flows at a bottleneck link.

2.2.2 Stochastic Fair BLUE (SFB)

Feng et al. proposed in 2001 the Stochastic Fair BLUE (SFB) algorithm that achieves an approximation of fairness among flows [FKSS01b, FKSS02]. SFB achieves this by detecting and rate-limiting unresponsive flows that use a large proportion of the bandwidth. SFB maintains $L \times N$ counters called accounting bins that count the number of packets of different groups of flows that arrived at the router recently.

The accounting bins are organized in L levels, each level has N bins. SFB maintains L independent hash functions, each hash function is associated with one level of the accounting bins. The L hash functions use the addresses and port numbers of the source and destination of a packet to compute the bin index for that packet within each of the L levels. When a packet arrives, packet counters of the bins indexed by the L hash functions are incremented. Similarly, when a packet departs, packet counters of the bins indexed by the L hash functions are decremented.

For each bin, a marking or dropping probability p_m as in BLUE is maintained and updated based on the queue occupancy of that bin. If the number of packets in a bin exceeds a certain threshold, p_m for that bin is increased. If the queue occupancy of a bin becomes zero, p_m is decreased.

High-bandwidth flows are identified when the marking or dropping probability p_m of their bins become 1. These high-bandwidth flows are then rate-limited. SFB works well when the number of high-bandwidth flows is small. As pointed out by the algorithm designers, when the number of high-bandwidth flows increases, a large number of bins become occupied and low bandwidth flows that hash to these bins are incorrectly identified as high-bandwidth and penalized [FKSS01b, FKSS02]. The pseudo code and parameters for SFB are shown in figure 2.11 and table 2.8.

Feng et al. performed simulations with 400 TCP flows and one unresponsive UDP flow on a 45-Mbps congested link in ns-2 to compare SFB with RED and Stochastic Fair Queuing (SFQ) [McK90]. The transmission rate of the UDP flow was varied between 2 Mbps and 45 Mbps. Feng et al.'s results showed that SFB was able to identify and control the UDP

On every packet arrival:
Calculate hashes $h_0, h_1, \ldots, h_{L-1}$
Update bins at each level
for $i = 0$ to $L - 1$ **do**
  if $(B[i][h_i].qlen > bin\_size)$ **then**
    $B[i][h_i].p_m \leftarrow B[i][h_i].p_m + \Delta$
    Drop packet
  else if $(B[i][h_i].qlen == 0)$ **then**
    $B[i][h_i].p_m \leftarrow B[i][h_i].p_m - \Delta$
  end if
end for
$p_{min} \leftarrow min(B[0][h_0].p_m, B[1][h_1].p_m, \ldots, B[L-1][h_{L-1}].p_m)$
if $(p_{min} == 1)$ **then**
  ratelimit()
else
  Mark or drop packet with probability p_{min}
end if
On every packet departure:
Calculate hashes $h_0, h_1, \ldots, h_{L-1}$
Update bins at each level
for $i = 0$ to $L - 1$ **do**
  if $(B[i][h_i].qlen == 0)$ **then**
    $B[i][h_i].p_m \leftarrow B[i][h_i].p_m - \Delta$
  end if
end for

Figure 2.11: Pseudo code for the SFB algorithm

flow and allocated approximated fair shares of bandwidth to all flows.

2.2.3 CHOKe

The CHOKe algorithm was proposed by Pan et al. in 2000 to provide each of the active flows at a router with a fair allocation of bandwidth [PPP00]. CHOKe attempts to achieve this goal by detecting and discriminating against flows that use more than their fair share. Further, CHOKe accomplishes this goal without maintaining state and thus has a simple implementation with minimal overhead.

CHOKe computes the weighted average queue avg by using a low-pass filter just as RED and its variants do. The algorithm also has two thresholds min_{th} and max_{th} like RED. If the average queue is below min_{th}, arriving packets are allowed to entered the queue. If the average queue is above max_{th}, CHOKe (just like RED and its variants) drops all arriving packets. When the average queue is between min_{th} and max_{th}, CHOKe compares the header of an arriving packet with the header of a randomly chosen packet in the queue. If both packets belong to the same flow, both are dropped. Otherwise, the new packet is dropped with a probability p that is computed exactly as in RED.

$$p = max_p \times \frac{max_{th} - avg}{max_{th} - min_{th}} \tag{2.18}$$

Figure 2.12 shows the pseudo code for CHOKe. Further, CHOKe has the same main parameters as RED shown in table 2.1.

Pan et al. ran ns-2 simulations on a 1-Mbps link with 32 TCP flows and 1 UDP flow. They varied the transmission rate of the UDP flow to allow different scenarios where unfairness between flows could arise. Their simulation results demonstrated the RED and drop-tail were not able to discriminate against the unresponsive flows. However, CHOKe was able to drop a large portion of UDP packets and increase throughputs of the TCP flows. Pan et al. showed that this conclusion also holds for simulations with multiple congested links.

2.2.4 Approximate Fairness through Differential Dropping (AFD)

The Approximate Fairness through Differential Dropping (AFD) algorithm was proposed by Pan et al. in 2003 to approximate fair bandwidth allocations over longer time scales by using a history of recent packet arrivals to estimate flows' sending rates [PBPS03]. The design of AFD allows a relatively simple implementation of the forwarding path but requires an additional amount of memory for the recent history of arriving packets.

Packets of a flow are marked or dropped with a probability that is a function of the flow's estimated sending rate. The algorithm uses a cache table called "shadow buffer" to store recent packet headers and uses these to estimate a flow's rate. The estimated rate

```
On every packet arrival:
if avg ≤ min_th then
  Enqueue packet
else
  Draw a random packet from the router queue
  if Both packets from the same flow then
    Drop both packets
  else if avg ≤ max_th then
    Enqueue packet with a probability p
  else
    Drop packet
  end if
end if
```

Figure 2.12: Pseudo code for the CHOKe algorithm

of a flow is proportional to the number of that flow's headers in the shadow buffer. When a packet arrives, its header is copied to the shadow buffer with probability $1/s$, where s is the sampling interval, and another header is removed randomly from the shadow buffer. While sampling reduces implementation overhead, it also reduces the accuracy in estimating flows' sending rate. This problem can be severe when most flows only send a few packets per round-trip time.

AFD uses a control theoretic algorithm borrowed from PI [HMTG01] to estimate the "fair share" of bandwidth that a flow is allowed to use. Like PI, AFD periodically samples the router queue and updates the "fair share" of bandwidth. Let T be the sampling period and $q(kT)$ be the router queue at time kT , the fair share of bandwidth $r_{fair}(kT)$ is determined by AFD using the following equation:

$$r_{fair}(kT) = r_{fair}((k-1)T) + \alpha(q((k-1)T) - q_{ref}) - \beta(q(kT) - q_{ref}) \tag{2.19}$$

where q_{ref} is the target queue reference, α and β are coefficients of the PI equation discussed in section 2.1.4.

$r_{fair}(kT)$ is then used to compute the drop probability for arriving packets. If flow i has m_i packet headers in the shadow buffer, the drop probability for an arriving packet belongs to flow i is computed as follows:

$$p_i = max(0, (1 - \frac{r_{fair} \times b}{R \times m_i})) \tag{2.20}$$

where b is the size of the shadow buffer and R is the aggregate arrival rate of all flows. The main parameters for AFD are described in table 2.9.

Pan et al. used ns-2 to perform simulations and compare AFD with RED and FRED.

Parameters	Description
q_{ref}	Target queue reference for the instantaneous queue
α and β	Coefficients for "fair share" of a flow
T	Sampling interval for the instantaneous queue
b	Shadow buffer size

Table 2.9: AFD Parameters

First, simulations were run with 35 responsive flows on a 10-Mbps congested link. Next simulations were performed with 350 responsive flows on a 100-Mbps congested link. The 35 and 350 responsive flows comprise of TCP flows with different AIMD coefficients and binomial coefficients [BB01]. Finally, simulations were performed with a mix of responsive and unresponsive flows. Simulation results of Pan et al. showed that AFD obtained a fairer degree of fairness than FRED which is turn was fairer than RED to all flows. Further, AFD was more effective than FRED and RED in identifying and controlling unresponsive flows.

2.3 Controlling Unresponsive High-Bandwidth Flows

AQM algorithms discussed in section 2.1 operates on the premise that end systems are well-behaved and respond to congestion notifications (in form of packet drops or packet mark as described in section 2.5) by reducing their transmission rates. However, selfish or malicious users may choose to ignore the congestion notifications and continue to transmit data at a high rate. This can lead to situations called *congestion collapse* where a single unresponsive flow completely dominate a bottleneck link while all responsive flows such as TCP continuously reduce their transmission rates upon packet losses and are eventually starved [FF99]. In this section, I will review AQM algorithms that were proposed to control unresponsive high-bandwidth flows and prevent congestion collapse.

2.3.1 Stabilized Random Early Drop (SRED)

As discussed previously in section 2.1.7, the SRED algorithm computes a "hit" for a flow that an arriving packet belongs. Hits were used to estimate the number of active flows on a bottleneck link and calculate the drop probability for arriving packets as described in equation 2.17.

Since high-bandwidth flows are likely to have a large number of packet headers in the "zombie list" (as described in section 2.1.7) and hence experience more hits, Ott et al. pointed out in 1999 that hits can also be used to identify high-bandwidth flows [OLW99]. They proposed that the drop probability for an arriving packet should depend on the hits of the flow that the packet belongs to. The previous equation 2.17 of SRED for computing drop probability for an arriving packet is modified as follows.

$$p_{zap}(q) = \begin{cases} p_{max} \times min\left(1, \frac{1}{(256 \times P(t))^2}\right) \times \left(1 + \frac{Hit(t)}{P(t)}\right) & \text{if } \frac{1}{3}B \leq q < B \\ \frac{1}{4}p_{max} \times min\left(1, \frac{1}{(256 \times P(t))^2}\right) \times \left(1 + \frac{Hit(t)}{P(t)}\right) & \text{if } \frac{1}{6}B \leq q < \frac{1}{3}B \\ 0 & \text{if } 0 \leq q < \frac{1}{6}B \end{cases} \quad (2.21)$$

Ott et al. conducted simulations with 100 TCP flows and one high-bandwidth UDP flow on a 45-Mbps link. Their simulation results showed that the UDP flow had a significantly higher hit probability and experienced higher packet loss rate. The results also showed that SRED was able to limit the amount of bandwidth that the UDP flow consumed.

2.3.2 RED with Preferential Dropping (RED-PD)

Mahajan et al. proposed the RED with Preferential Dropping (RED-PD) algorithm in 2001 to provide protection for responsive flows [MFW01]. RED-PD keeps state for just high-bandwidth flows and preferentially drops packets of these flows. By preferentially dropping packets from the high-bandwidth flows, RED-PD limits the amount of bandwidth that they consume and improves the performance of responsive flows. Mahajan et al. argued that this approach is effective because the majority of bytes on Internet links comes from a small number of high-bandwidth flows. Hence, RED-PD only has to maintain a small amount of state.

RED-PD is based on the assumption that high-bandwidth flows also have a high number of packet drops in the RED drop history (i.e., packets that are dropped by the regular RED algorithm). The algorithm uses a history of recent packet drops to identify high-bandwidth flows and then monitors them. The history of recent packet drops is divided into M consecutive intervals called lists. A flow is identified as high-bandwidth if it has packet drops in at least K from the M lists. The length of a list is also called *an epoch* and is calculated as

$$epoch\ length = \frac{r}{\sqrt{1.5p}} \quad (2.22)$$

where p is the current drop rate and r is a target RTT of flows traversing the link. This calculation is based on the assumption that a TCP flow experiences one packet loss out of $1/p$ packets [MFW01].

After being identified as high-bandwidth a flow is monitored until it does not experience any packet drop in a certain time period. Packets of a high-bandwidth flow are dropped with a higher probability than packets from other flows. Further, packets from a monitored flow are dropped with a probability dependent on the sending rate of that flow. The absence of packet drops of a high-bandwidth flow in the drop history indicates that that flow has likely reduced its sending rate. In this case, that flow is released from the list of monitored

```
for each flow f (monitored flows that don't appear in any of the M drop lists) do
  P ← drop probability of f
  if (P > 2 × max_decrease) then
    P ← P − max_decrease
  else
    P ← P/2
  end if
  if P ≥ P_minThresh then
    drop probability of f ← P
  else
    release f from the list of monitored flows
  end if
end for
```

Figure 2.13: RED-PD's pseudo code for reducing drop probability for a flow

flows. Figures 2.13 and 2.14 show RED-PD's pseudo code that is used to adjust the drop probability for packets from monitored flows. The parameters of RED-PD are summarized in table 2.10.

Mahajan et al. conducted simulations in ns-2 to show that RED-PD was effective in identifying high-bandwidth TCP and UDP flows. Further, RED-PD needed less than 0.5 seconds to identify high-bandwidth flows. Mahajan et al. also performed simulations with a mix of TCP and UDP flows that have different round-trip times. They showed that RED-PD was able to approximate fair shares of bandwidth to all flows. On the other hand, RED was not able to allocate fair shares of bandwidth to all flows because RED does not differentiate flows based on their sending rates.

2.4 Improving Performance for Short Connections

Guo and Matta pointed out a number of issues of TCP that adversely affect the performance of short connections [GM01]. First, when a connection is being established, the loss of SYN and SYN-ACK can only be recovered through a long period of TCP timeouts. Further, after a connection is established, TCP has a conservative initial congestion window and has to resort to the RTO mechanism discussed in Chapter 1 to detect and recover loss of data packets (although the TCP congestion window is increased exponentially in the slow start phase, most short connections only have a few data packets to exchange and these connections are completed before duplicate ACKs, a more efficient mechanism than RTO, can be used to detect and recover loss of data packets). Because of these reasons, short TCP connections suffer significant performance degradation when experiencing packet loss. Guo and Matta pointed out that this is an important problem because the distributions of flow sizes of Internet traffic are heavy-tailed according to measurement studies [SCJO01, ZBPS02].

avg_drop_count: average number of drops for flows identified
p: current ambient drop rate
$p \leftarrow max_p \frac{max_{th} - avg}{max_{th} - min_{th}}$
for each flow f (flows that appear in at least K of M drop lists) **do**
  if (f is monitored) **then**
    $P_f \leftarrow$ drop probability of f
  else
    $P_f \leftarrow 0$
  end if
  $drop_f \leftarrow$ number of drops f
  $P_{delta} \leftarrow (drop_f / avg\_drop\_count) \times p$
  if ($P_{delta} > P_f + p$) **then**
    $P_{delta} \leftarrow P_f + p$
  end if
  Increase the drop probability of flow f by P_{delta}
end for

Figure 2.14: RED-PD's pseudo code for increasing drop probability for a flow

Parameters	**Description**
w_q	Coefficient of a low-pass filter for computing the average queue size
max_p	Maximum mark or drop probability when average queue size varies between min_{th} and max_{th}
min_{th}	Low queue threshold for computing drop or mark probability for arriving packets
max_{th}	High queue threshold for computing drop or mark probability for arriving packets
M	Number of drop list
K	Number of drop list above which a flow is classifed as high-bandwidth
$P_{minThresh}$	Threshold for drop probability below which a monitored flow is released
$max\_decrease$	Maximum adjustment of drop probability for a monitored flow

Table 2.10: RED-PD Parameters

2.4.1 RIO-PS

Guo and Matta designed an algorithm called RED with In and Out with Preferential treatment to Short flows (RIO-PS) in 2001 that gives preferential treatment to short flows at bottleneck links [GM01]. With preferential treatment, short flows experience a lower packet loss rate than long flows. This allows short flows to reduce the probability of experiencing timeouts and thus obtain performance improvement.

In RIO-PS, routers at the edge of a network (edge routers) maintain a table of per-flow packet counters for flows entering the network. The flow table is periodically updated. If a flow does not send any packets after an update interval T_u,, its packet counter is removed from the flow table. Edge routers use a packet threshold L_t to determine whether a flow is short or long. Packets belong to short flows are classifed as "Short" or "In". Packets from long flows are labeled as "Long" or "Out". With this method, the first L_t packets of a long flow are classified as "Short" packets. The value of L_t can be dynamically adjusted so that the ratio between short and long flows (Short-to-Long-Ratio or SLR) achieves a certain target value.

Routers inside the network (core routers) use the standard Red with In and Out (RIO) algorithm [CF98]. RIO has two sets of RED parameters $(min\_in, max\_in, Pmax\_in)$ and $(min\_out, max\_out, Pmax\_out)$ for In and Out packets. Further, the router maintains a variable avg_in for the average queue for the In packets and a variable avg_total for the total average queue for both In and Out packets. The drop probability for an In packet depends on avg_in whereas the drop probability for an Out packet depends on avg_total. The two sets of RED parameters for RIO are chosen such that Out packets are dropped more aggressively than In packets. For example, this can be achieved by having $min\_out < min\_in$, $Pmax\_out > Pmax\_in$, and $max\_out < max\_in$. Because of this, core routers drop Long or Out packets with a higher probability than Short or In packets.

Guo and Matta conducted simulations in the network simulator ns-2 to compare RIO-PS with RED and drop-tail. Simulations were run on a 100-Mbps congested link using a Web traffic model developed by Feldmann et al. for the HTTP 1.0 protocol [FHGW99]. The number of simulated Web browsing users was chosen so that the traffic they generated came close to the capacity of the bottleneck link. Simulation results showed that RED obtained better performance than drop-tail (i.e., the average response times were shorter under RED than drop-tail). When compared to RED, RIO-PS improved the average response times by 25% to 30% for small and medium object sizes (which constituted the majority of web objects).

Parameters	Description
w_q	Coefficient of a low-pass filter for computing the average queue size
$Pmax\_in$	Maximum mark or drop probability when avg_in varies between min_in and max_in
min_in	Low queue threshold for computing drop or mark probability for arriving "In" packets
$max_{\in}$	High queue threshold for computing drop or mark probability for arriving "In" packets
$Pmax\_out$	Maximum mark or drop probability when avg_total varies between min_out and max_out
min_out	Low queue threshold for computing drop or mark probability for arriving "Out" packets
max_out	High queue threshold for computing drop or mark probability for arriving "Out" packets
SLR	Ratio of short to long flows

Table 2.11: RIO-PS Parameters

2.5 Explicit Congestion Notification

AQM algorithms traditionally notify end systems of incipient congestion by dropping arriving packets at a router. Transport protocols at the end systems such as TCP infer the presence of congestion when they detect packet losses and react to these losses by reducing their sending rates. With reliable data delivery semantics (such as provided by TCP), the lost data packets have to be retransmitted. This results in decreased throughput and increased latency for applications. The IETF proposed a protocol for an explicit signaling mechanism called Explicit Congestion Notification (ECN) by using bits in TCP and IP headers [RFB01]. With ECN routers can mark a packet by setting a bit in the header (instead of dropping the packet) to deliver the congestion signal explicitly to the end systems. This approach avoids packet losses and the potential impact of packet losses on applications.

Senders indicate their ECN capability by setting the ECN-Capable Transport (ECT) codepoint in the IP header. When congestion is detected, routers mark packets that have the ECT codepoint set to convey an explicit congestion signal to the end systems. ECN marking is done by setting the Congestion Experienced (CE) codepoint in the IP header. When the receiver receives a data packet with the CE codepoint set, it sets the ECN-Echo flag in the TCP header of its next ACK packet to notify the sender of congestion in the network. Upon receiving an ACK packet with the ECN-Echo flag set, the sender reduces its congestion window as if it had lost a packet. The sender also sets the CWR flag in the TCP header of its next packet to confirm the receipt of the receiver's ECN-Echo flag.

Since an uncooperative or malicious user can set the ECT codepoint and ignores the routers' congestion signal, the standard specification for ECN recommends that routers only

mark packets when their average queue size is low. When the average queue size exceeds a certain threshold, the standard specification for ECN recommends routers to drop packets rather than set the CE codepoint in the IP header. The ARED algorithm described in section 2.1.3 follows this recommendation and drops all arriving packets when its average queue size grows larger than max_{th}.

2.6 Evaluation of AQM and ECN

Most evaluation studies on AQM and ECN have been thus far based on simulations [HMTG01, ALLY01, FGS01, PPP00, GM01, KS01, PBPS03] (details of these simulations are shown in table 2.12). While simulation is a useful tool for research to gain insights into new network protocols and mechanisms that have not been implemented or deployed yet, simulation results do not necessarily obtain realistic results. This is because when simulators are built, numerous abstractions and simplifications from real implementations have to be made. For example, many researchers pointed out in recent discussions that the TCP implementation in the widely used network simulator does not include the TCP handshake [ei05]. Since most Internet flows are short [SCJO01, ZBPS02], TCP handshake arguably has as important an impact on the performance of these flows as the actual phase of data transfer. Further, many researchers also agreed that the widely used network simulator ns-2 has numerous implementation bugs and questioned the validity of simulation results obtained from ns-2 [ei05]. Thus, simulation results could lead to inaccurate conclusions.

For the reason mentioned above, evaluation studies with a real implementation of AQM and ECN in a real network and under controlled and realistic conditions are very important. This is because results from these evaluation studies are more credible than simulations results. Hence, conclusions drawn from results obtained under realistic conditions are more convincing than those obtained from simulation results. Despite their important role, there have been only a few evaluation studies of AQM and ECN in real networks. In this section, I will review existing evaluation studies in real networks and discuss their limitations.

2.6.1 Evaluation of AQM

May et al. performed experiments to compare RED and drop-tail on a real 10-Mbps congested link. They used Chariot [Inc98], a network load generator, to generate traffic that approximated real network traffic. The synthetic traffic generated by Chariot includes FTP, HTTP, real-time audio and video traffic and was transmitted over real TCP/IP and UDP/IP implementations of Microsoft Windows NT 4.0.

May et al. concluded from their experimental results that when a small buffer size (40 packets) is used, RED does not provide any advantage over drop-tail. When a large buffer size (200 packets) is used, RED does indeed improve systems performance. In this case,

AQM algorithm	Evaluations
AFD	ns-2 simulations with 35 responsive flows on a 10-Mbps link and with 350 responsive flows on a 100-Mbps link
ARED	ns-2 simulations with 5 to 100 long-lived TCP flows on a 15-Mbps link
AVQ	ns-2 simulations with 40 to 210 long-lived TCP flows and short-lived TCP flows arriving at a rate of 10 to 50 flows per second
BLUE	ns-2 simulations with 1000 and 4000 Pareto on/off TCP on a 45-Mbps link Experiments with 1000 and 4000 long-lived TCP flows and 6 computers in a local area network
CHOKe	ns-2 simulations with 25 to 32 long-lived TCP flows and 1 to 5 UDP flows on 1-Mbps and 10-Mbps links
PI	ns-2 simulations with 16 to 400 long-lived TCP flows and 180 to 360 HTTP sessions on a 15-Mbps link
REM	ns-2 simulations with 20 to 160 long-lived TCP flows on a 64-Mbps link
RIO-PS	ns-2 simulations with Web (HTTP 1.0) sessions [FHGW99] on a 100-Mbps link

Table 2.12: Evaluation of AQM algorithms

RED can reduce packet loss rate while slighly increasing link utilization. However, May et al. pointed out that choosing good parameter settings for RED is not straightforward.

The study of May et al. has a number of limitations. First, they only investigated the performance of RED on a one-way congested link. Thus, the effects of reverse traffic such as ACK compression [ZSC91] were not considered. Further, May et al. did not emulate propagation delays and did not consider the effects of wide-area networks [NRSA01]. They also only evaluated RED. However, many (supposedly) more advanced AQM algorithms have been proposed recently but have not been evaluated yet.

Christiansen et al. performed experiments on a real 10-Mbps congested link to compare RED and drop-tail. In their experiments, web traffic was generated by emulating the behavior of browsing users from an empirical model [Mah97] and was used to drive offered loads on the congested link. The offered loads on the congested link were varied by changing the number of emulated browsing users. Further, *dummynet* [Riz97] was used to emulate propagation delays between the end systems. The delays were derived from measurement data and ranged between 7 and 137 milliseconds to represent a sample of Internet round-trip times between a given pair of machines within the continental U.S.

RED and its effects were evaluated on web performance across a range of parameter settings and offered loads. When response times for web request and response exchanges are the primary performance mesure, Christiansen et al. draw the following conclusions from their experimental results: (1) up to 90% offered loads, RED provides minimal performance

advantage over drop-tail, (2) response times for offered loads at or below 90% are not substantially affected by choosing parameter settings for RED, (3) between 90% and 100% offered loads, RED can be carefully tuned to obtain somewhat better performance than drop-tail, however, the best parameter settings for RED were not obvious and only found through exhaustive search, (4) at these high loads, there is a trade-off between improving response times and achieving high link utilization. Christiansen et al. concluded that RED provides no clear advantage over drop-tail for web response times.

The study of Christiansen et al. has a number of limitations. Like May et al., Christiansen et al. only considered RED and one-way traffic in their study. Their results were limited only to web traffic and they examined only the HTTP 1.0 protocol. Thus, the effects of the HTTP 1.1 protocol such as pipelining [NGBS+97] and general TCP traffic were not analyzed. Further, Christiansen et al. used only a uniform distribution to emulate propagation delays and did not consider the effects of general distribution of RTTs [AKSJ03] in their experiments.

2.6.2 Evaluation of ECN

Salim and Ahmed evaluated the performance of ECN using a Linux implementation of ECN at the router and the end systems [SA00]. The router also runs an implementation of RED in combination with ECN. For bulk transfers, as network load increases, ECN, when combined with finely tuned RED parameters, can help TCP connections avoid timeouts and improve the end-user throughput. For transactional transfers (emulating HTTP request/response exchanges), as network load increases, ECN increases the number of completed transactions per second.

While Salim and Ahmed's results are encouraging and demonstrate the positive effects of ECN, they are nevertheless limited because experiments were performed with a small number of flows (5-10 flows) and all flows had the same round-trip times. Further, those results were obtained with finely tuned RED parameters and the link was congested in only one direction. Another limitation is that Salim and Ahmed only reported results for RED and RED/ECN (they did not compare evaluate performance of drop-tail and other AQM algorithms). These limitations as well as the limitations of May et al. and Christiansen et al. will be addressed in the subsequent Chapters of my dissertation.

2.7 Summary

AQM has been proposed to replace drop-tail in order to achieve more effective congestion control. While all AQM algorithms attempt to achieve this common goal, many AQM algorithms have been invented for slightly different purposes such as stabilizing router queues, approximating fairness among flows, controlling unresponsive high-bandwidth flows,

and improving performance for short flows. In this Chapter, I reviewed the most prominent AQM algorithms in the aforementioned categories and discussed how these AQM algorithms were evaluated. I also reviewed the limitations of existing evaluation for AQM algorithms such as unrealistic simulations, one-way traffic and lack of synthetic general TCP traffic. These limitations will be addressed in subsequent Chapters of my dissertation.

Chapter 3

Experimental Methodology

Experimental methodology plays an important role in evaluating network mechanisms and protocols. If experimental workloads do not capture the characteristics of traffic in real networks, obtained results may not be representative for real networks and can lead to incorrect conclusions for the network mechanisms and protocols under evaluation. This Chapter describes the experimental methodology that is used to evaluate various AQM algorithms. The Chapter is structured as follows. Section 3.1 describes the setup of a laboratory network that is used to carry out the experiments. Sections 3.2 and 3.3 explain how synthetic traffic is generated and calibrated in the laboratory network. Section 3.4 describes the experimental procedure and how experimental results are collected.

3.1 Network Setup

In order to study the effects of AQM algorithms on network and application performance, I performed experiments in a laboratory network. The network setup is shown in Figure 3.1. The network emulates a peering point between two ISPs where traffic flows on the links connecting the two routers. Although the network has a simple dumbbell topology, an extended version of *dummynet* for per-flow delays (as discussed below) and different distributions of round-trip times are used to emulate the effects of wide-area networks. Furthermore, congestion on both links between the routers creates realistic network effects such as loss of ACKs and ACK compression that may not be captured in simulations. These effects allow to obtain realistic experimental results as if the experiments were performed in the Internet. For example, ACK compression can cause a TCP sender to release multiple data packets in a short interval and generates bursty traffic behavior that puts stress on the routers.

The traffic that drives experiments for evaluating various AQM algorithms is either Web or general TCP traffic. The traffic is generated by 22 end systems on each side of the network (44 machines total) The details for generation of Web and general TCP traffic are

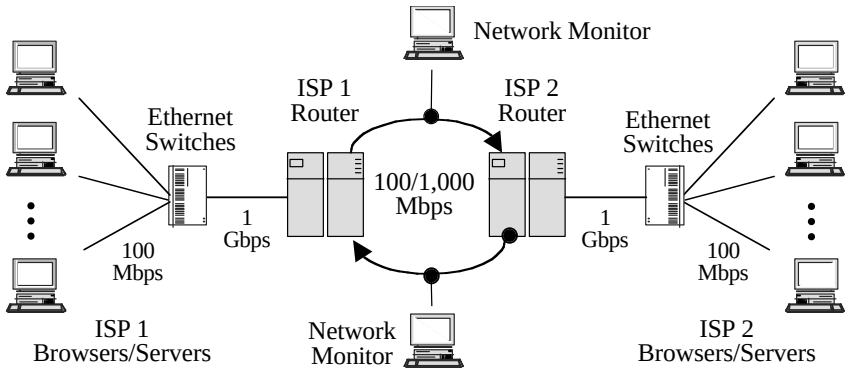

Figure 3.1: Network setup.

given in sections 3.2.1 and 3.2.2.

All systems shown in Figure 3.1 are Intel-based machines that run FreeBSD 4.5. The end systems have 100 Mbps Ethernet interfaces and are attached to 100 Mbps ports on a set of Ethernet switches. These ports are grouped to VLANs which are connected to the routers via 1 Gbps links. The routers are PCs with a 1 GHz Pentium III and over 1 GB of memory. The AQM algorithms are implemented in the FreeBSD kernel of the router machines using the framework and utilities provided by ALTQ [Cho98]. ALTQ is a software package that allows to implement queuing disciplines and algorithms for traffic management in the FreeBSD kernel.

The routers are connected to each other via 1 Gbps and 100 Mbps links. When routers are connected at 1 Gbps, static routes are configured so that traffic flows on the full-duplex gigabit Ethernet links and there is no bottleneck in the network. When routers are connected at 100 Mbps, static routes are reconfigured so that traffic in each direction uses separate 100 Mbps links and emulates the full-duplex behavior of typical wide-area network links. In this case, the 100 Mbps links form a bottleneck in the network. The 100 Mbps links are used for congested experiments to evaluate different AQM algorithms while the 1 Gbps are used for uncongested experiments. The uncongested experiments are performed to establish the baselines for comparisons with the results obtained from the congested experiments with various AQM algorithms. The uncongested experiments are also performed to calibrate the nominal loads on the links between the router machines. During all experiments, utilization of the links between the routers is measured by running a modified version of *tcpdump* program on the monitoring machines connected to these links as shown in Figure 3.1.

An extended version of *dummynet* [Riz97] developed at UNC is used to emulate different minimum round-trip times for each TCP connection. This is done by delaying packets for a certain amount of time in the TCP/IP stack of the end systems. All packets of a flow

experience the same amount of minimum delay that is randomly and uniformly chosen from a distribution that is an input parameter to the experiments. The round-trip times experienced by a connection is the sum of the delay induced by *dummynet* and additional queuing delays incurred at the router machines. End systems are so configured that they are not the bottleneck in all experiments and only cause minimal delays (approximately 1 millisecond).

The two monitoring programs were used to collect network statistics during experiments. One program monitors the router interfaces and collects the statistics of the number of forwarded and dropped packets at each router interface. Another program runs on link-monitoring machines that are connected to the links between the routers (through hubs on the 100 Mbps segments or fiber splitters on the Gigabit link). This program uses a locally-modified version of the tcpdump utility and monitors the utilization on the links between the routers by capturing the TCP/IP headers in each frame traversing the links and processing them in real-time. The program produces a log of throughput of traffic traversing the links between the routers over chosen intervals (typically 100 milliseconds).

The rest of this Chapter is organized as follows. Section 3.2 discusses how synthetic traffic is generated in the testbed network. Section 3.4 describes the procedures for running experiments.

3.2 Synthetic Traffic Generation

Two types of synthetic traffic are used in this study: web traffic and general TCP traffic. Since Web is an interactive application where end-user response times are a key performance measure, Web traffic is used to test the whether existing AQM algorithms can realize the goals of AQM stated in IETF's RFC 2309 [BCC+98]. The general TCP traffic models the source-level characteristics of the mix of TCP connections commonly found on Internet links including HTTP, FTP, SMTP, NNTP, text messaging, and peer-to-peer file-sharing traffic. Since most current Internet applications run on top of TCP, the general TCP traffic goes beyond traffic generated by web application and accounts for traffic of most Internet applications. Both models are derived from measurements of actual Internet links and hence represent the characteristics of Web and general TCP traffic as seen by routers in real networks. For this reason I believe that these models, combined with the setup of the testbed network, allows a realistic environment for controlled experiments in evaluating AQM algorithms.

3.2.1 Web Traffic Generation

The Web traffic that drives the experiments is based on a model derived from a large-scale analysis of web traffic [SCJO01]. The model is an application-level description of

Element	Description
Request size	HTTP request length in bytes
Response size	HTTP response length in bytes (top-level & embedded)
Page size	Number of embedded objects per page
Think time	Time between retrieval of two successive pages
Persistent connection use	Number of requests per persistent connection
Servers per page	Number of unique servers used for all objects in a page
Consecutive page retrievals	Number of consecutive pages requested from a given server

Table 3.1: Elements of the HTTP traffic model

the behavior of web browsing users and consists of a collection of empirical distributions necessary for generating realistic web traffic. The model is instantiated on the end systems and used to generate traffic in the testbed network by emulating the behavior of a large collection of web browsing users. The web model has two main parts: a client that generates requests and a server that responds with web objects to the requests. The models dictates how the requests and responses are generated as well as the interarrivals of the requests.

Among other elements, the model characterizes the use of persistent HTTP connections as currently implemented by many browsers and servers. Persistent HTTP connections can be used to exchange multiple requests and responses between a client and a server (unlike persistent HTTP connections, non-persistent connections can be used to exchange a single pair of request and response). The web model also distinguishes between "top-level" objects (typically an HTML file) and embedded objects (e.g., an image file). The most important elements of the model are summarized in Table 3.1. Most of these elements are implemented in the client-side request-generating program (the "browser" program).

14 machines on each side of the network are used as client machines and run the client-side program of the model (the browser). Each machine running the client-side program is called a client and emulates the aggregate behavior of a specified number of browsing users. The network load on the links between the routers is a function of both the total number of browsing users spread across all client machines and the distributions of round-trip times emulated by *dummynet*. The total number of simulated users and the distributions of round-trip times are input parameters of the controlled experiments. However, the number of browsing users is the main parameter that is used to control the offered load on the router links.

During experiments, the browsers generate requests and fetch web objects from server programs running on 8 server machines on the other side of the network. Each instance of a browsing user is modeled as a simple on/off process and implemented as a simple state machine switching between two states: "thinking" (idle) and requesting. When a browser is in the requesting state, a request is created for the primary object of a web page (i.e., the index of a web page) and sent to a server on the other side of the network. After the

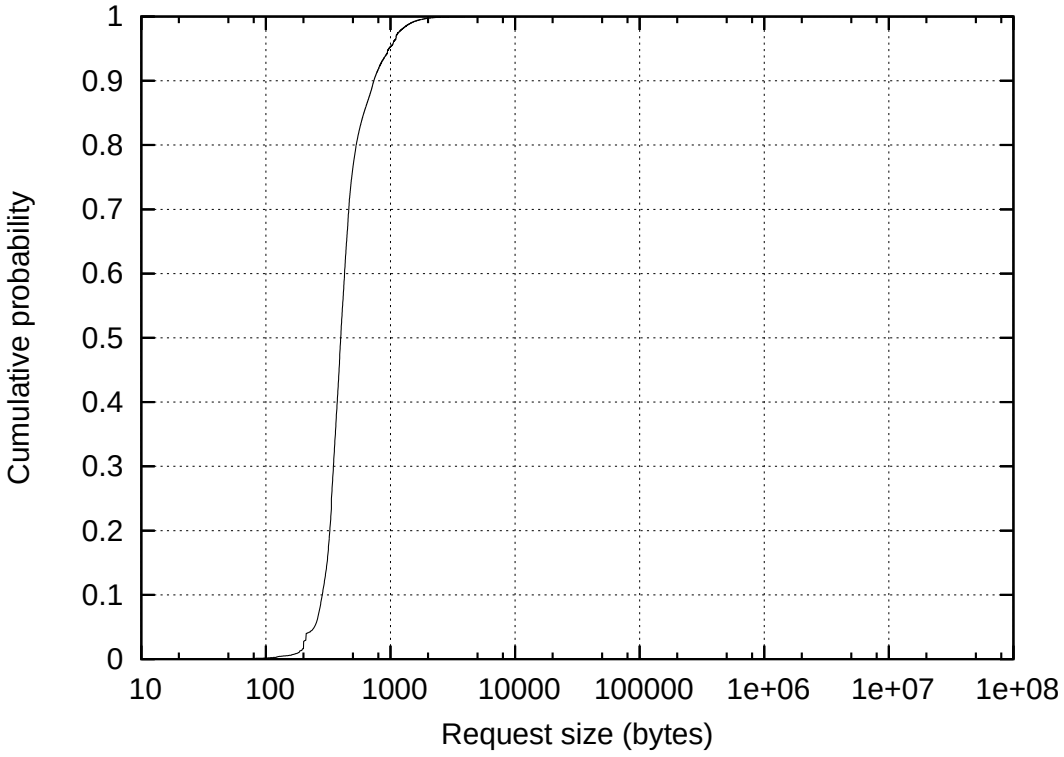

Figure 3.2: CDF of request sizes

primary object is received, the browser sends requests for embedded objects to a number of servers.

The number of servers that a browser contacts per web page and the number of embedded objects per web page are randomly chosen from their appropriate empirical distributions. The browsers use both persistent and non-persistent connections to connect to the servers. The number of web objects exchanged via a persistent connection is also randomly selected from an appropriate distribution. Approximately 15% of all connections are effectively persistent (i.e., they are used to request more than one object) but over 50% of all objects (40% of bytes) were transferred over these connections.

For each request, the browser creates a request of random size (sampled from the distribution of request sizes) and sends it to a server on the other side of the network. The request contains the size of the web object (sampled from the distribution of response sizes) to be returned to the client by the server. The distributions of of request and response sizes are shown in 3.2 and 3.4.

A primary measure of performance in evaluating the effects of various AQM algorithms on application performance is the distribution of response times for the exchanges of requests and responses as observed by a client. The response time for a request-response exchange is defined as the elapsed time between the time the request is sent and the time the entire

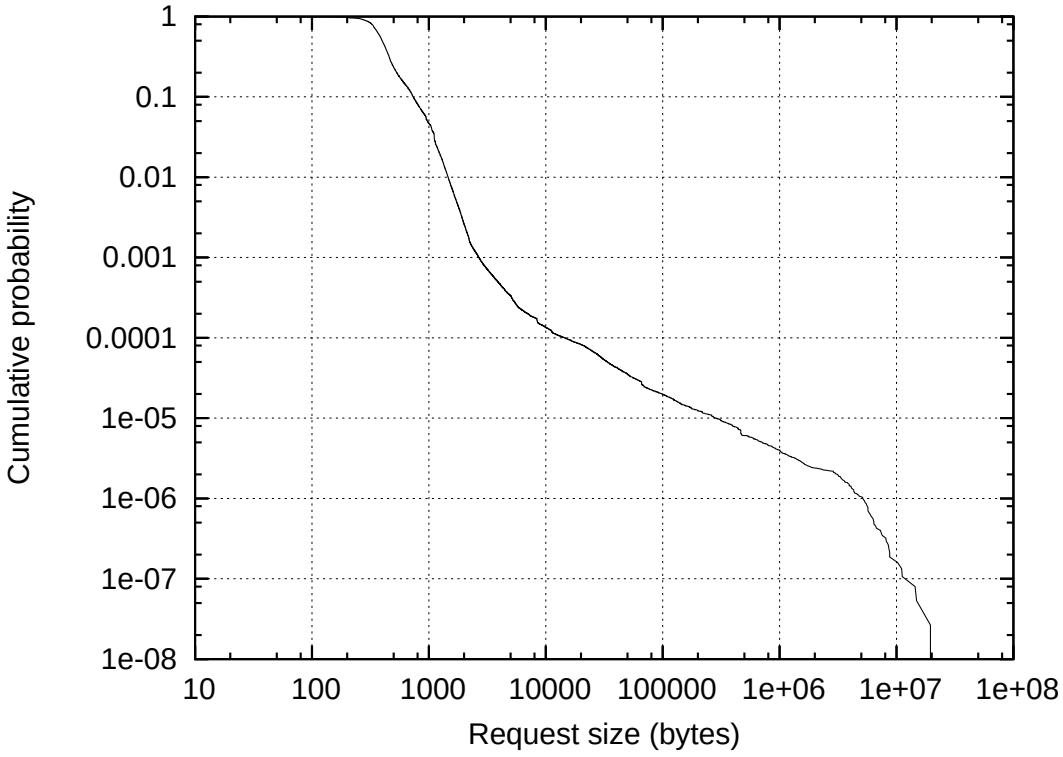

Figure 3.3: CCDF of request sizes

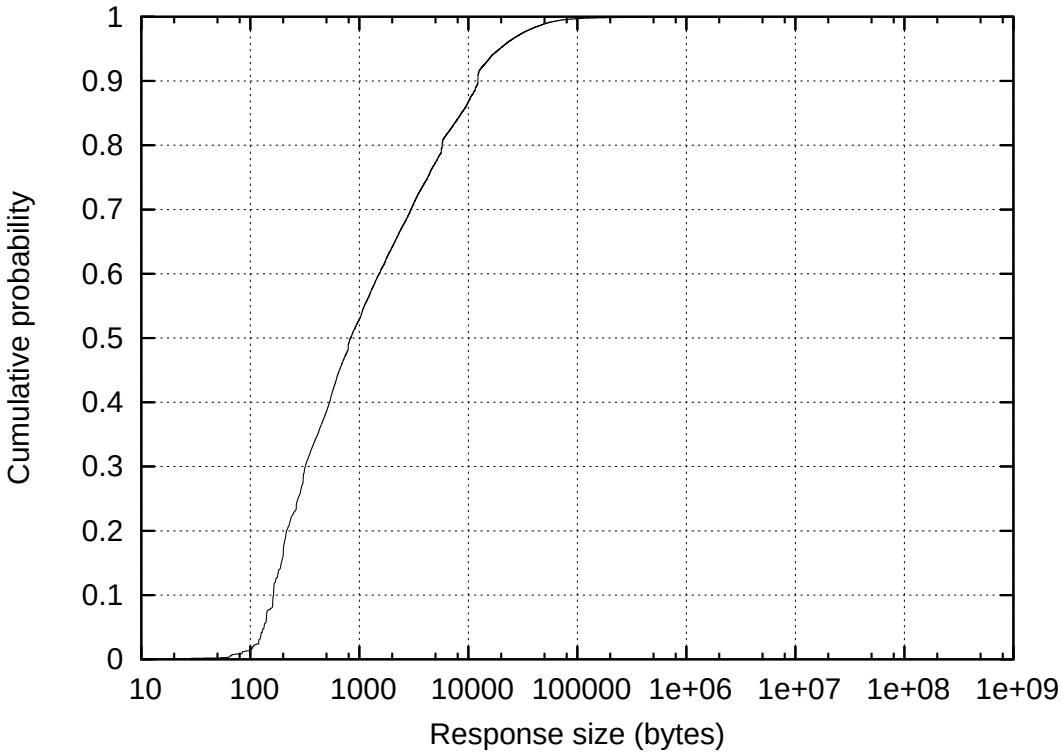

Figure 3.4: CDF of response sizes

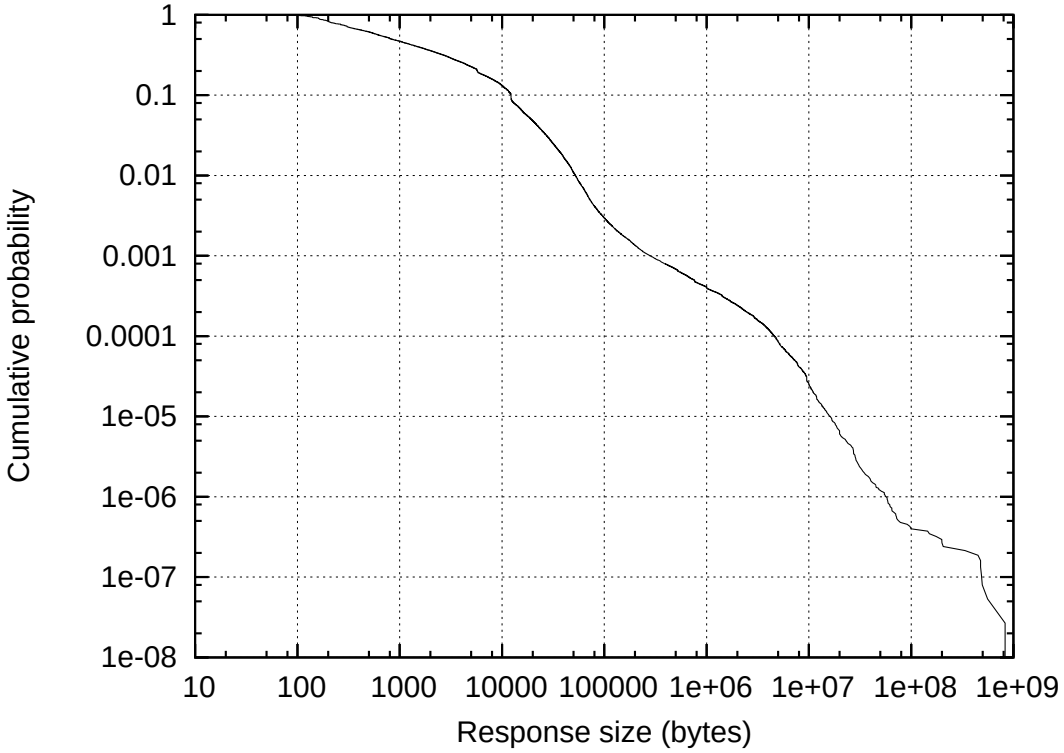

Figure 3.5: CCDF of response sizes

response is received by the client. For the first request-response exchange of a connection, the response time also includes the elapsed time for the TCP connection establishment, i.e., the response time is defined as the time interval between the client's socket call connect() to connect to a server and the client's socket call read() to retrieve the response from the server. For request-response exchanges after the first exchange has been completed via persistent connections, response times are defined as the time interval between the client's socket call write() to send a request to server and the client's socket call read() to retrieve the response from the server.

3.2.2 General TCP Traffic Generation

The model discussed in section 3.2.1 is used to generate an application-specific workload (HTTP in this case). Construction of such a model requires detailed knowledge of the HTTP protocol and its usage by contemporary implementation of the protocol (browsers and servers). Even then, the construction requires a significant amount of work. Since Internet links can carry traffic of many different applications, some of them using closed protocols, it is not feasible to generate realistic general TCP traffic by constructing application-specific models for every application model. A novel approach to generating general TCP traffic is based on an abstract model of communication patterns within a TCP connection. General

TCP traffic is modeled by an aggregation of TCP connections with application-specific communication patterns that are derived from actual Internet packet traces.

They key idea is to generate traffic by emulating the behavior of applications at the socket layer and running the traffic generators layered over real TCP implementations. This closed-loop form of traffic generation is necessary for evaluating the effects of AQM on TCP congestion control mechanisms and the performance of TCP applications (which cannot be done with a simple open-loop packet-level trace replay). The synthetic traffic is generated in such a way that it is statistically similar to the traffic on the measured links (e.g., the distributions of packet sizes, object sizes, active connections per second, throughput per second, etc. observed on the real network can be reproduced in the laboratory network). The technique for reproducing the mix of application traffic seen on real networks is called Source-Level Trace Replay [CJS04].

The data source used for deriving realistic TCP workloads and evaluating the effects of various AQM algorithms on application performance was a packet trace acquired from the NLANR repository. This packet trace was chosen to represent real network traffic and its characteristics on the Internet. The packet trace was filtered for all TCP connections including HTTP, FTP, SMTP, NNTP, text messaging, and peer-to-peer file-sharing traffic. This technology is capable of deriving and generating realistic TCP workloads from packet header traces captured on Internet links. The synthetic traffic mix generated in the network testbed represents the characteristics of existing Internet traffic as seen by routers in real networks and allows the most realistic method for network protocols and mechanisms in a laboratory network.

The key idea of source-level trace replay is to characterize and reproduce the application-level communication patterns based on the two endpoints of a TCP connection exchanging data in units defined by their specific application-level protocol. These communication patterns are specific to the applications and are derived from the packet traces by a set of analysis tools [CJS04, SCJO01]. The derived patterns characterize the number of exchanges of application data units (ADUs) between two end points of a connection and the sizes of the ADUs. The patterns also allow for possible idle times between two consecutive exchanges of application data units within a connection. The idle times are, for example, think times in the case of web applications when a user spends time reading a web page that he/she is visiting. The idle times, sizes of ADUs, and the number of exchanges of ADUs are network-independent and represent the characteristics of applications that generate the traffic captured in the packet traces. Further, the analysis tools also extract the minimum delays for each TCP connection from the packet header traces. These minimum delays are used as parameters by the *dummynet* mechanism, as described above, to emulate per-flow propagation delays in the laboratory network.

After the above characteristics for connections are derived from actual packet traces,

they can be applied to generate workloads of general TCP traffic for evaluating the AQM algorithms. During the replay, each TCP connection is reproduced as a sequence of socket calls for the data unit exchanges and sleep intervals for the think times. The connections are initiated at the same instant and the same order as they appear in the original trace.

3.2.3 Modeling Propagation Delays

In order to obtain experimental results as if the experiments would have been performed on a real wide-area network, I used the *dummynet* software described in section 3.1 to emulate propagation delays between end systems in the laboratory network. Two distributions of round-trip times were used for the experiments in the laboratory network to study the role of round-trip times (in combination with various AQM algorithms) on Web traffic.

The first distribution of round-trip times is uniform and is used to generate uniformly random delays between 10 and 150 milliseconds. This range of delays is is used to approximate typical Internet round-trip times in the continental U.S. and is selected to compare the results obtained in this study with those reported in [CJOS01].)

The second distribution of round-trip times was obtained from a measurement study [AKSJ03]. This distribution of RTTs was used as an input by the *dummynet* software to model a more general and diverse RTTs between the end systems in the laboratory network.

The two distributions of round-trip times are shown in Figures 3.6 and 3.7. While the uniform distribution could approximate the body of the general distribution up to 60%, it cannot capture the characteristics in the tail of the general distribution of round-trip times. Thus, the two distributions of round-trip times allowed to study the sensitivity of AQM performance to round-trip times and the effects of of diversity in distributions of round-trip times. Results for AQM algorithms with the two distributions of round-trip times are given in Chapters 4 and 5.

3.3 Experiment Calibrations

3.3.1 Calibrations for Web traffic

A primary simulation parameter of experiments with the web traffic is the number of emulated browsing users. This parameter allows to change the nominal load on the links between the routers. Before actual experiments with various AQM algorithms are performed, this parameter needs to be calibrated to establish a relationship between the number of emulated browsing users and the throughput of traffic that these users generate on the links between the routers.

Calibration is performed by removing the bottleneck links between the routers and running uncongested experiments on the 1-Gbps links. After calibration is done, the offered

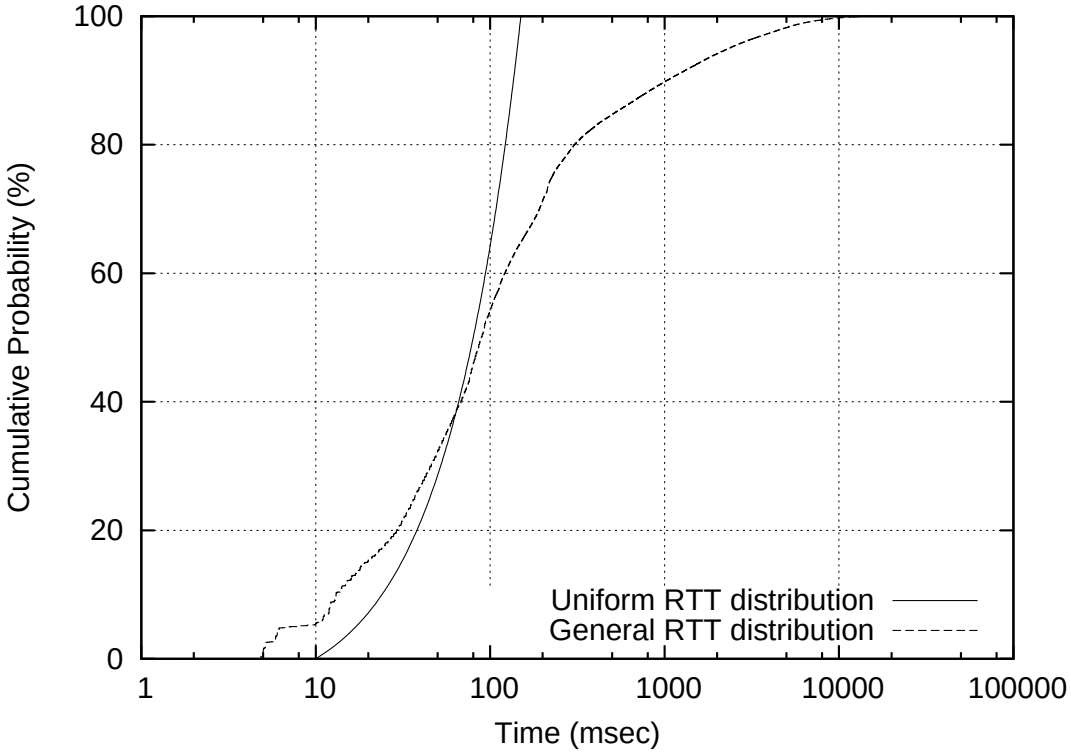

Figure 3.6: CDF of the general RTT distribution

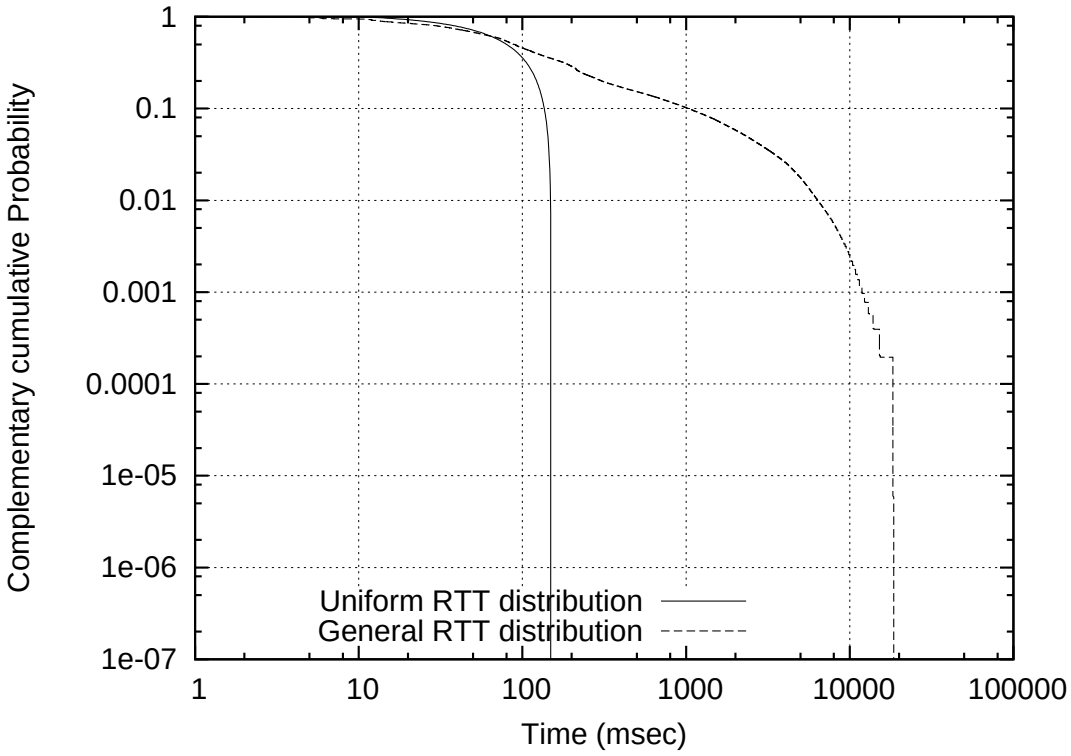

Figure 3.7: CCDF of the general RTT distribution

load on the bottleneck links can be predictably controlled by setting the size of the emulated browsing user population. Calibration experiments also ensure that there is no bottleneck on the end-to-end path in the network other than the links between the two routers when they are configured at 100 Mbps, e.g., end systems are not the bottleneck of the experiments.

For the calibration experiments, the router links are configured to run at 1 Gbps to avoid congestion in the network. The routers are configured to run with large drop-tail queues to sure that there are no packet drops in the network. The total number of emulated browsers is fixed in each calibration experiment. Since throughput is a function of round-trip time [PFTK98], calibration needs to be performed for both the uniform and general distribution of round-trip times. Figure 3.8 shows the long-term average throughput in one direction of the 1 Gbps link between the routers when the total number of browsers is varied between 7,000 and 35,000 over several experiments. The average throughput measured in the opposite direction was essentially the same and was not shown in the figure.

Since the link throughput can be approximated very well by a linear function and can easily exceed the capacity of a 100 Mbps link, there are no resource limitations in the network and the end systems. The linear relationship between the link throughput and the number of emulated browsing users is also used in subsequent congested experiments to determine the number of browsers to be emulated for a desired offered load. For example, in order to generate an offered load of 98 Mbps (98% of a 100 Mbps link) with the uniform distribution of RTTs, Figure 3.8 shows that approximately 9,520 browsing users need to be emulated on each side of the network.

3.3.2 Calibrations for General TCP Traffic

The degree of congestion induced in a network via source-level trace replay is a function of the load on the original traced network and the capacity of the target network. Two methods can be used to scale the offered loads of general TCP traffic. One method is to expand or compress the idle times of connections in the packet traces to reduce or increase offered loads. A second method is to scale offered loads via a connection subsampling or superimposing process that decreases or increases the nominal load from the original trace while preserving the mix and statistical characteristics of the TCP connections [CJS04].

The above scaling processes were used to achieve offered loads of 80, 90, and 95 Mbps on an uncongested 1-Gbps network. These loads are termed 80%, 90%, and 95% because they represent the throughputs that could be possibly achieved on a 100-Mbps link. However, note that as the offered loads approach the saturation of a 100-Mbps link, the actual link utilization will be less than the intended offered load. This is because as the network becomes congested, TCP dynamics will regulate the transmission rates of the end systems.

An interesting aspect of using actual packet traces to drive controlled experiments in a laboratory network is that Internet traffic is usually asymmetrical between forward and

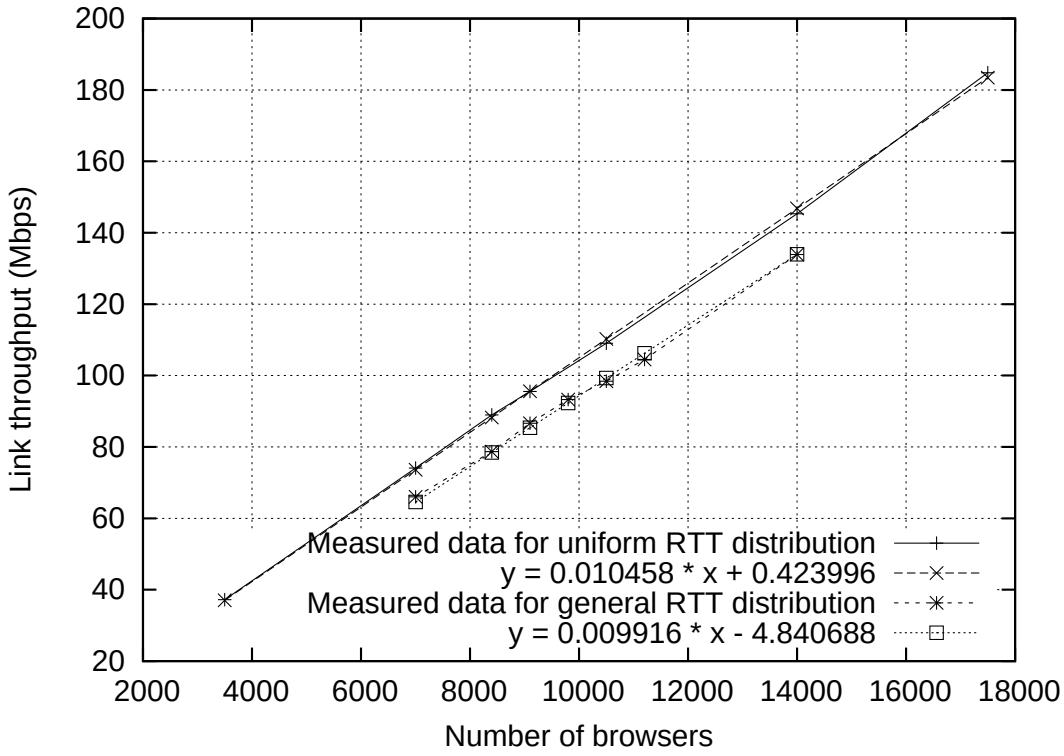

Figure 3.8: Link throughput as a function of emulated browsing users.

reverse path. For example, the 95% offered loads derived from the packet trace acquired from the NLANR repository were approximately 95.73 Mbps (forward path) and 90.70 Mbps (reverse path).

3.4 Experimental Procedures

The procedure for running experiments is as follows. First the routers and end systems are initialized and configured with their parameters. In case of web traffic, the server programs are started followed by the browser programs. Each browser program runs an equal number of browser instance, as described previously, to collectively generate a specified offered load on the network based on the calibration perform in section 3.3. The offered load for the experiments with web traffic is defined as the long-term average throughput of network traffic on an uncongested network. The offered load is controlled by using the web traffic generators to emulate a fixed number of browsing users in each experiment. The offered loads in the experiments are chosen to be 80%, 90%, 98%, and 105% of the capacity of the 100 Mbps links between the routers (i.e., the long-term average throughput of the generated traffic would be 80, 90, 98, and 105 Mbps on the uncongested 1 Gbps links). Each experiment is run for 120 minutes but data is collected only for 90 minutes to eliminate the startup effects at the beginning and the termination synchronization anomalies at the end.

In case of general TCP traffic, traffic generation programs are started after the routers and end systems are initialized and configured. The programs reproduce application-level communication patterns extracted from packet traces as described in 3.2.2. The offered loads of general TCP traffic is controlled as described in 3.3.2.

The key performance metrics of the experimental results is the end-to-end response times for the request/response exchanges reported as plots of cumulative distributions up to 2 seconds. Other performance measures such as link utilization, packet loss rate, and the number of completed request/response exchanges are also reported.

3.5 Summary

Creating a realistic network environment plays a crucial role in evaluating network protocols and mechanisms. Experimental results obtained under unrealistic conditions could lead to inaccurate conclusions about the network protocols and maechanisms under evaluation. This Chapter describes the experimental methodology that is used in my dissertation research. In particular, network setup, synthetic traffic generation, modeling of round-trip times, calibration of experiments, and experimental procedure were discussed in details.

Chapter 4

Results with Web Traffic and Uniform RTT Distributions

This Chapter presents experimental results for different AQM algorithms with Web traffic. The motivations for performing experiments with Web traffic are twofold. First, Web is arguably the currently most important Internet application for both business and private users. Hence, the effects of AQM algorithms on Web applications should be investigated before AQM algorithms are deployed. In particular, an overlooked aspect of network performance is response times for exchanges of requests and responses. Thus, studying AQM in the context of the Web gives us a useful and important basis for assessing the impact of AQM on user-centric measures of performance. Second, as discussed in Chapter 3, the model that is used to generate web traffic shows heavy-tailed distributions for both think times (OFF times) and response sizes (ON times). Thus, the aggregate traffic generated by a large collections of emulated browsing users should have the behavior of long-range dependence (LRD) processes [WTSW97] and can be considered as "stress tests" for the AQM algorithms (Indeed, the LRD behavior was verified by performing. experiments on an uncongested network, collecting the TCP/IP packet headers, and deriving a time series of the number of packets and bytes arriving at the routers in 1 millisecond intervals. The estimated Hurst parameter of the time series lies between 0.8 and 0.9 and indicates the LRD behavior of the synthetic web traffic.) I will first focus on studying the effects of AQM on the Web in this Chapter and in Chapter 5. Chapters 6 and 7 will give results for similar experiments using a more general traffic model.

In addition to studying the impact of AQM on Web traffic, I also seek to understand the role of round-trip times on AQM performance. In this chapter, I consider the effects of a uniform distribution of round-trip times between 10 and 150 milliseconds. (This range of random delays approximates typical Internet round-trip times in the continental U.S. and was selected to compare the results obtained in this study with those reported in [CJOS01].) Results for a more general distribution of round-trip times will be given in Chapter 5.

As discussed in section 3.2.1, the key performance measure that is used to evaluate the effects of various AQM algorithms on web performance is the distribution of response times for the exchanges of requests and responses as observed by a client. The distribution of response times is presented in plots of cumulative distribution function (CDF). These plots are combined with other performance measures such as link utilization, packet loss rates, and the number of completed exchanges of requests and responses within an experiment reported in table 4.1, to form the basis in evaluating the effects of AQM algorithms on Web performance. Further, the 50th, 75th, and 90th percentiles of response times from all experiments are given in table 4.2 to allow a quantitative discussion of experimental results.

Experimental results for various AQM algorithms presented in this Chapter were obtained by using the recommended parameter settings by their inventors. For PI, REM, and ARED, experiments were performed with target queue lengths of 24 and 240 packets. The target queue length of 24 packets is chosen to yield a small queuing delay. On the other hand, the target queue length of 240 packets can potentially provide good packet buffering and achieve high link utilization. For all AQM algorithms, the maximum queue size is sufficiently large so that tail drops do not occur.

The rest of this Chapter is structured as follows. Section 4.1 presents the results for drop-tail queues that are used as base results in comparison with experimental results obtained with various AQM algorithms. The results are used in all subsequent chapters as a baseline for comparison with AQM performance. Comparisons against results of drop-tail form the basis for conclusions about absolute viability of AQM. If AQM is to be of any use, it must consistently (or ideally significantly) improve upon performance obtained with drop-tail. Sections 4.2 and 4.3 show the results for ARED, PI, REM, BLUE, and AVQ when they are used with packet drops. Section 4.4 demonstrates the effects of balancing queuing delay and packet loss rates using an algorithm that I developed call LQD [LJS06]. Section 4.5 presents experimental results for the effects of the Explicit Congestion Notification signaling protocol. Section 4.6 demonstrates the effects of measuring router queues in bytes rather than packets in AQM algorithms in a case study for the ARED algorithm. Section 4.7 presents another case study for the ARED algorithm that studies the effects of dropping packets in ECN mode. Section 4.8 demonstrates the effects of differential treatment of flows using another algorithm that I developed called DCN. Sections 4.9 and 4.10 present a comparison of all results and a summary of the Chapter.

4.1 Results for Drop-Tail

In order to assess the effects of various AQM algorithms on web performance, some base results need to be established first. These base results are obtained with drop-tail FIFO queues and are used as a baseline to demonstrate the advantages (or disadvantages) of AQM

algorithms.

The sizes for drop-tail queues for the experiments were chosen by following guidelines discussed on the IRTF *end2end-interest* mailing list. Discussions on this list concerning queue sizes roughly converged around the heuristics of maintaining a queue capable of buffering 100 milliseconds worth of packets at the link capacity. Since the capacity of the bottleneck link in experiments performed in this dissertation is 100 Mbps and the average packet size is about 500 bytes, the drop-tail queue should have about 2,400 elements for 100 milliseconds of buffering. To further explore the parameter space, experiments were also performed with a drop-tail queue of 24 and 240 packets. The shorter drop-tail queues of 24 and 240 packets were chosen to study the effects of a (potentially) underprovisioned queue (1% and 10% of the recommended queue). For comparison purposes, experiments were also performed on an uncongested network. The results obtained on an uncongested network represented the best possible results that AQM algorithms could have.

Figures 4.1, 4.2, 4.3, and 4.4 show the distributions of response times of experiments with drop-tail queues of 24, 240, and 2,400 packets at 80%, 90%, 98%, and 105% load. Futher, Figures 4.5, 4.6, 4.7, and 4.8 show the tails of these distributions for completeness.

At 80% load, all drop-tail queues obtain good response time performance as they come very close to achieving the performance of the uncongested network. In addition, queue length does not have any obvious impact on response times at this load. The drop-tail queues of 24, 240, and 2,400 packets achieved the same 50th, 75th, and 90th percentiles (0.137, 0.237, and 0.362 seconds respectively). Further, these values only deviate slightly from the values obtained on the uncongested network (0.137, 0.237, and 0.312 seconds at the 50th, 75th, and 90th percentile of response times). As the queue length was increased from 24 to 240 and to 2400 packets, the packet loss rate was reduced from 0.2% to 0.0%. A packet loss rate of 0.0% is an indication that the queue never overflowed in the experiment. This result implies that a queue of 2,400 packets is overprovisioned for 80% offered load.

At 90% load, there is a considerable degradation of performance in terms of response times. For example, the 50th percentile of response times for a drop-tail queue of 240 packets increased from 0.137 to 0.187 seconds as the offered load rose from 80% to 90%. Further, it can be observed in Figure 4.2 that over 80% of flows experience better response times with a queue size of 24 or 240 packets than with a queue size of 2,400 packets and complete in less than 500 milliseconds. However, a drop-tail queue with 2,400 packets gives better response times for flows that need more than 500 milliseconds to complete (approximately 20% of all flows). A drop-tail queue of 24 packets delivers approximately the same performance as a drop-tail queue of 240 packets for about 80% of flows but is slightly inferior to a drop-tail queue of 240 packets for the rest 20% of flows.

These results show the trade-off between small queues (24 and 240 packets) and large queues (2,400 packets). A small drop-tail queue obtains small queuing delay but incurs a

high packet loss rate (for example, the packet loss rate was 1.8% for a queue of 240 packets but only 0.1% for a queue of 2,400 packets at 90% offered load). Because of this, flows that do not experience any packet loss enjoy good response times with a small drop-tail queue, especially those that are small. On the other hand, a large drop-tail queue can reduce the packet loss rate but subjects flows to more queuing delay. Hence, for large flows that dominate the links (in terms of number of packets) and are likely to experience some packet losses (assuming that packets are dropped randomly and uniformly), the impact of increased queuing delay is outweighted by the effects of reduced packet losses. Thus, large flows receive better performance under a large drop-tail queue. Results shown in table 4.2 demonstrate this phenomenon quantitatively. For example, the 75th percentile of response times was 0.312 seconds for a queue of 240 packets and 0.412 seconds for a queue of 2,400 packets. However, the 90th percentile of response times was 1.037 seconds for a queue of 240 packets and 0.612 seconds for a queue of 2,400 packets.

While the distinction between small and large flows are subjective and are often blurred, this trade-off between achieving good performance for small and large flows always exists due to the fundamental trade-off between achieving low queuing delay and low packet loss rates. Thus, there is no single queue size that can obtain best performance for all flows.

It is also interesting to note that a queue of 2,400 packets can significantly reduce the packet loss rate (from 2.7% for a queue of 24 packets and 1.8% for a queue of 240 packets to merely 0.1%) but did not increase the link throughput at 90% offered load (all queues obtained approximately a link throughput of 90 Mbps at this load). This result implies that an offered load of 90% represents severe congestion for the network where increasing queue length does not result in improved link throughput.

At 98% and 105% loads, the performance degradation is even more significant for all queue sizes. For example, the 75th percentile of response times for a queue size of 240 packets increased from 0.312 seconds at 90% offered load to 1.137 seconds and to 1.363 seconds at 98% and 105% offered loads. Loss rates also increased sharply. The loss rate for a queue size of 2,400 packets increased from 0.1% at 90% offered load to 3.6% and 7.9% at 98% and 105% offered loads. Although a drop-tail queue of 240 packets gives better response times than the other drop-tail queues for most of the flows, it is worth noting that the performance degradation for all queue sizes at 98% and 105% loads is very significant in terms of response times and packet loss rates. Thus, it appears that there is little incentive to operate a network at these high loads using drop-tail queues.

It is also interesting to note that the CDF lines for all drop-tail queues in Figures 4.3 and 4.4 were flat between 500 and 1,000 milliseconds at 98% and 105% offer loads. This means that at these high offered loads, most flows either completed within 500 milliseconds or needed more than 1 second to complete. The reason for these flat lines can be explained as follows. It can be observed from Figures 3.2 and 3.3 that 99% of HTTP requests are

smaller than 1,500 bytes and can fit in a single packet. Further, it can be observed from Figures 3.4 and 3.5 that approximately 90% of HTTP responses are smaller than 10,500 bytes and can be transmitted in 7 packets. Assuming that a TCP sender (the server in this case) starts out with a congestion window of one packet and increases the congestion window by one packet for each ACK that it receives, it would take the sender 3 round-trip times to transmit 7 packets. As discussed previously, the maximum "emulated" propagation delay in these experiments is 150 milliseconds. Further, as will be discussed in Chapter 7, these experiments ran with a link-level buffer of about 250 packets that corresponded to an approximate queuing delay of 10 milliseconds (given that the average packet size is 500 bytes). Hence, three round-trip times for 90% of responses (those that fit in 7 packets) take at most 480 milliseconds, close to the observed value of 500 milliseconds.

Further, since the initial retransmission timeout in FreeBSD is 1 second and most flows were small (as discussed above, 90% of flows only needed about three round-trip times or approximately 500 milliseconds to complete), it can be inferred that flows completing within 500 milliseconds did not experience any packet loss and their response times were mostly a function of object sizes, propagation and queuing delay. Flows that needed more than 1 second to complete likely experienced at least one packet loss and their response times were dominated by time-out intervals. This phenomenon is more pronounced at 98% and 105% offered loads than at 90% offered load because the loss rate at 98% and 105% offered load is significantly higher than at 90% load (for example, the loss rate for a drop-tail queue of 240 packets increased from 1.8% at 90% load to 6.0% at 98% load and to 8.8% at 105% load). The high packet loss rate at these high offered loads caused a larger portion of flows to experience packet losses and go into time-out. As will be seen in subsequent sections, this phenomenon can also be observed with other AQM algorithms as well.

In summary, experimental results presented in this section show the fundamental trade-off between queuing delay and loss rate in choosing a queue size for drop-tail and that there is no single queue size that can improve performance for all flows. Overall, the drop-tail queue with 240 packets is chosen as a baseline for comparison with AQM algorithms because it appears to achieve a good trade-off for drop-tail queues between improving response times for a large number of small objects and a small number of large objects.

4.2 Results for ARED, PI, and REM with Packet Drops

This section presents experimental results for ARED, PI, and REM. These AQM algorithms were chosen to be evaluated in a previous study because they were the most most prominent AQM algorithms in research literature [LAJS03]. Another reason for choosing these three AQM algorithms was that ARED was developed based on intuitions while PI was designed within a control theoreotic framework and REM was derived from an opti-

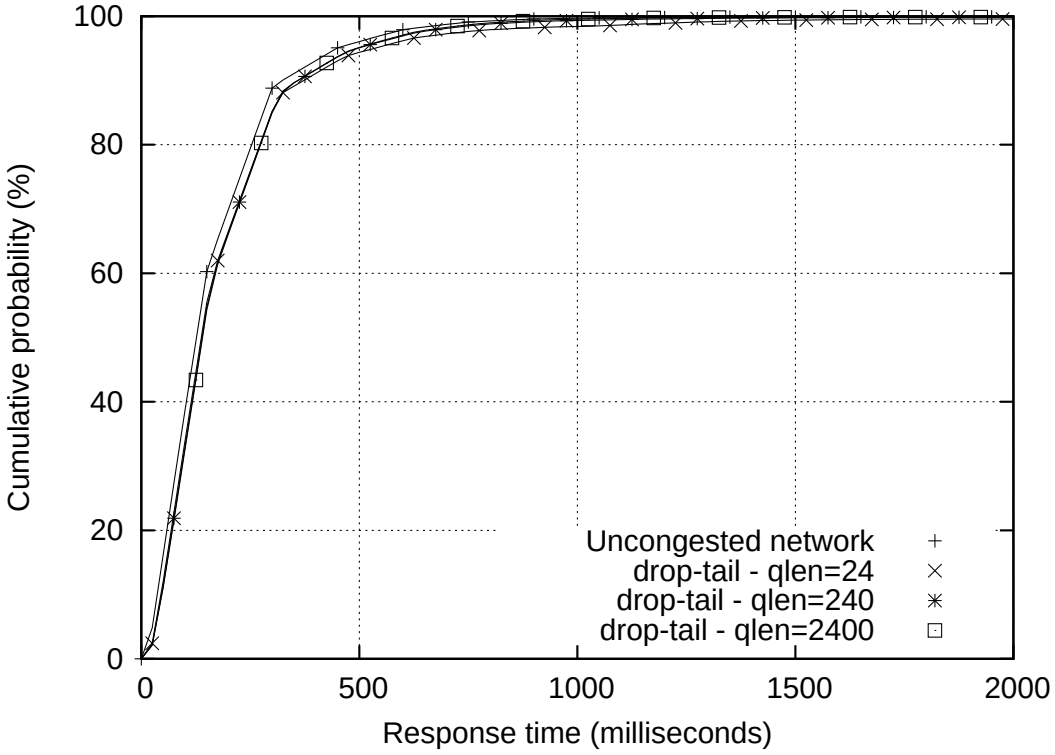

Figure 4.1: Drop-tail performance at 80% load

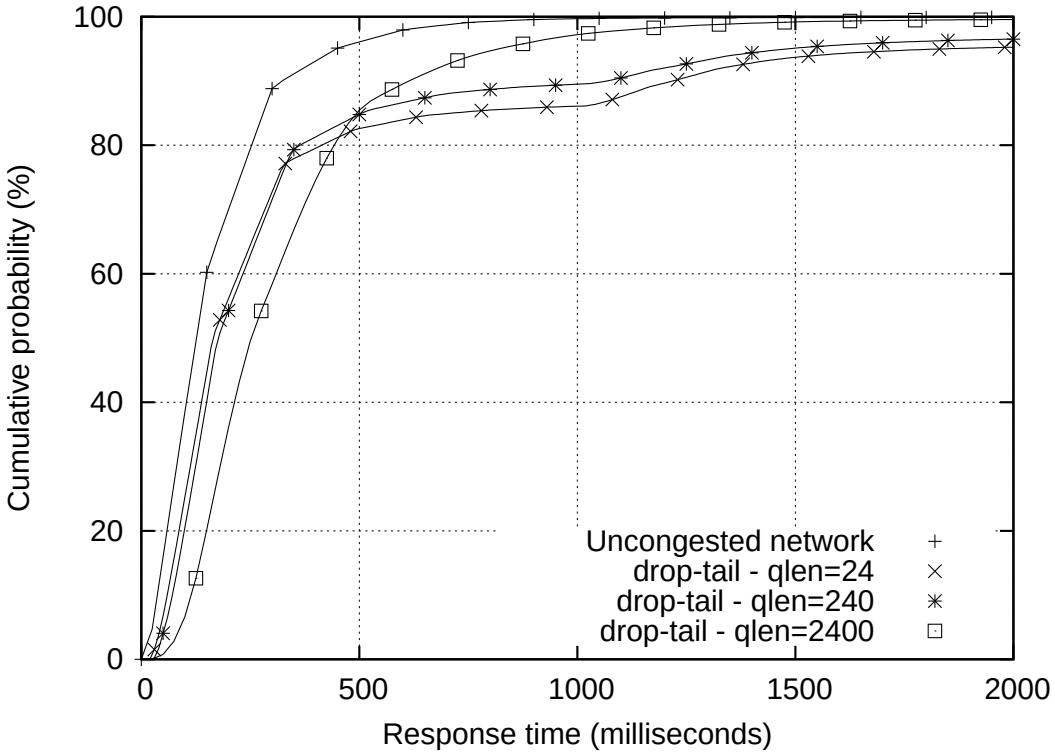

Figure 4.2: Drop-tail performance at 90% load

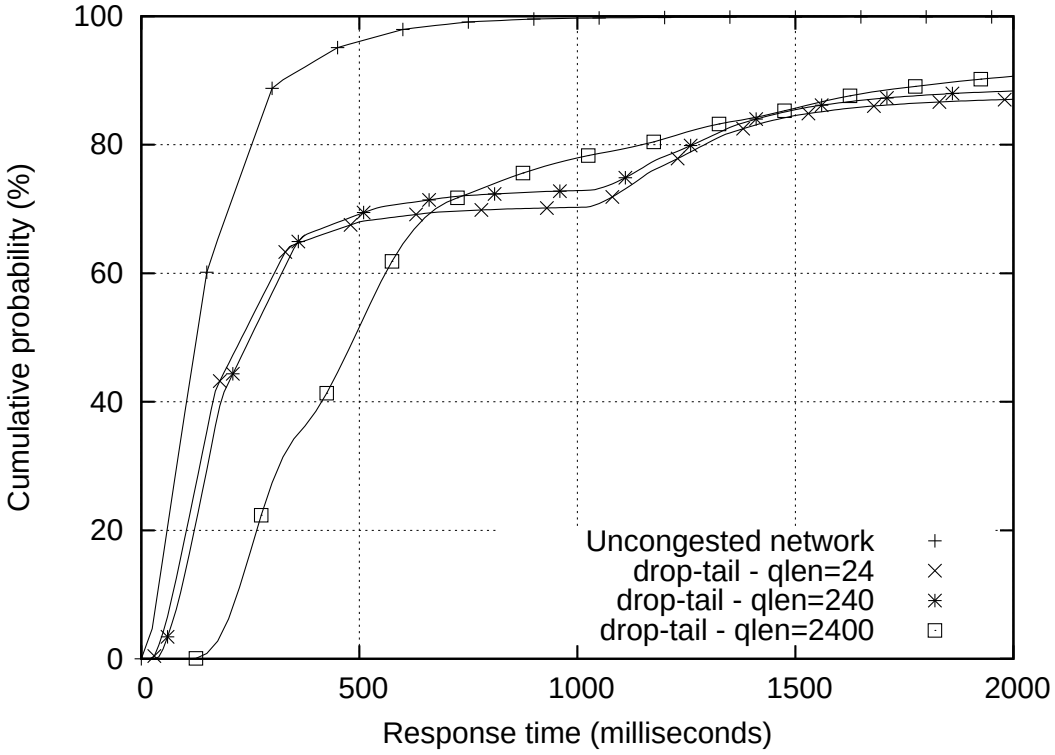

Figure 4.3: Drop-tail performance at 98% load

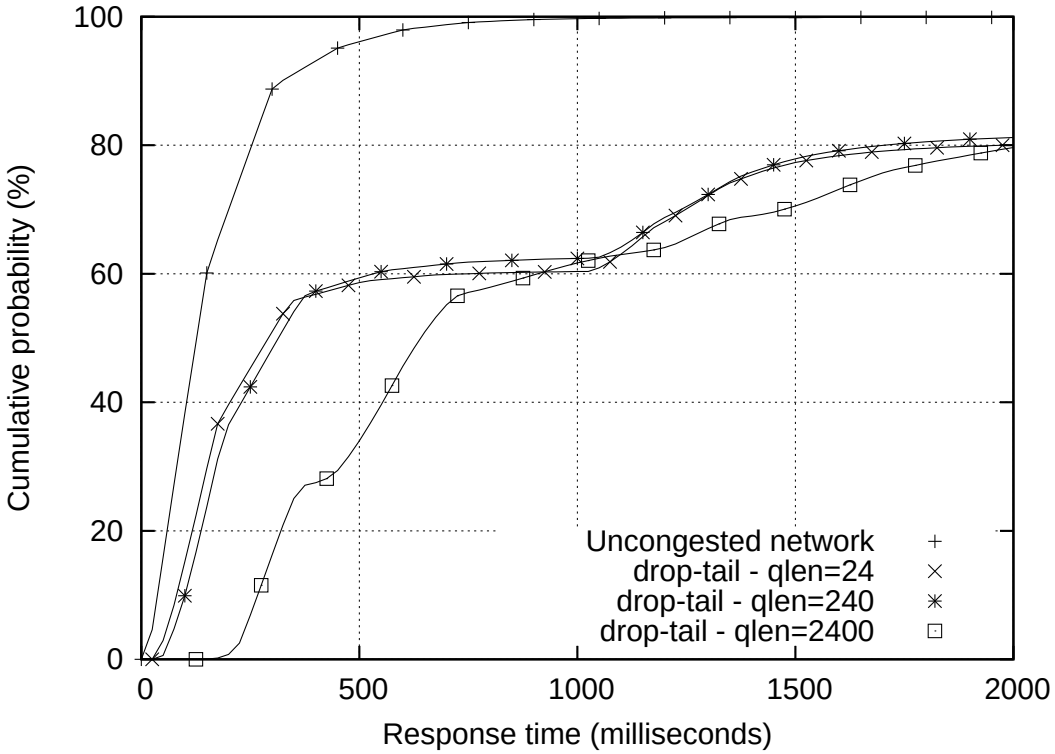

Figure 4.4: Drop-tail performance at 105% load

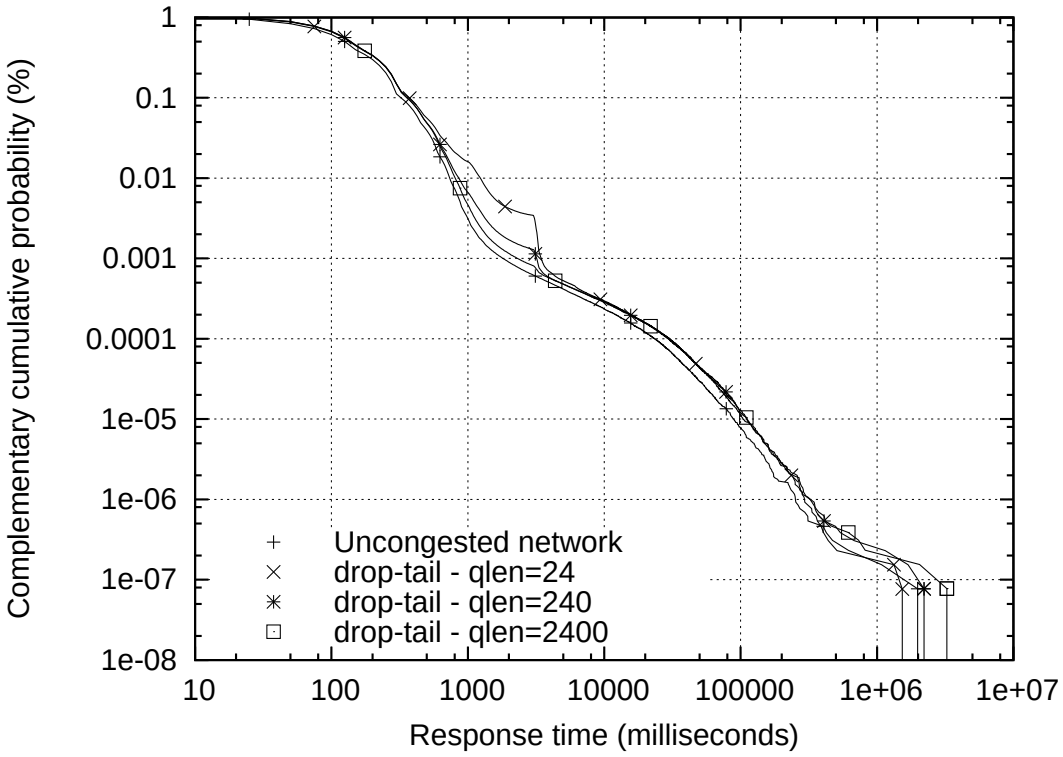

Figure 4.5: Drop-tail performance at 80% load (CCDF)

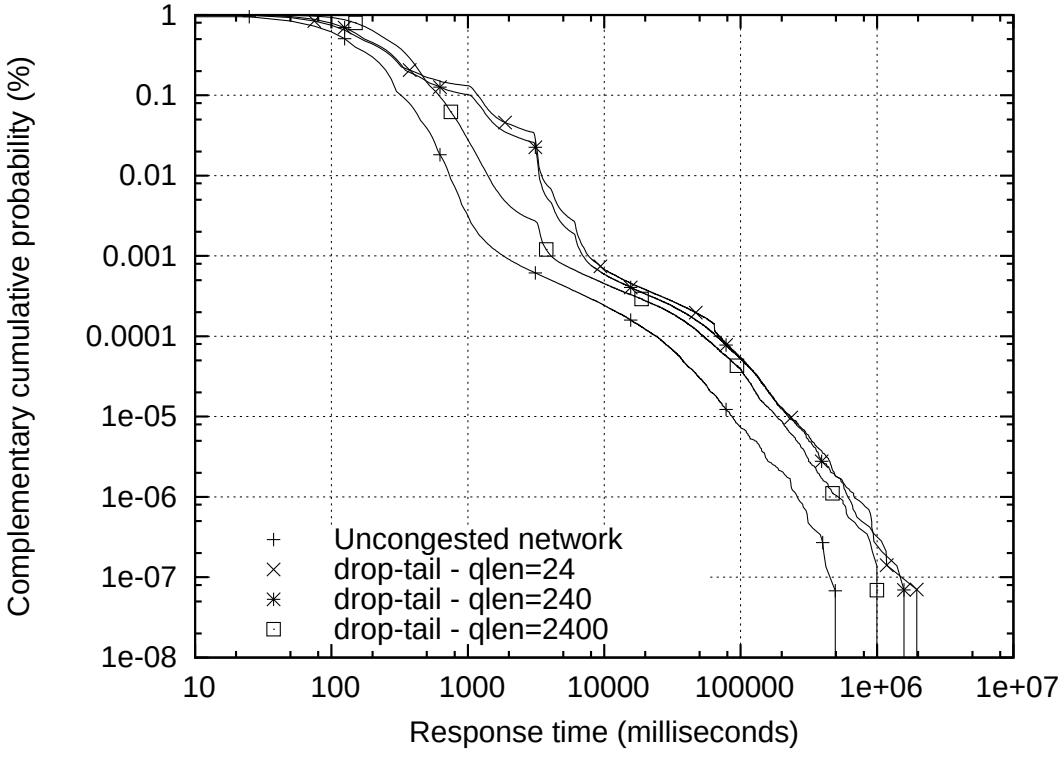

Figure 4.6: Drop-tail performance at 90% load (CCDF)

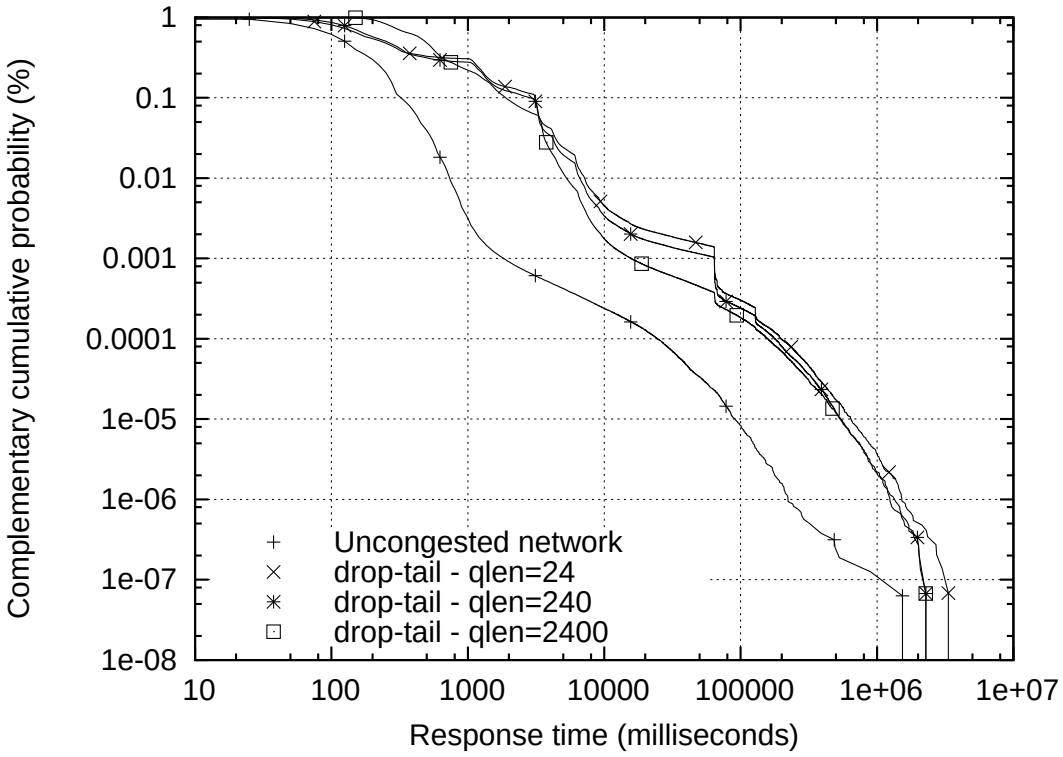

Figure 4.7: Drop-tail performance at 98% load (CCDF)

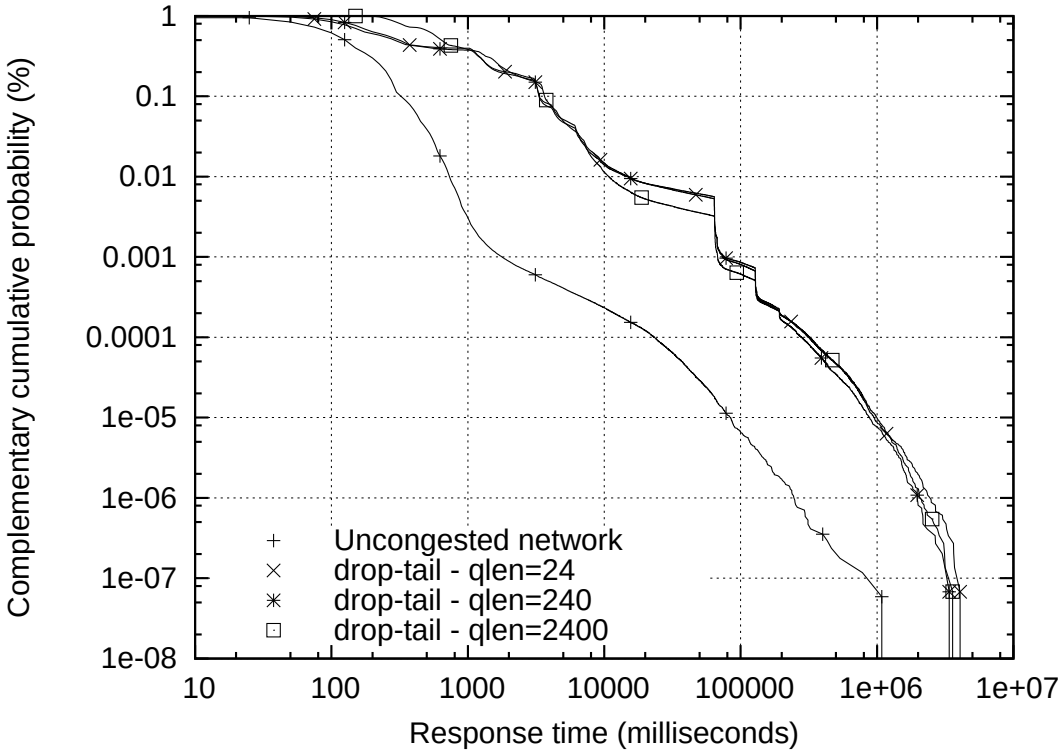

Figure 4.8: Drop-tail performance at 105% load (CCDF)

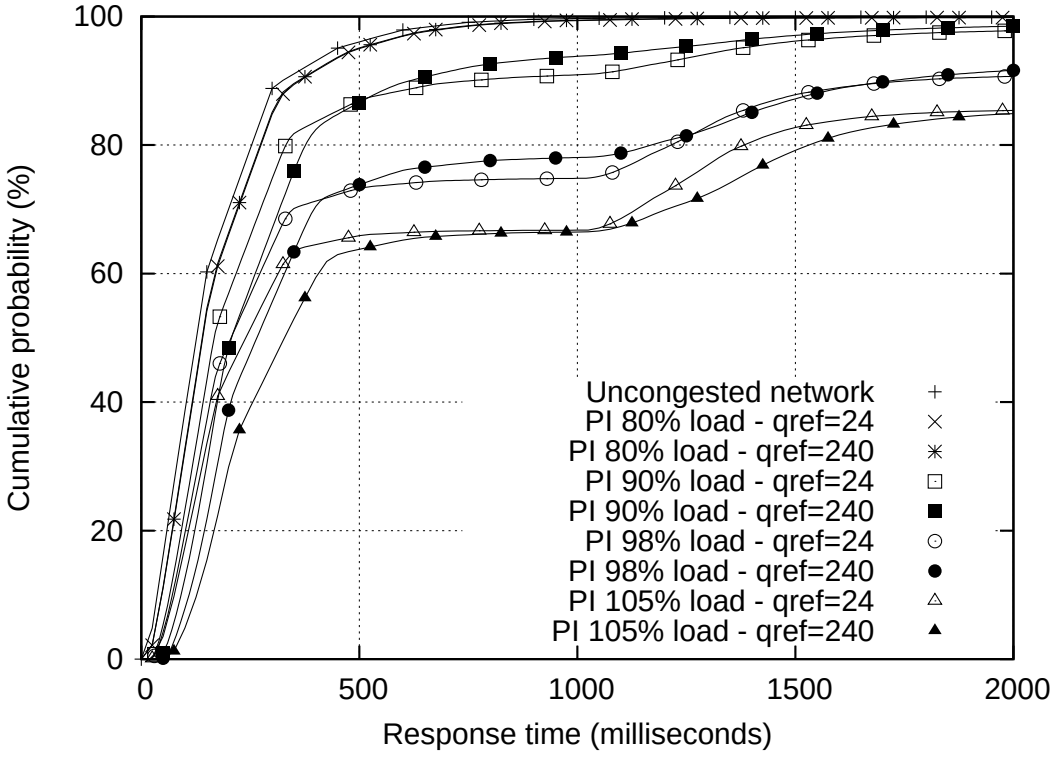

Figure 4.9: PI performance at 80%, 90%, 98%, and 105% load

mization framework. Hence, it was interesting to see how these three AQM algorithms derived from different approahces compare to each other.

Experimental results were obtained for the ARED, PI, and REM algorithms when they were used with packet drops. The results presented for PI and REM in this section were obtained when a queue reference of 24 or 240 packets was used. While the ARED algorithm does not have an explicit target queue reference, this algorithm operates in such a way that the average queue size is approximately midway between th_{min} and th_{max}. Thus, a queue reference of 24 packets is achieved for ARED by setting th_{min} to 12 packets and th_{max} to 36 packets. Similarly, a queue reference of 240 packets is achieved for ARED by setting th_{min} to 120 packets and th_{max} to 360 packets.

Figure 4.9 shows the distributions for response times for PI with a queue reference of 24 and 240 packets at 80%, 90%, 98%, and 105% loads. Futher, Figure 4.10 shows the tails of these distributions for completeness.

At 80% load, PI achieves similar results for both queue reference values. Furthermore, PI's response-time performance with both queue reference values closely approximated the performance of the uncongested network. The uncongested network and PI with both queue reference values gave the same 50th and 75th percentiles of response times (0.137 and 0.312 seconds) at 80% offered load. Further, the 90th percentile of response times for PI was

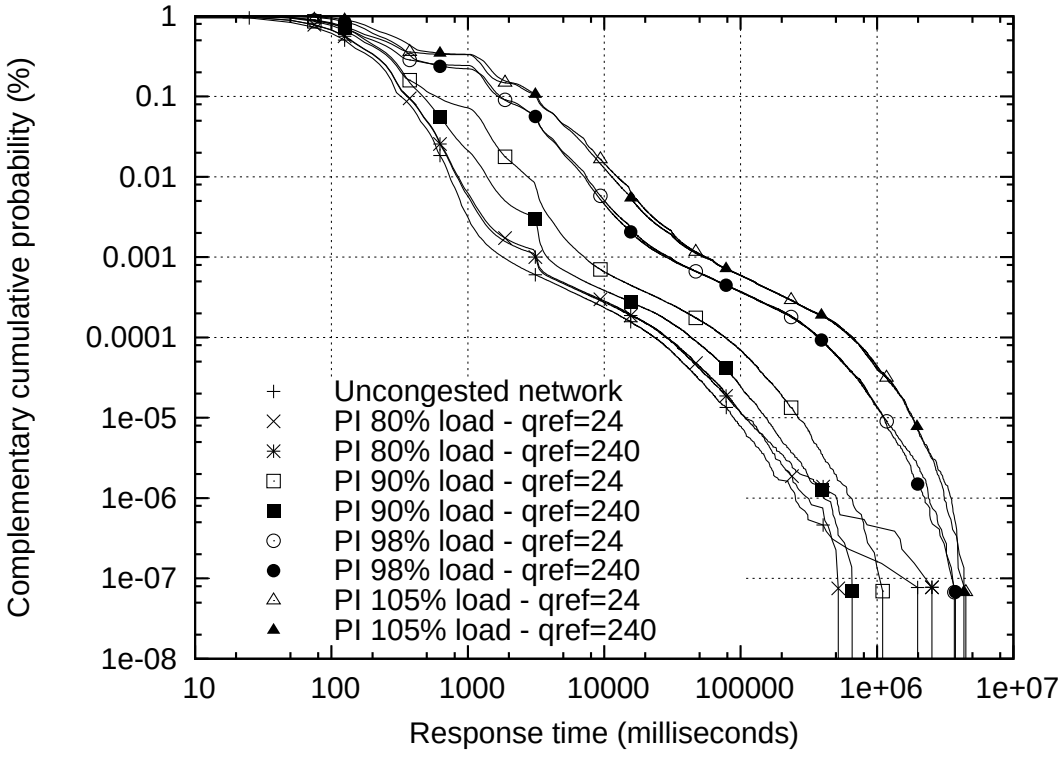

Figure 4.10: PI performance at 80%, 90%, 98%, and 105% load (CCDF)

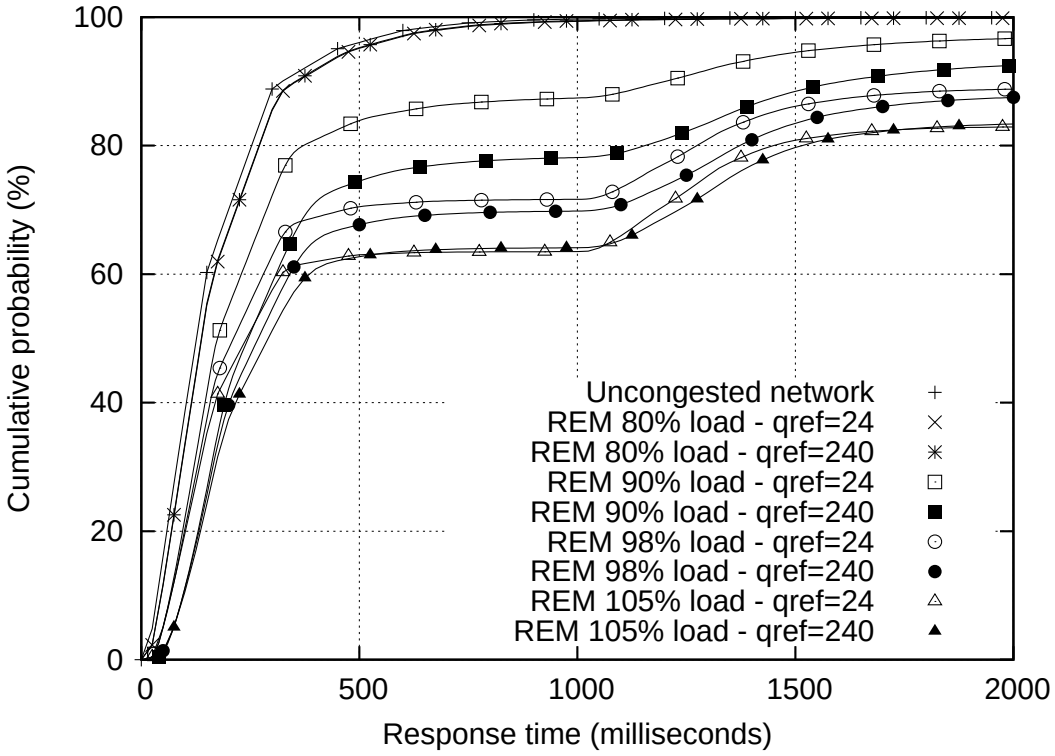

Figure 4.11: REM performance at 80%, 90%, 98%, and 105% load

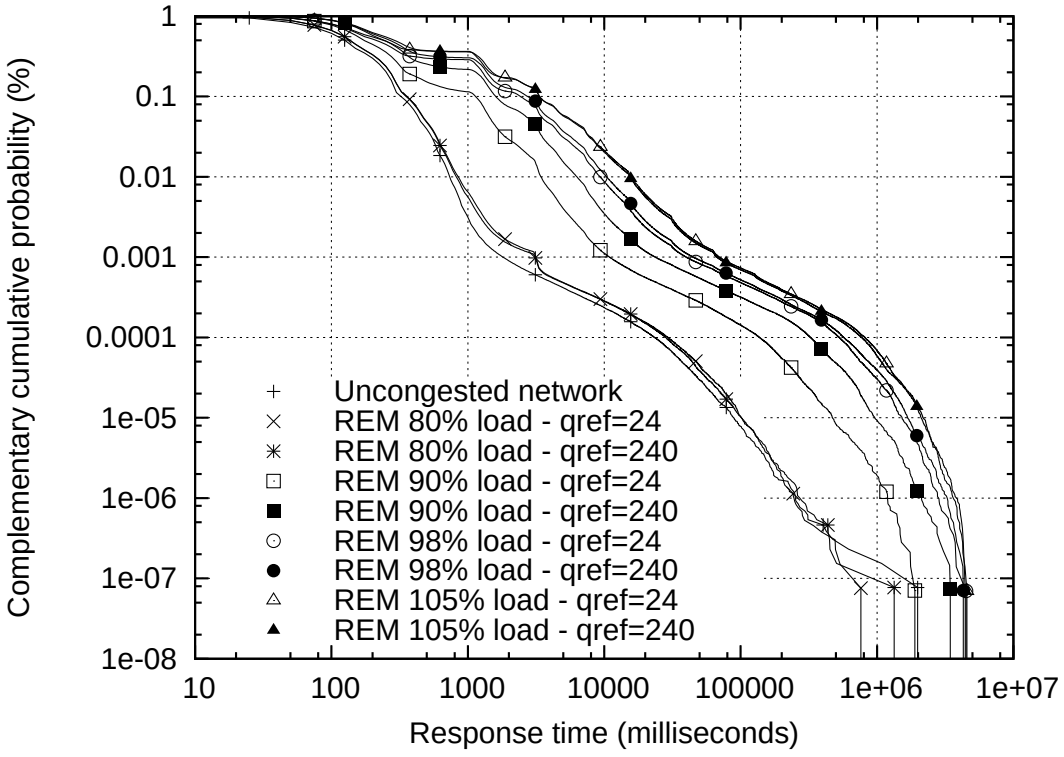

Figure 4.12: REM performance at 80%, 90%, 98%, and 105% load (CCDF)

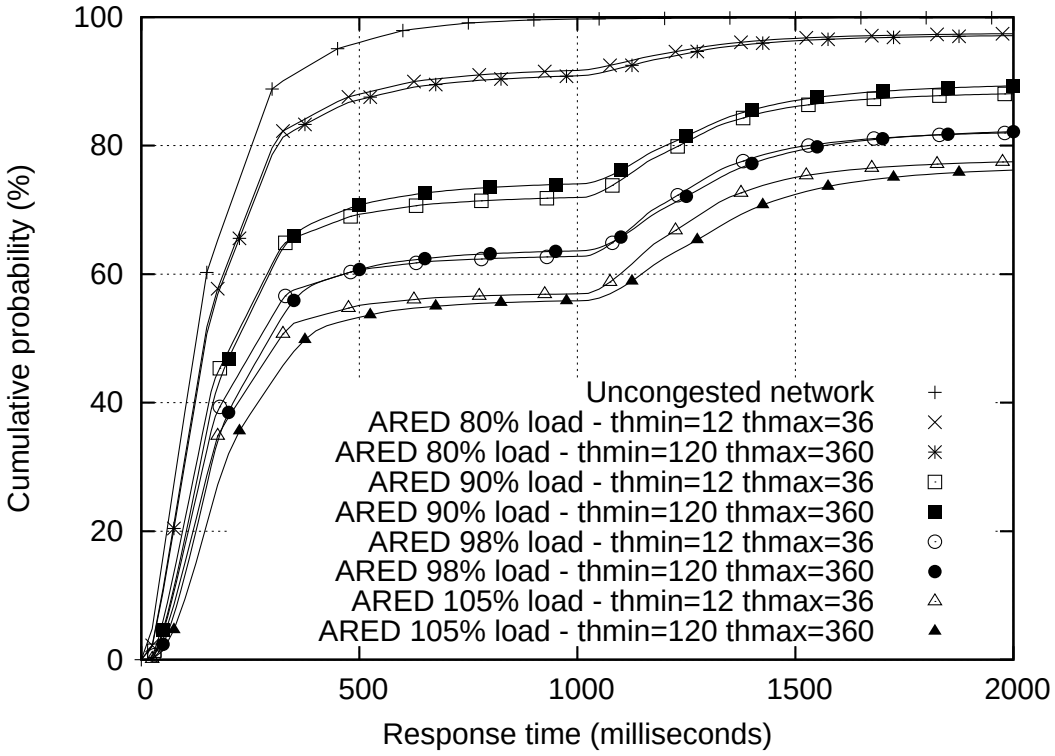

Figure 4.13: ARED performance at 80%, 90%, 98%, and 105% load

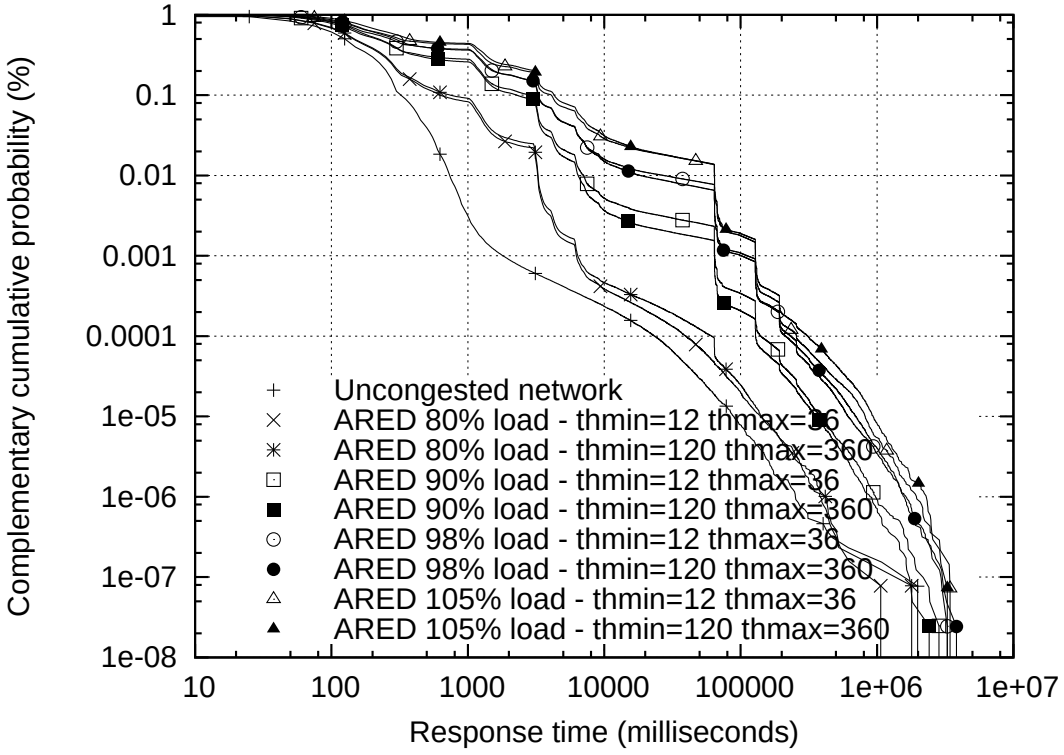

Figure 4.14: ARED performance at 80%, 90%, 98%, and 105% load (CCDF)

only slightly higher than that of the uncongested network (0.362 seconds compared to 0.312 seconds). This good result for PI can be explained by the low packet loss rate of PI at this load (0.0% for both queue reference values). The small difference in response times at 90th percentile can be attributed to queuing delay at the bottleneck link.

As the network load increases to 90%, the performance for PI with both queue reference values degrades considerably. For example, the 90th percentile of response times increased from 0.362 to 0.587 seconds for a queue reference value of 24 packets and from 0.362 seconds to 0.462 seconds for a queue reference value of 240 packets. At the 90% offered load, a queue reference value of 24 packets obtains slightly better results for approximately 80% of flows (those that finish within 500 milliseconds) than a queue reference of 240 packets. For the other 20% of flows (those that need more than 500 milliseconds to complete), a queue reference of 240 packets give slightly better performance. The trade-off in optimizing response times for short and long flows can be explained by the effects of a queue reference value for PI on the resulting packet loss rate and queuing delay. A queue reference value of 240 packets gave a lower packet loss rate for PI than a queue reference value of 24 packets (0.1% compared to 1.3%) but also incurred higher queuing delays for packets.

At 98% and 105% loads, the performance for PI with both queue reference values degraded significantly. For example, the 75th percentile of response times for PI with a queue

reference value of 24 packets increased from 0.287 seconds at 90% load to 0.637 seconds at 98% load and to 1.262 seconds at 105% load. Further, although link throughput only increased slightly (for example, 87.9 Mbps at 90% load compared to 89.3 Mbps at 98% load and 89.9 Mbps at 105% load for PI with a queue reference value of 24 packets), the loss rate increased significantly (for example, 1.3% at 90% compared to 3.9% at 98% load and 6.5% at 105% load for PI with a queue reference value of 24 packets).

Like at 90% load, there is a trade-off for PI in choosing a small queue reference that reduces queuing delay and a large queue reference that can potentially allows the queue to grow and reduces the packet loss rate at 98% load. This trade-off is reflected in the fact that PI obtained better response times for short flows but gave worse response times for long flows with a queue reference value of 24 packets. For example, the 50th percentile of response times was lower with a queue reference value of 24 packets than with a queue reference value of 240 packets for PI at 98% load (0.212 seconds compared to 0.262 seconds). However, the 75th percentile of response times was higher with a queue reference value of 24 packets than with a queue reference value of 240 packets for PI at this load (0.637 seconds compared to 0.562 seconds).

When the offered load increases to 105%, PI obtained better performance with a queue reference of 24 packets than with a queue reference of 240 packets. This result indicates that the potential negative effects of increasing the queuing delay by using a higher queue reference outweighs the benefits of reducing loss rates for PI at this high offered load.

Figure 4.11 gives the distributions of response times for REM with a queue reference of 24 and 240 packets at 80%, 90%, 98%, and 105% loads. Further, Figure 4.12 shows the tails of these distributions for completeness.

Like PI, REM obtained similarly good results for both queue reference values at 80% load and these results closely approximate the result obtained on an uncongested network. This good result for REM can be explained by the very low packet loss rate that REM gave at this load (0.01% for a queue reference value of 24 packets and 0.0% for a queue reference value of 240 packets).

At 90% offered load, REM's performance degraded significantly for both queue reference values. The 75th percentiles of response times increased from 0.237 to 0.312 seconds for a queue reference value of 24 packets and from 0.237 to 0.512 seconds for a queue reference value of 240 packets. Overall, REM achieves better response-time performance with a queue reference of 24 packets than with a queue reference of 240 packets at this load. This result is also in agreement with the lower loss rate that REM obtained with a queue reference value of 24 packets (1.8% for a queue reference value of 24 packets compared to 3.3% for a queue reference value of 240 packets). Further, REM also obtained a higher link utilization with a queue reference value of 24 packets than with a queue reference value of 240 packets (86.4 Mbps compared to 83.3 Mbps). A queue reference of 24 packets is clearly better for

REM at this load.

As the offered load increased to 98% and 105%, a queue reference of 24 packets is still superior to a queue reference of 240 packets for REM in terms of response times, loss rates and link utilization. However, the performance gap between both queue reference values at these loads is not as drastic as at 90% load. For example, the 50th and 75th percentiles of response times were 0.212 and 1.162 seconds for REM with a queue reference value of 24 packets and 0.262 and 1.237 seconds for REM with a queue reference value of 240 packets at 98% load. Like PI and drop-tail, the performance for REM degraded even more significantly at these high loads. For example, the 75th percentile of response times for REM with a queue reference value of 24 packets increased from 0.312 seconds at 90% load to 1.162 seconds at 98% load and the packet loss rate for REM with a queue reference value of 24 packets increased from 1.8% at 90% load to 5.0% at 98% load. Overall, REM obtained better performance with a queue reference value of 24 packets although the performance for REM with both queue reference values was rather poor (in terms of response times and packet loss rates) at 98% and 105% offered loads.

Figure 4.13 presents the results for ARED with a target queue reference of 24 and 240 packets at 80%, 90%, 98%, and 105% loads (as discussed above, the target queue reference for ARED is implicitly specified by choosing the queue thresholds th_{min} and th_{max}). Futher, Figure 4.14 shows the tails of the distributions of response times for completeness.

In contrast to PI and REM, the performance of ARED degraded considerably at 80% load when compared to that of the uncongested network. While ARED with both parameter settings gave the same 50th percentile of response times as the uncongested network (0.137 seconds), ARED obtained higher values for the 75th and 90th percentile of response times than the uncongested network at 80% load (0.262 and 0.637 seconds for ARED with (th_{min} = 12 packets, th_{max} = 36 packets) and 0.287 and 0.862 seconds for ARED with (th_{min} = 120 packets, th_{max} = 360 packets), compared to 0.237 and 0.312 seconds for the uncongested network). From these results, it can be observed that ARED achieved slightly better performance with (th_{min} = 12 packets, th_{max} = 36 packets) than with (th_{min} = 120 packets, th_{max} = 360 packets) at this load. However, the performance gap between the two queue reference values was negligible.

At 90% load, the performance of ARED degraded even more significantly. For example, the 75th percentile of response times increased for ARED with (th_{min} = 12 packets, th_{max} = 36 packets) from 0.262 at 80% load to 1.112 seconds at 90% load and for ARED with (th_{min} = 120 packets, th_{max} = 360 packets) from 0.287 at 80% load to 0.837 seconds at 90% load. Overall, ARED gave slightly better response times with (th_{min} = 120 packets, th_{max} = 360 packets) than with (th_{min} = 12 packets, th_{max} = 36 packets) at this load. Further, ARED also achieved slightly lower packet loss rate with (th_{min} = 120 packets, th_{max} = 360 packets) than with (th_{min} = 12 packets, th_{max} = 36 packets) at 90% offered

load (1.4% compared to 1.5%).

At 98% offered load, ARED gave a loss rate of 4.1% and 4.8% and a link throughput of 87.4 and 87.9 Mbps for the parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets) and ($th_{min} = 120$ packets, $th_{max} = 360$ packets). While a drop-tail queue of 240 packets gave a packet loss rate of 6.0% at 98% load, slightly higher than ARED, it is worth noting that a drop-tail queue of 240 packets also obtained a link throughput of 92 Mbps, higher than ARED. In terms of response times, the 50th and 90th percentiles of response times were 0.137 and 0.312 seconds for the uncongested network, and 0.262 and 2.862 seconds for a drop-tail queue of 240 packets. These values were 0.262 and 3.287 seconds for ARED with the parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets), and 0.287 and 3.287 seconds for ARED with the paremeter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets). Thus, at 98% load, ARED's performance was poorer than the performance of drop-tail and deviates significantly from the performance of the uncongested network. This trend continue to hold for ARED with both parameter settings at 105% offered load.

4.3 Results for BLUE and AVQ with Packet Drops

This section presents the experimental results for BLUE and AVQ when they are used with packet drops. Unlike the AQM algorithms evaluated in section 4.2 (PI, REM, and ARED), BLUE and AVQ do not control a router's queue around a target queue reference but adapt the packet loss rate such that the router's queue does not overflow. For their operations, BLUE and AVQ need a parameter that specifies the router's queue size. When the router's queue reaches the specified queue size, a queue overflow occurs and arriving packets are dropped. (However, note that the BLUE algorithm bases its operations directly on the physical packet queue whereas the AVQ algorithm operates on a virtual queue as explained in Chapter 2.)

Because of the aforementioned reason, experiments for BLUE and AVQ were performed with a router's queue size of 240 and 500 packets. (In case of AVQ, this queue size only applies to the virtual queue. The size of the physical router's queue is set to a sufficiently large number such that tail drops do not occur as long as the algorithm still allows arriving packets to enter the router's queue). The queue sizes for BLUE and AVQ were chosen to be a few times larger than the queue reference values for PI, REM, and ARED. The rationale for this is that while PI, REM, and ARED can allow the router's queue to temporarily grow above the specified target queue reference values, BLUE and AVQ do not let the router's queue increase above the specified queue sizes.

Because BLUE and AVQ also drop arriving packets when the router's queue overflows like FIFO, another motivation for choosing a queue size of 240 packets for these algorithms was to enable a direct comparison between them and drop-tail (as discussed in section 4.1,

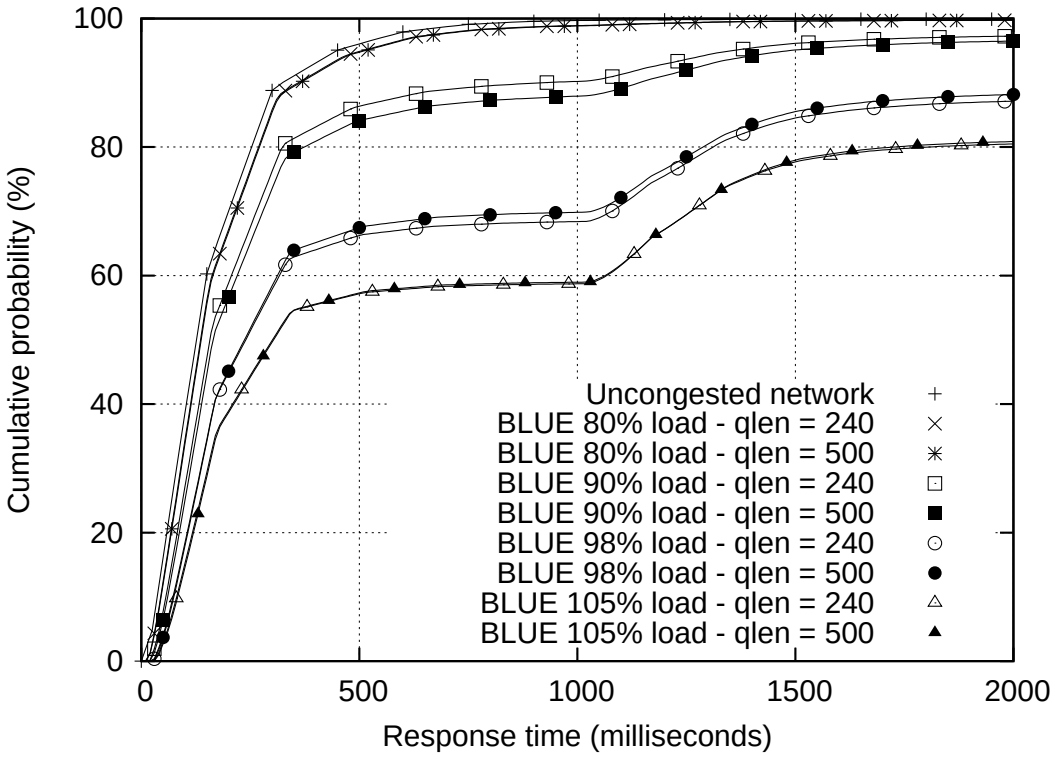

Figure 4.15: BLUE performance at 80%, 90%, 98%, and 105% load

240 was the empirically-determined optimal queue size for drop-tail).

4.3.1 Results for BLUE

Figure 4.15 shows the experimental results for BLUE with a queue length of 240 and 500 packets at 80%, 90%, 98%, and 105% loads. Futher, Figure 4.16 presents the tails of the distributions of response times for BLUE at these loads for completeness.

At 80% offered load, BLUE achieved equally good performance with both queue sizes and closely approximated the performance of the uncongested network.

As the offered load increased to 90%, BLUE experienced a noticeable performance degradation for both queue sizes. BLUE achieved better response times for all flows with a queue length of 240 packets than with a queue length of 500 packets at this load.

At 98% offered load, the performance for BLUE with both queue sizes degraded even further. BLUE obtained slightly better performance with a queue length of 500 packets than with a queue length of 240 packets at this load.

At 105% offered load, BLUE experienced even more performance degradation for both queue sizes. BLUE obtained equally poor performance with both queue lengths at this extreme load.

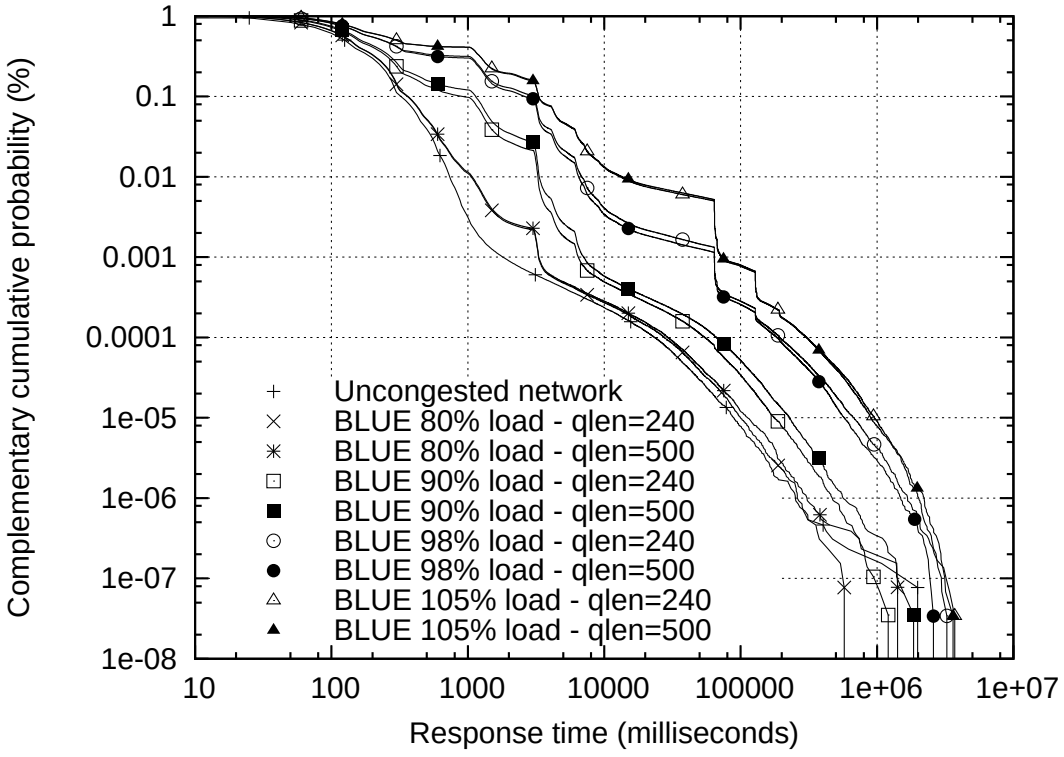

Figure 4.16: BLUE performance at 80%, 90%, 98%, and 105% load (CCDF)

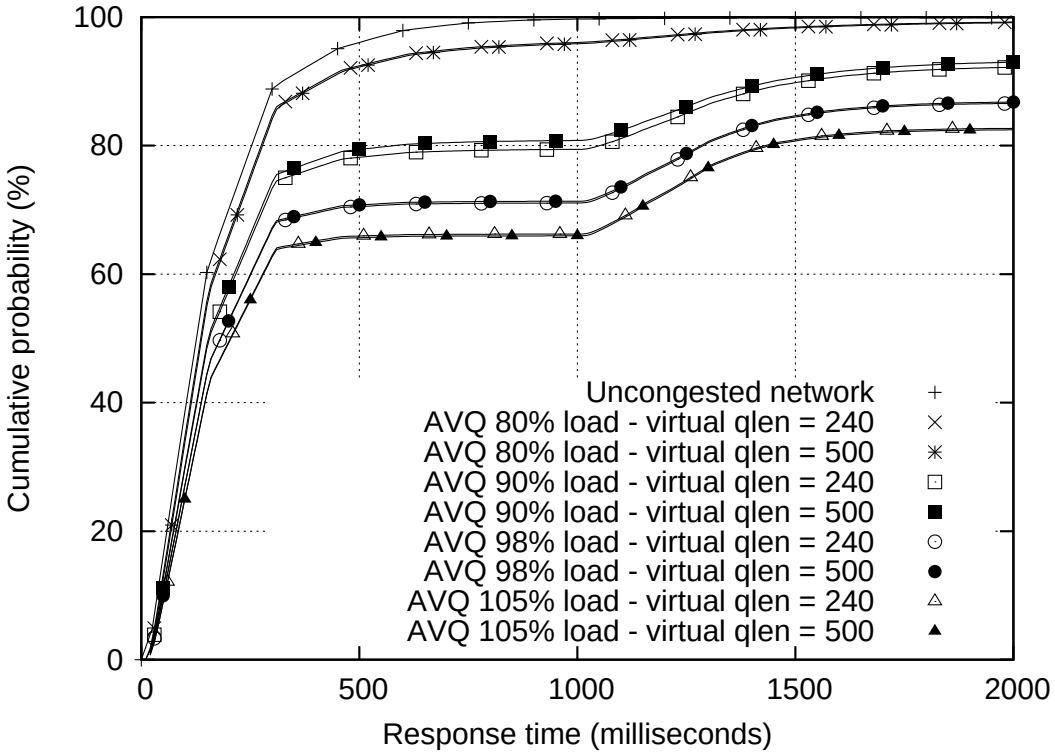

Figure 4.17: AVQ performance at 80%, 90%, 98%, and 105% load

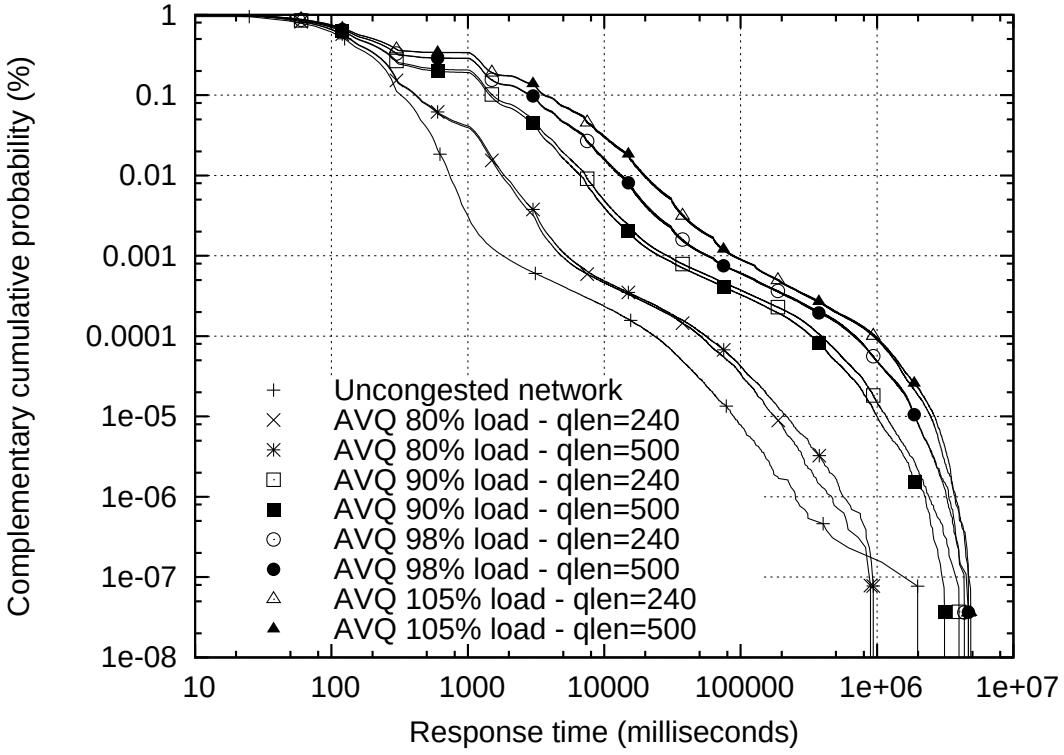

Figure 4.18: AVQ performance at 80%, 90%, 98%, and 105% load (CCDF)

4.3.2 Results for AVQ

Figure 4.17 presents the experimental results for AVQ with a virtual queue length of 240 and 500 packets at 80%, 90%, 98%, and 105% loads. For completeness, the tails of the distributions of response times for AVQ at these loads are also shown in Figure 4.18.

At 80% offered load, the performance of AVQ degraded noticeably for both virtual queue lengths of 240 and 500 packets when compared to that of the uncongested network. AVQ obtained approximately the same performance with both virtual queue lengths at this offered load.

As the offered load increased to 90%, the performance of AVQ degraded further for both virtual queue lengths of 240 and 500 packets. AVQ obtained slightly better response times for all flows with a virtual queue length of 500 packets than with a virtual queue length of 240 packets at this load.

At 98% and 105% loads, the performance of AVQ degraded even more for both virtual queue lengths of 240 and 500 packets. AVQ achieved similarly poor performance with both virtual queue lengths at these loads.

4.4 The Effects of Balancing Queuing Delay and Loss Rates

AQM algorithms evaluated in previous sections explicitly try to control router queues by probabilistically dropping packets. Although the ultimate goal of AQM is to ensure that queues never overflow (i.e., ensure that true congestion does not occur), existing AQM algorithms typically achieve this goal by dropping packets aggressively when a router's queue grows larger than a certain threshold. However, I believe that while controlling router queues is important, this control needs to be tempered by a consideration of the overall loss rate at the router. Solely attempting to control queue length can induce loss rates that have as negative an effect on application and network performance as the large queues that existing AQM algorithms were trying to avoid. Thus, controlling queue length without regard to loss rate can be counterproductive.

This section demonstrates the positive effects of balancing between queuing delay and loss rates by using a case study of a new AQM algorithm that attempts to simultaneously optimize queue length and loss rate. The new algorithm, called loss and queuing delay control (LQD), explicitly treats loss rate as a control parameter (in addition to a target queue length parameter) and dynamically balances loss rate and queuing delay at a router to improve network and application performance [LJS06].

Controlling the length of a router's queue is an important and difficult task. A large queue subjects arriving packets to a long queuing delay and can also cause instability in the TCP control feedback loop [LPW+02]. A short queue can be achieved by dropping packets aggressively, however, a short target queue length runs the risk that the queue can drain quickly and become empty before new packets arrive. In this case, the link is underutilized and the router has unnecessarily dropped packets that could have been enqueued and forwarded without ill effect. As argued above, while controlling router's queues is an important goal, it should not be achieved by simply dropping arriving packets. This issue is particularly important because of the bursty characteristics of Internet traffic that can cause temporary congestion at routers [FGW98, PF95]. (AQM algorithms such as RED, ARED, and their derivatives attempt to deal with bursty arrivals by using a low-pass filter to smooth the measure of average queue length, however, as shown in 4.2 and also in following Chapters, this control is ineffective.)

In contrast to existing AQM algorithms, LQD provides a framework for balancing loss rate and queuing delay. LQD is flexible enough to absorb transient bursts where the input rate temporarily exceeds the link capacity. On the other hand, LQD can control and stabilize router's queue when persistent congestion occurs. This design distinguishes LQD from existing AQM algorithms that simply try to control router's queue at any cost (independent of its effect on the environment).

Like most AQM algorithms, on each packet arrival LQD computes a drop probability $p(t)$ which is used to decide whether the arriving packet is to be dropped or forwarded. Let

T be the sampling interval and $l(t)$ be the estimated packet loss rate (i.e., the ratio of the number of dropped packets to the number of arriving packets). The drop probability at time kT is computed as

$$p(kT) = p((k-1)T) + a(q(kT) - q_{ref}) - b(l(kT) - p_{ref}) \tag{4.1}$$

where a and b are coefficients of the LQD controller, and $p_{ref} = 0$, and $q_{ref} > 0$ are the target loss rate and target queue length respectively. Observe that the drop probability is increased when the queue length is larger than the queue target and is decreased otherwise. However, when the loss rate grows larger than its threshold, the drop probability is adjusted downward and the queue is allowed to grow temporarily.

The coefficients a and b allow a router to balance between queuing delay and packet loss rate. The coefficient a specifies how large the queue can grow and the coefficient b allows the router to adjust the loss rate and absorb transient congestion. In general, a should be significantly smaller than b since the range of values for queue length (tens to hundreds) is significantly larger than the range of values for packet loss rate (hundredths to tenths). Experimental data suggests that the difference between the actual queue length and the queue reference is on the order of tens in dynamic environments and the packet loss rate is on the order of hundredths. Based on these results of empirical analysis, a and b were set to 0.0001 and 0.1 for LQD to balance the relative contributions to the drop probability of the queue length mismatch and the loss rate miss match in all experiments.

Figure 4.19 shows the experimental results for LQD with a queue reference of 24 and 240 packets at 80%, 90%, 98%, and 105% loads.

At 80% offered load, LQD delivered equally good performance with both queue reference values of 24 and 240 packets and the performance for LQD came very close to the performance of the congested network.

At 90% offered load, the performance for LQD degraded slightly with both queue reference values of 24 and 240 packets. LQD achieved slightly better performance with a queue reference of 240 packets at this load. In contrast to AQM algorithms evaluated in section 4.2, LQD still gave quite good performance at this high load and demonstrated the positive effects of balancing between queuing delay and loss rates.

As the offered load increased to 98%, the performance for LQD degraded considerably with both queue reference values of 24 and 240 packets. LQD achieved very similar performance with both queue reference values at this load although a queue reference of 24 packets delivered slightly better performance.

At 105% load, the performance for LQD with both queue reference values degraded even further. LQD delivered better response times for approximately 70% of all flows with a queue reference of 24 packets than with a queue reference of 240 packets. For the rest 30% of flows (those that needed more than 500 milliseconds to complete), LQD obtained better

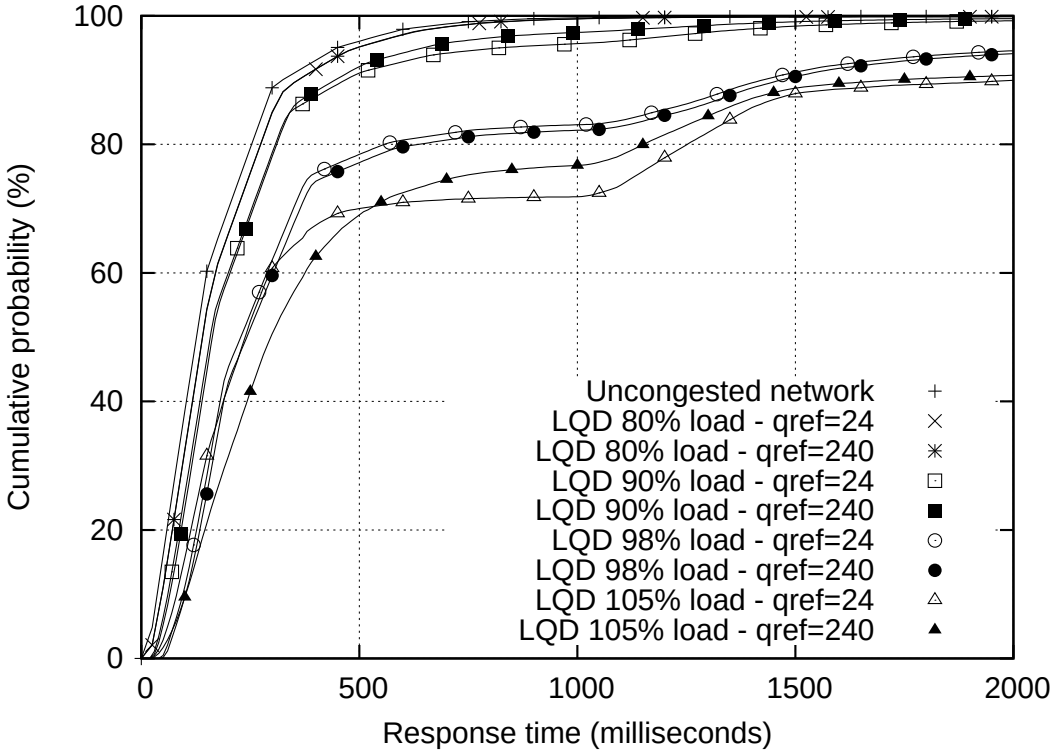

Figure 4.19: LQD performance at 80%, 90%, 98%, and 105% load

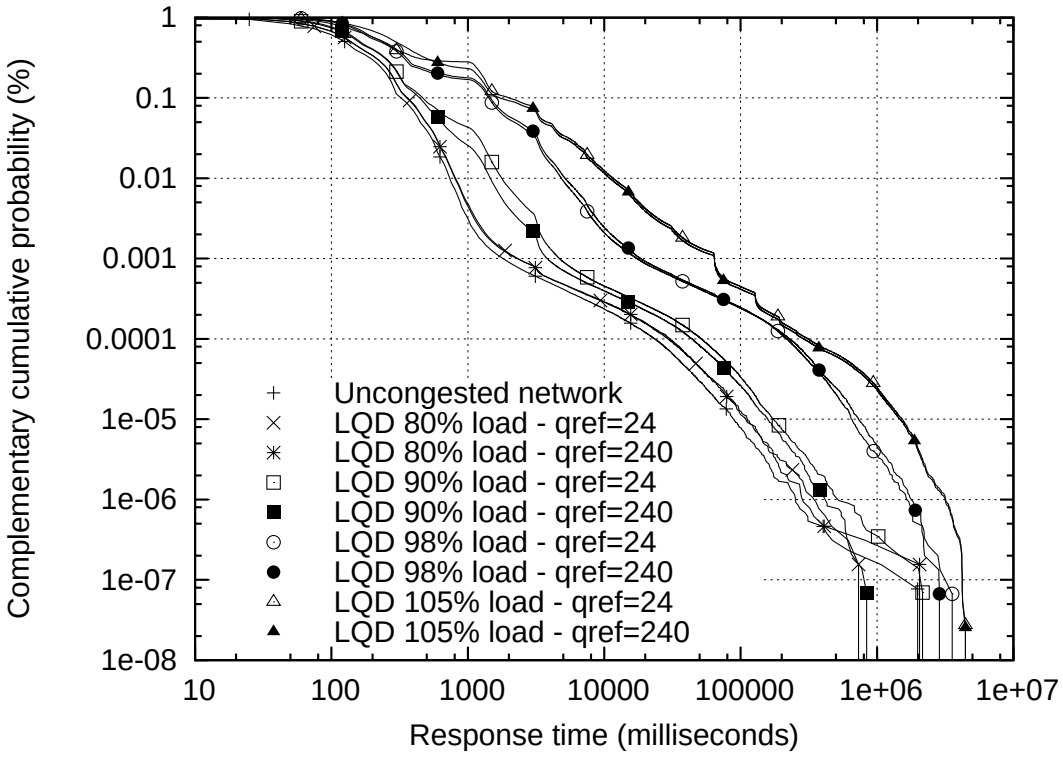

Figure 4.20: LQD performance at 80%, 90%, 98%, and 105% load (CCDF)

performance with a queue reference of 240 packets. This result shows again the trade-off between optimizing response times for short and long flows that was discussed in section 4.2.

4.5 The Effects of Explicit Congestion Notification

Results presented in previous sections demonstrated the effects of AQM algorithms on network and application performance when dropping packets is used as an implicit mechanism for congestion notification. When AQM algorithms are used in combination with the Explicit Congestion Notification protocol, they can set a bit in the header of IP packets to explicitly inform end systems when congestion is about to occur. To assess the effects of explicit congestion notification, experiments were performed with ECN enabled at the routers and end systems.

4.5.1 Results for PI/ECN

Figures 4.21, 4.22, 4.23, and 4.24 show the performance of PI with and without ECN and compare it with the performance of drop-tail at 80%, 90%, 98%, and 105% loads. As in section 4.2, experiments for PI were performed with a target queue reference of 24 and 240 packets.

At 80% offered load, the addition of ECN did not improve the performance for PI. This is because PI already obtained very good performance without ECN at this load and there was virtually no room for improvement.

At 90% offered load, the performance of PI when combined with ECN was improved significantly and came close to that of the uncongested network. Without ECN, PI only provided a small performance improvement over drop-tail. However, when PI was combined with ECN, it obtained considerable better performance than drop-tail.

The performance improvement for PI with ECN was even more impressive at 98% and 105% offered loads. PI when combined with ECN outperformed drop-tail significantly and clearly demonstrated the advantage of AQM.

It is also interesting to note that when PI was used without ECN at 90% and 98% offered loads, there was a crossover between the two queue reference values that occurred at approximately 500 milliseconds (flows completing in less than 500 milliseconds experienced better response times with a queue reference of 24 packets, but flows needing more than 500 milliseconds to complete fared better with a queue reference of 240 packets). As discussed in sections 4.2 and 4.4, this crossover presents the trade-off between queuing delay and loss rates. However, when PI was used with ECN, this crossover no longer existed because ECN helped PI avoided dropping packets and significantly reduce the packet loss rates. When used with ECN, PI performed better with a queue reference of 24 packets than with a queue reference of 240 packets.

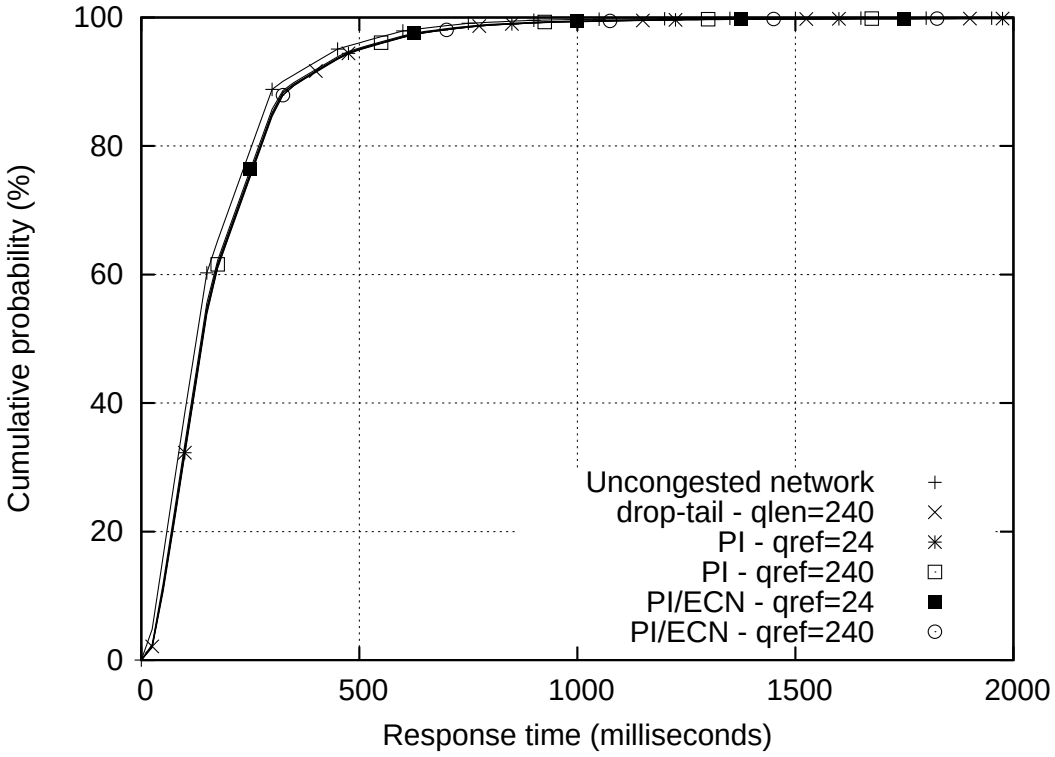

Figure 4.21: PI/ECN performance at 80% load

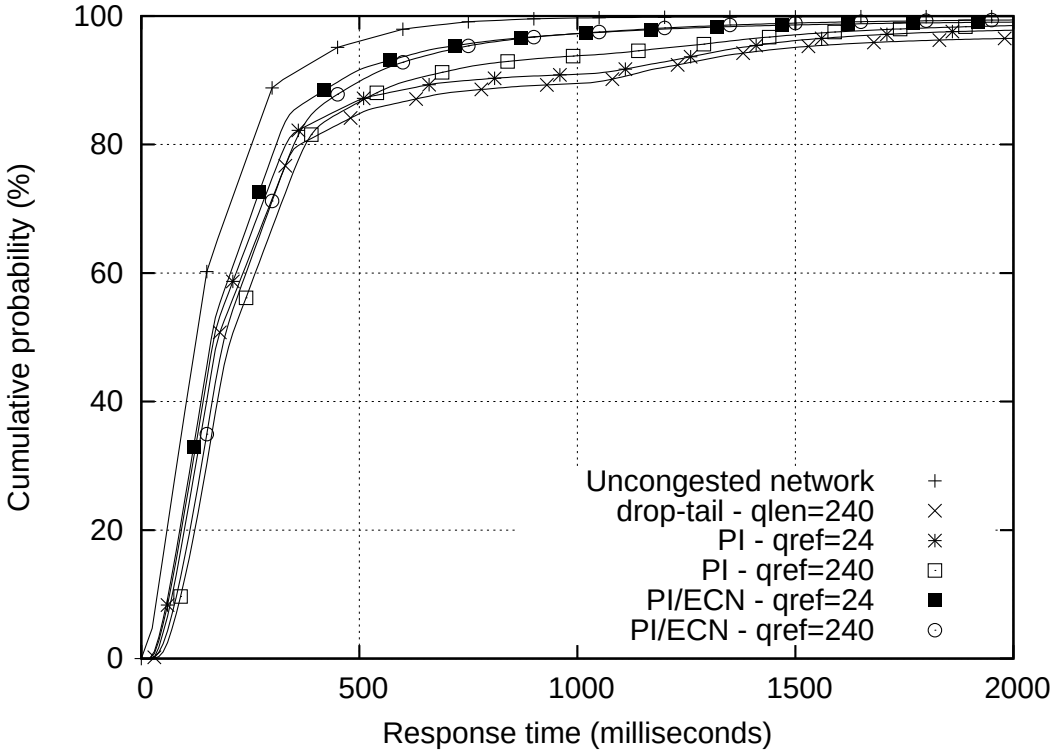

Figure 4.22: PI/ECN performance at 90% load

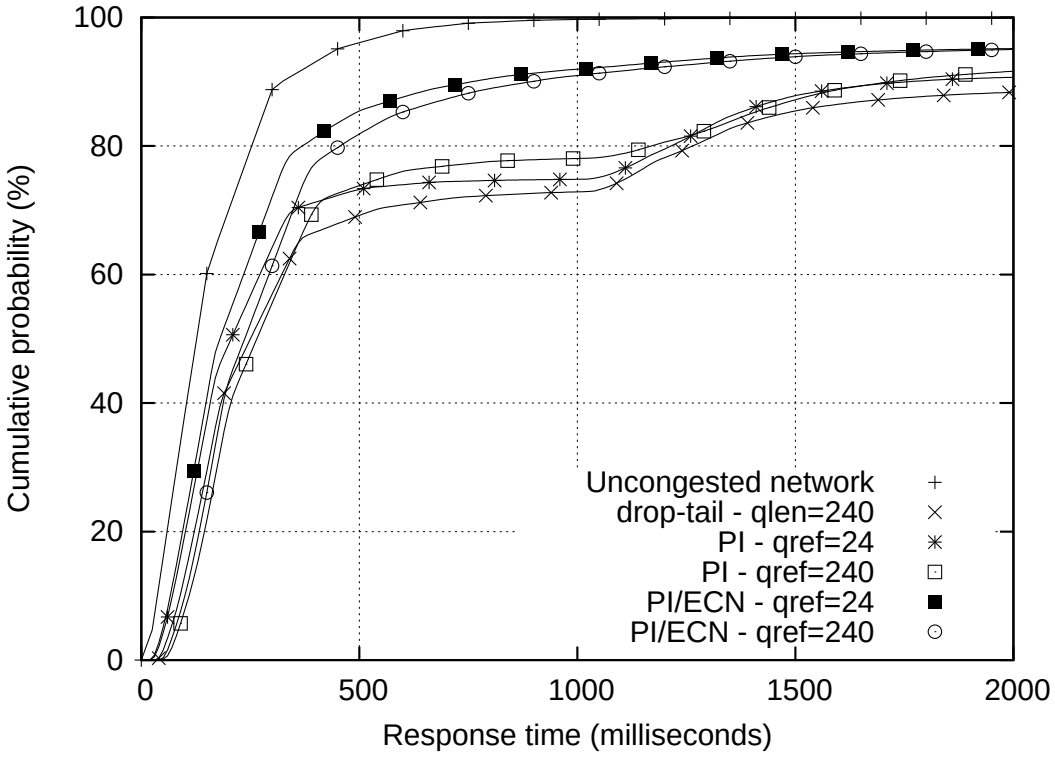

Figure 4.23: PI/ECN performance at 98% load

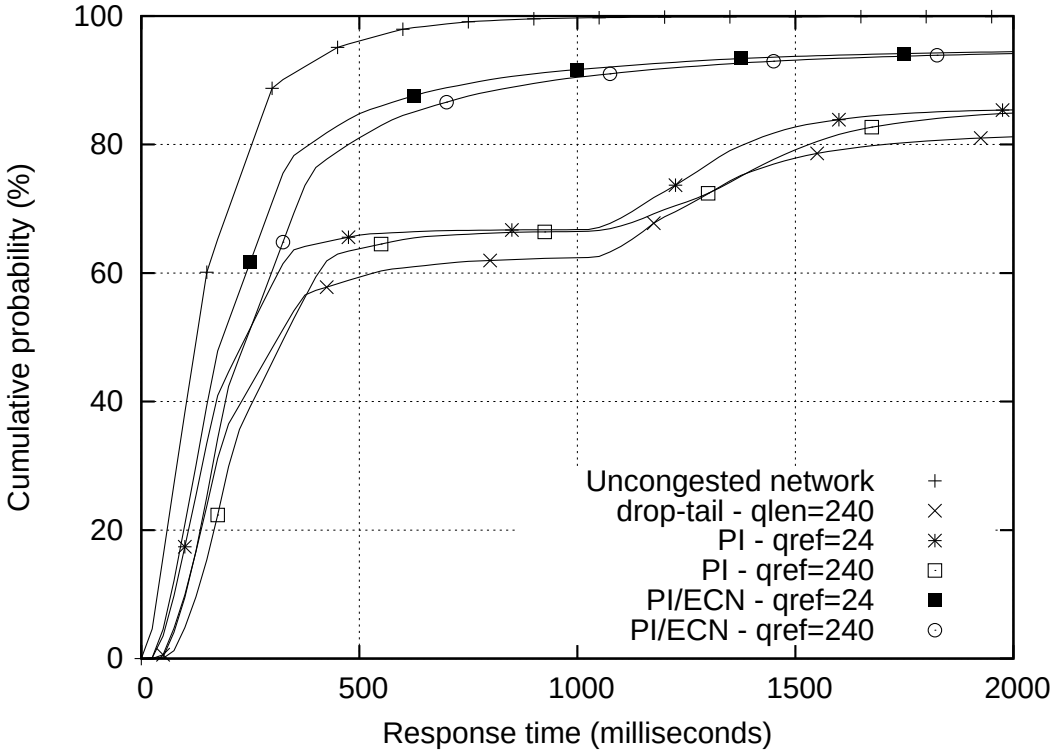

Figure 4.24: PI/ECN performance at 105% load

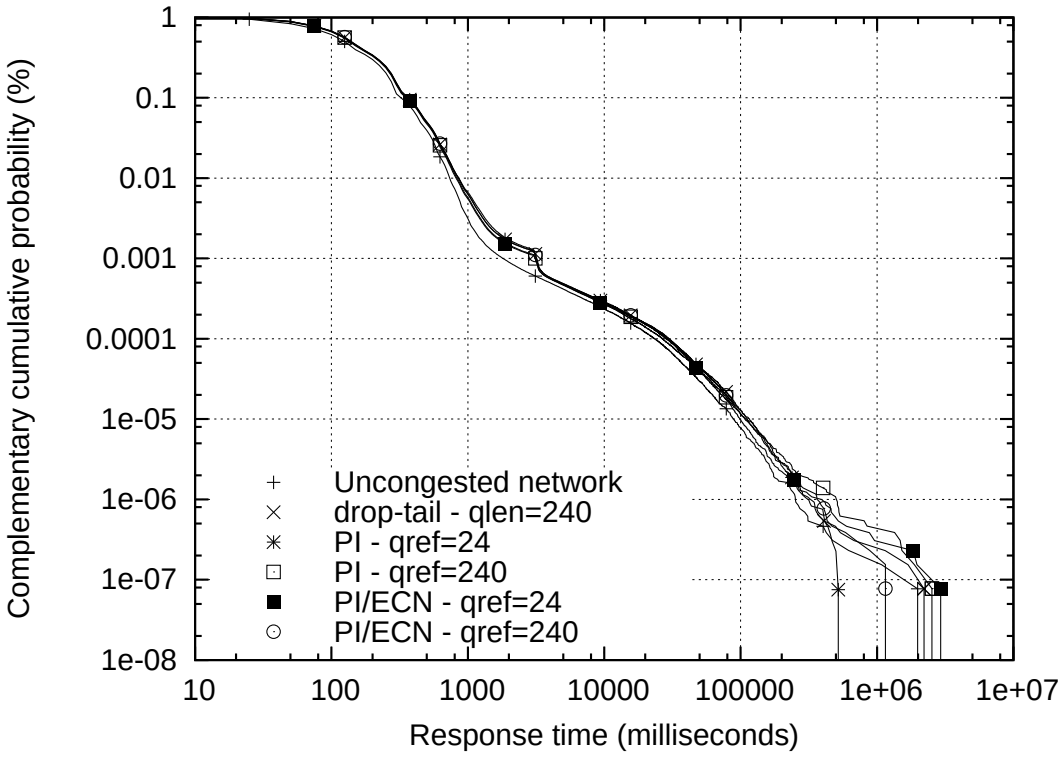

Figure 4.25: PI/ECN performance at 80% load (CCDF)

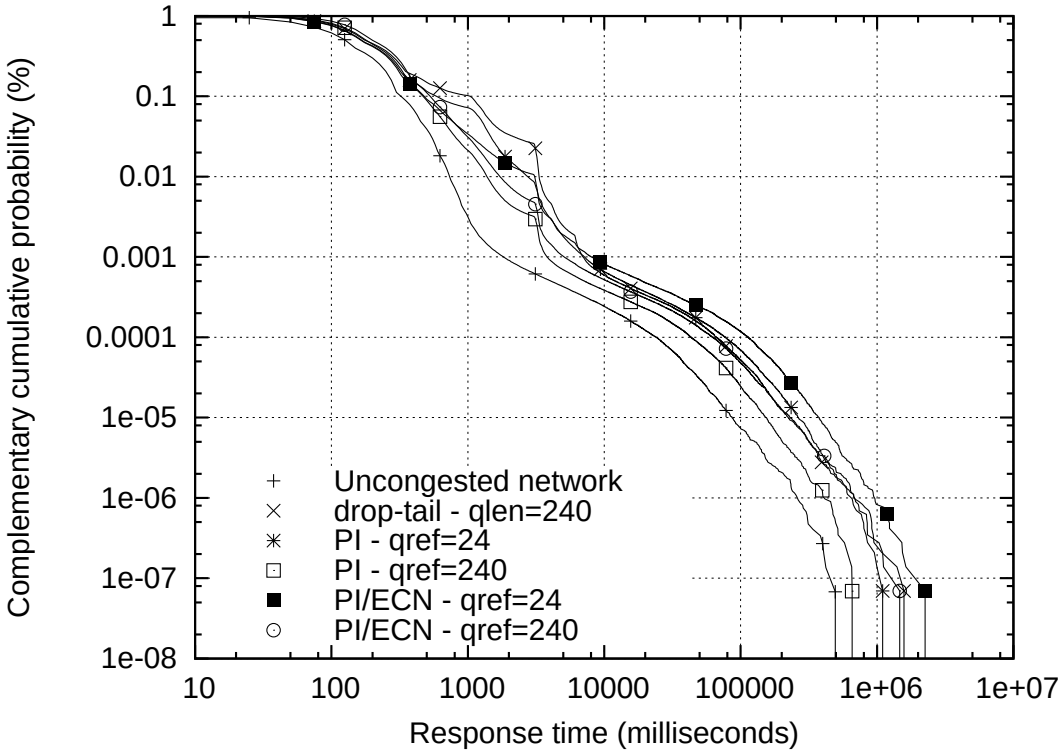

Figure 4.26: PI/ECN performance at 90% load (CCDF)

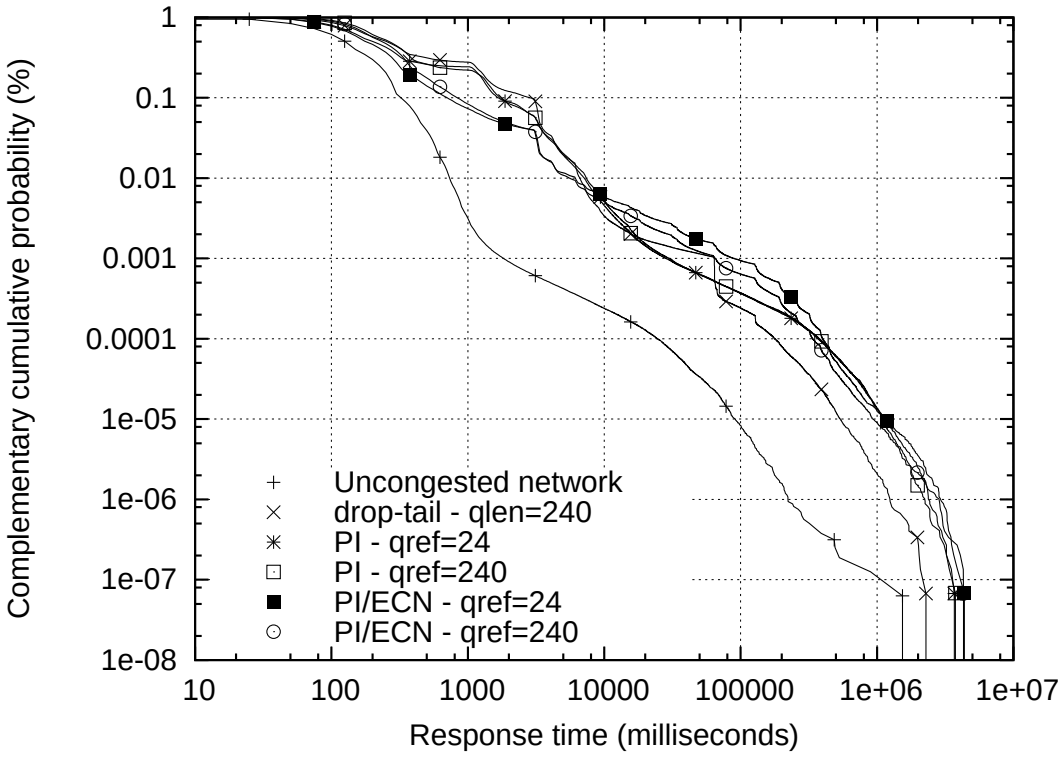

Figure 4.27: PI/ECN performance at 98% load (CCDF)

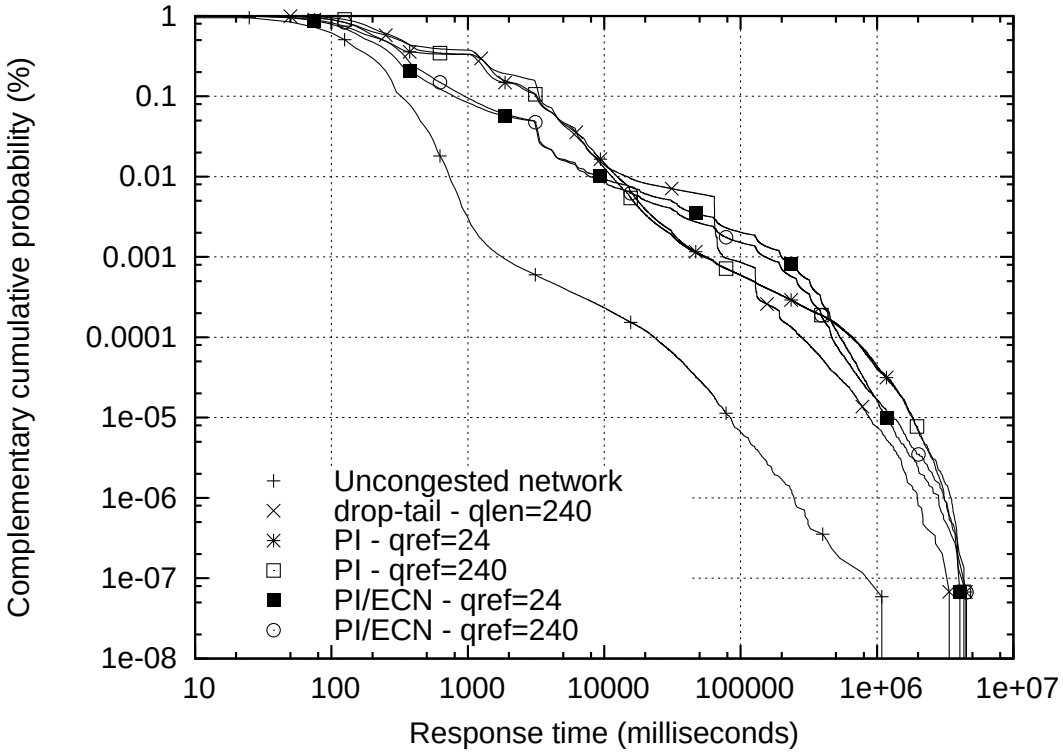

Figure 4.28: PI/ECN performance at 105% load (CCDF)

4.5.2 Results for REM/ECN

Figures 4.29, 4.30, 4.31, and 4.32 show the performance of REM with and without ECN and compare it with the performance of drop-tail at 80%, 90%, 98%, and 105% loads. Experimental results for REM with and without ECN were obtained with a queue reference of 24 and 240 packets.

At 80% offered load, the addition of ECN did not help improve performance for REM because REM already obtained very good performance without ECN at this load. Surprisingly, ECN slightly degraded the performance for REM when REM was used with a queue reference of 24 packets.

As in the case with PI, the performance of REM was improved significantly when combined with ECN at 90% offered load. When REM was used without ECN at this load, REM obtained slightly worse performance than drop-tail with a queue reference of 24 packets. Further, REM delivered significantly worse performance than drop-tail with a queue reference of 240 packets. However, when REM was used with ECN, REM obtained significant performance improvement and outperformed drop-tail with both queue reference values. At 90% load, the performance of REM with ECN was almost comparable with the performance of the uncongested network.

As the offered load increased to 98% and 105%, REM without ECN and with a queue reference of 24 packets obtained slightly better performance than drop-tail. However, this small performance improvement for REM over drop-tail could not offset the significant performance degradation that REM and drop-tail experienced at these extreme loads. Further, REM without ECN and with a queue reference of 240 packets delivered slightly worse performance than drop-tail.

When REM was used with ECN at 98% and 105% loads, it obtained dramatic performance improvement for both queue reference values and significantly outperformed drop-tail at these high loads.

4.5.3 Results for ARED/ECN

Figures 4.37, 4.38, 4.39, and 4.40 show the performance of ARED with and without ECN and compare it with the performance of drop-tail at 80%, 90%, 98%, and 105% loads. The performance for ARED was obtained with a target queue threshold of 24 and 240 packets by choosing appropriate values for th_{min} and th_{max} as discussed in section 4.2.

Contrary to PI and REM, ARED did not benefit from the addition of ECN and the performance of ARED with ECN was essentially the same as without ECN. Furthermore, ARED consistently performed poorer than drop-tail. These observations hold for ARED at all loads. The reason for the poor performance of ARED without and with ECN are explored in sections 4.6 and 4.7.

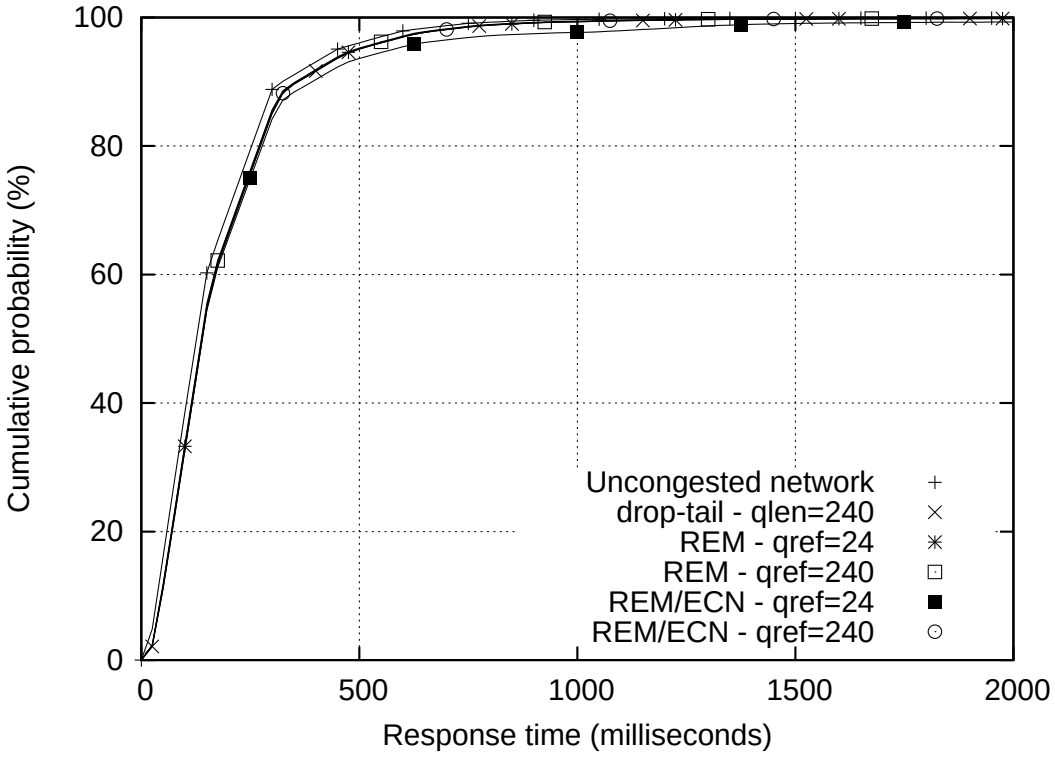

Figure 4.29: REM/ECN performance at 80% load

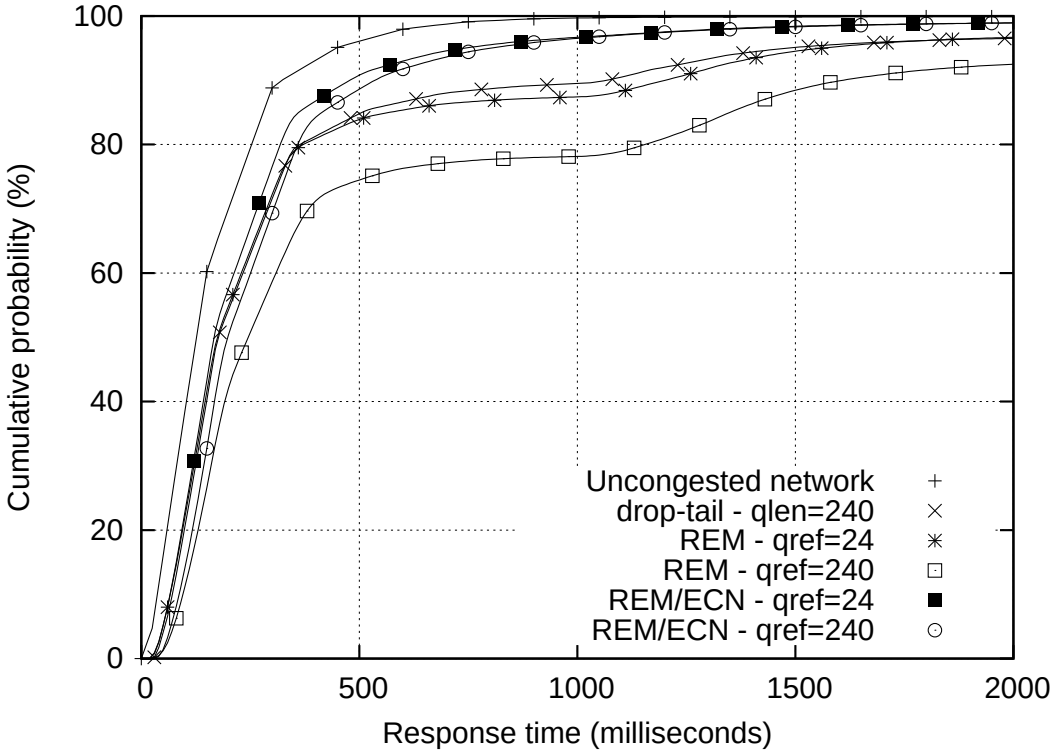

Figure 4.30: REM/ECN performance at 90% load

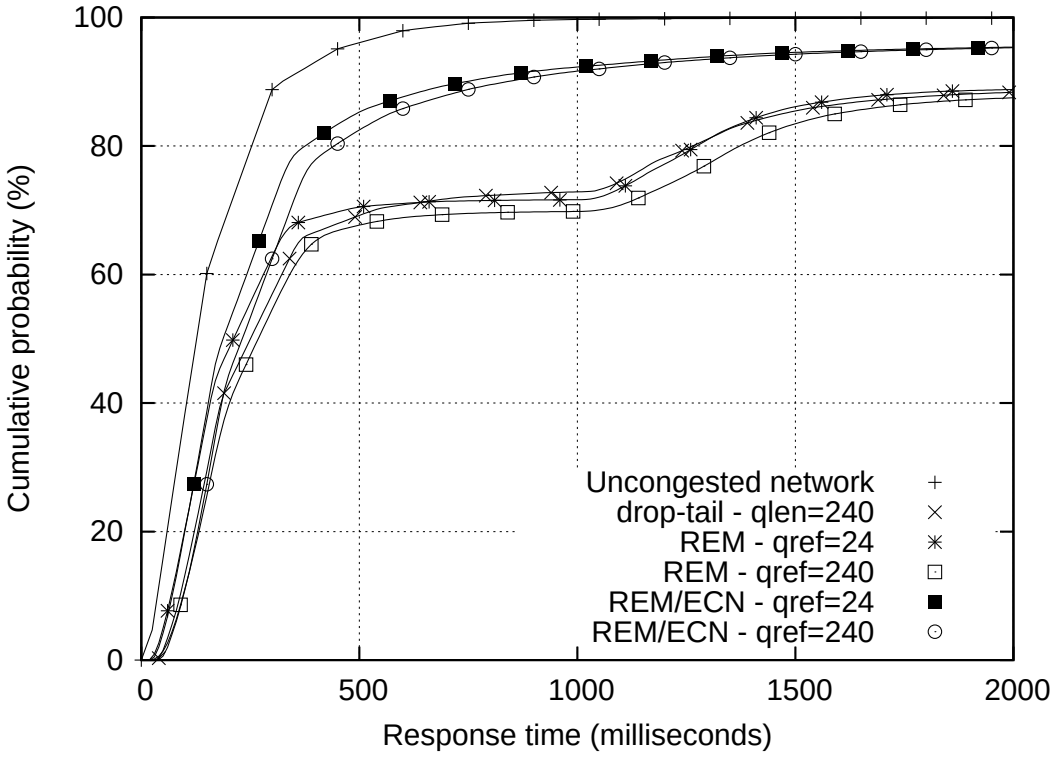

Figure 4.31: REM/ECN performance at 98% load

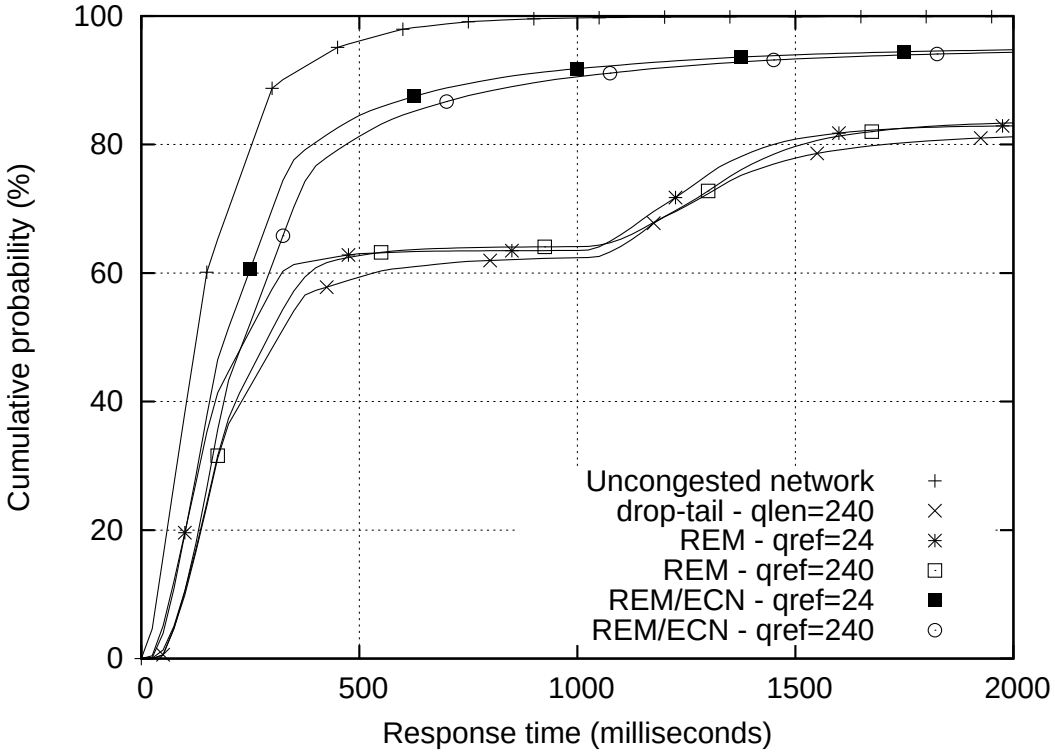

Figure 4.32: REM/ECN performance at 105% load

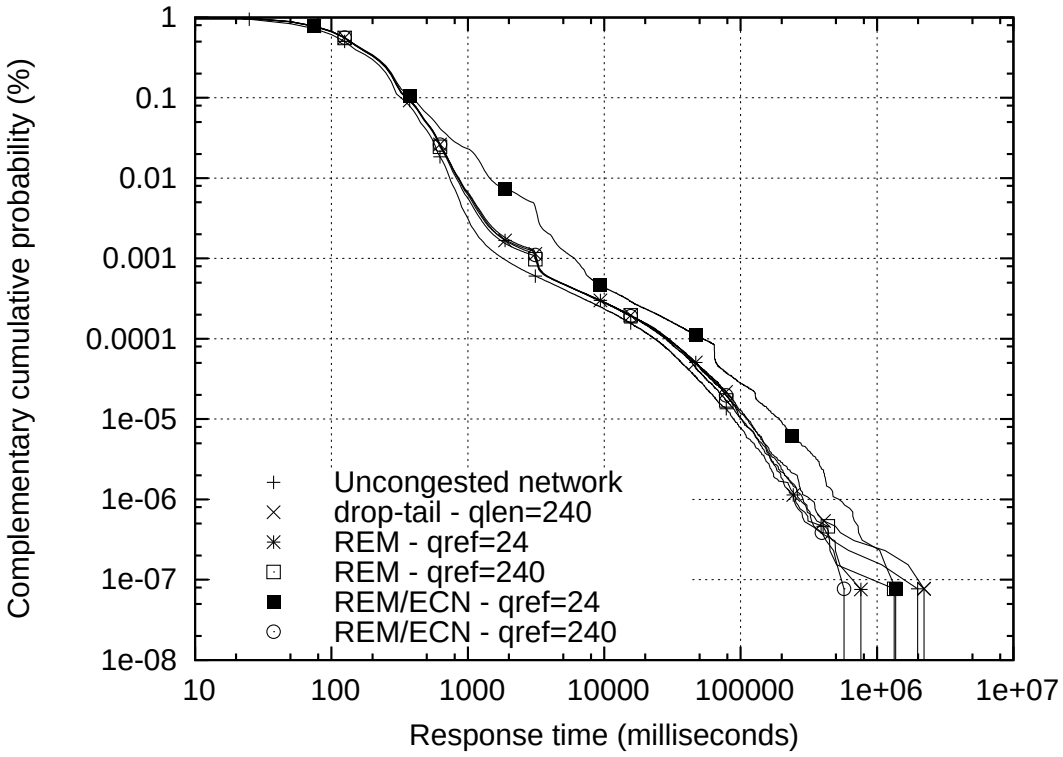

Figure 4.33: REM/ECN performance at 80% load (CCDF)

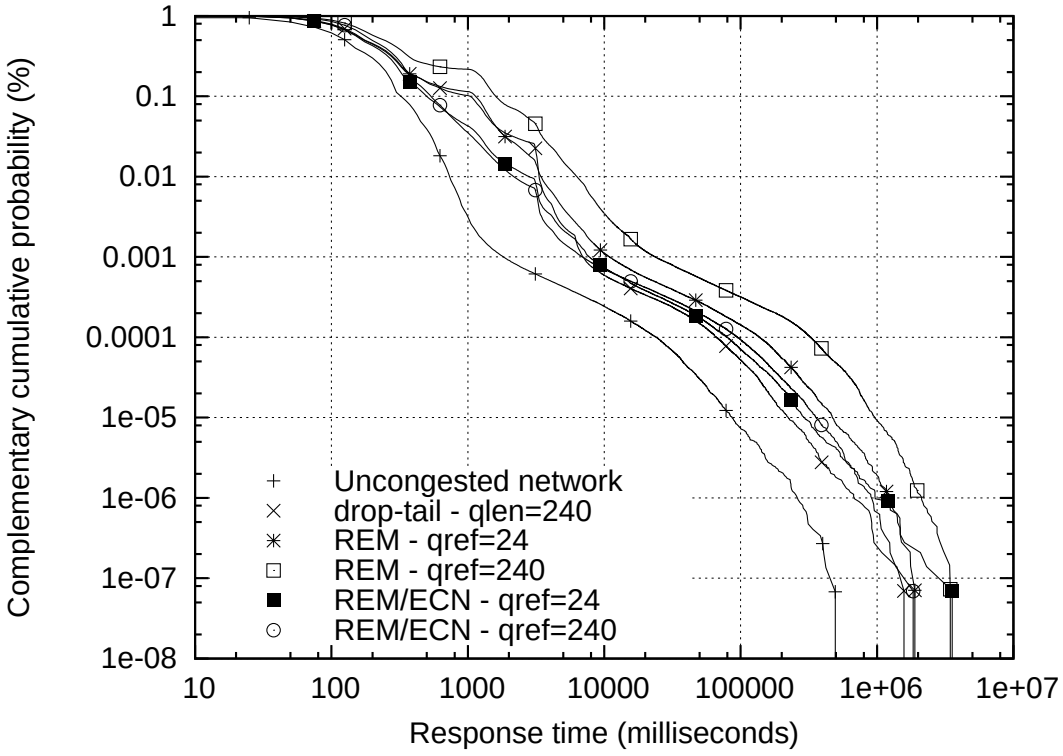

Figure 4.34: REM/ECN performance at 90% load (CCDF)

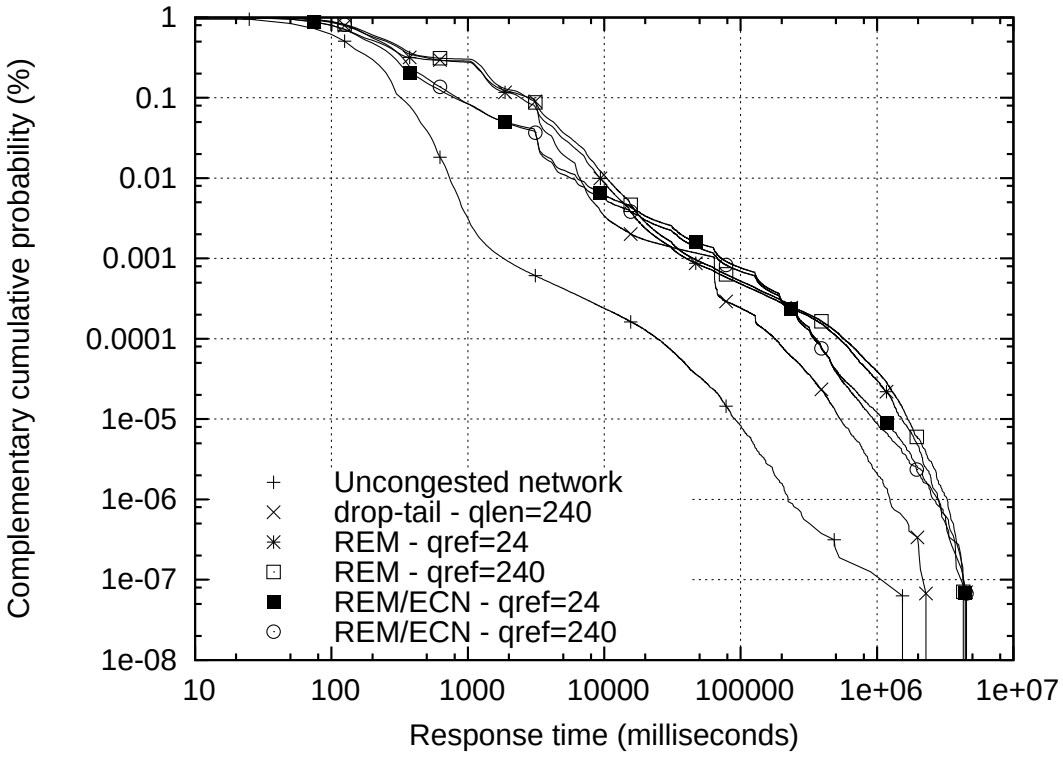

Figure 4.35: REM/ECN performance at 98% load (CCDF)

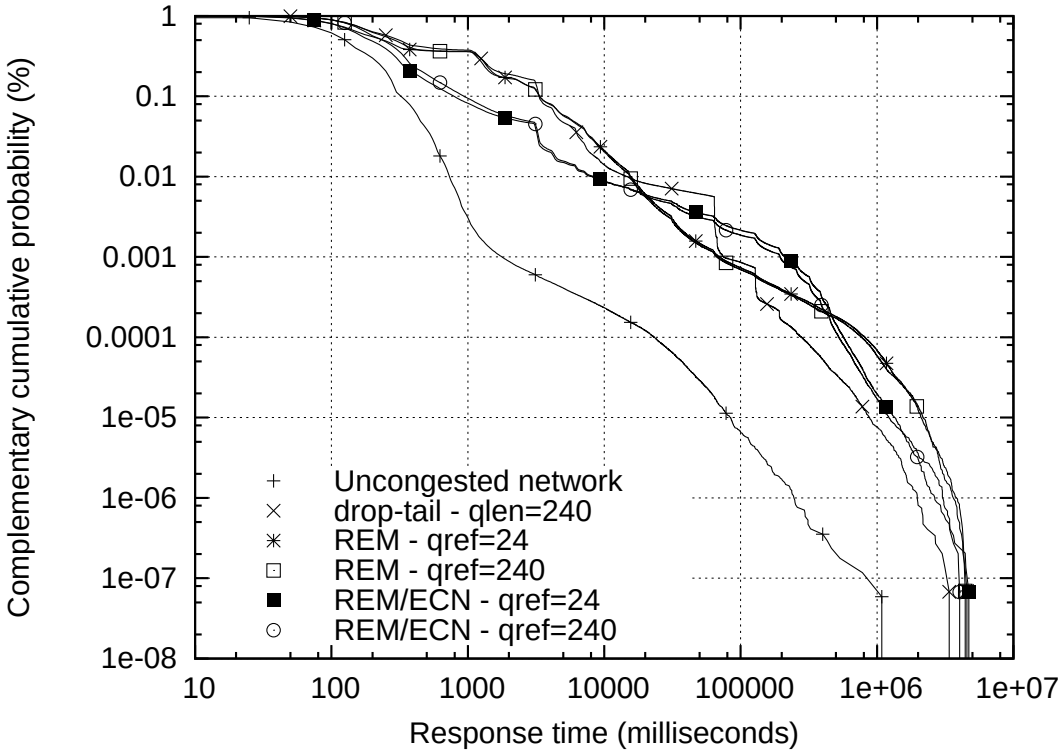

Figure 4.36: REM/ECN performance at 105% load (CCDF)

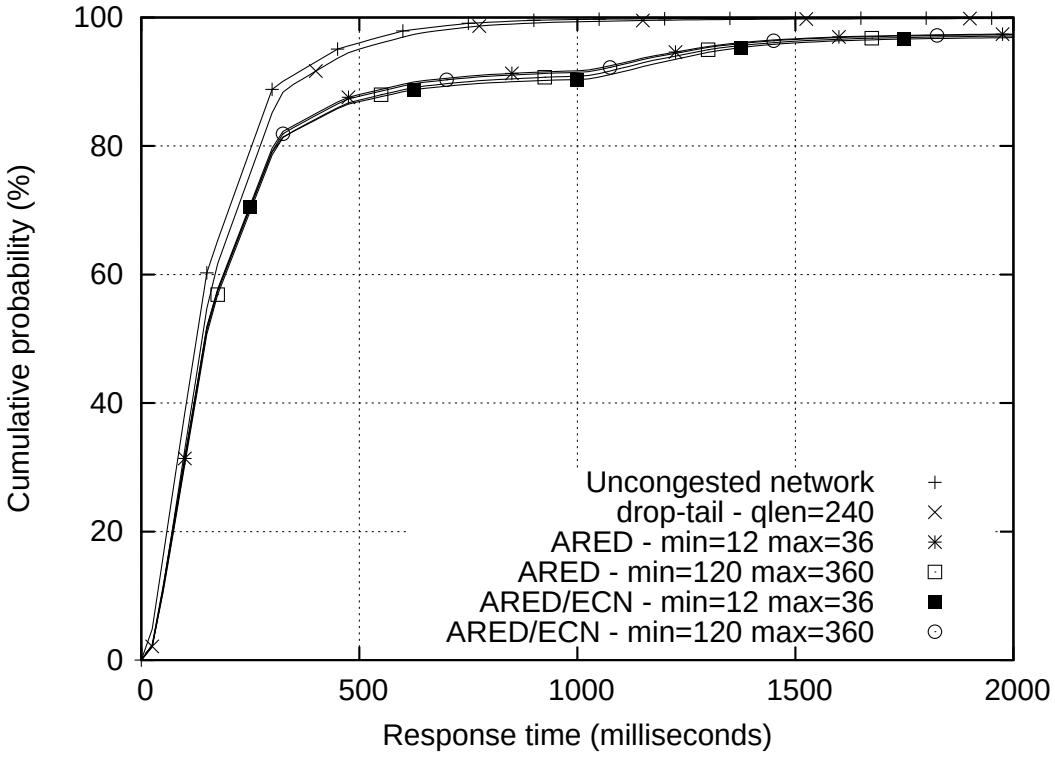

Figure 4.37: ARED/ECN performance at 80% load

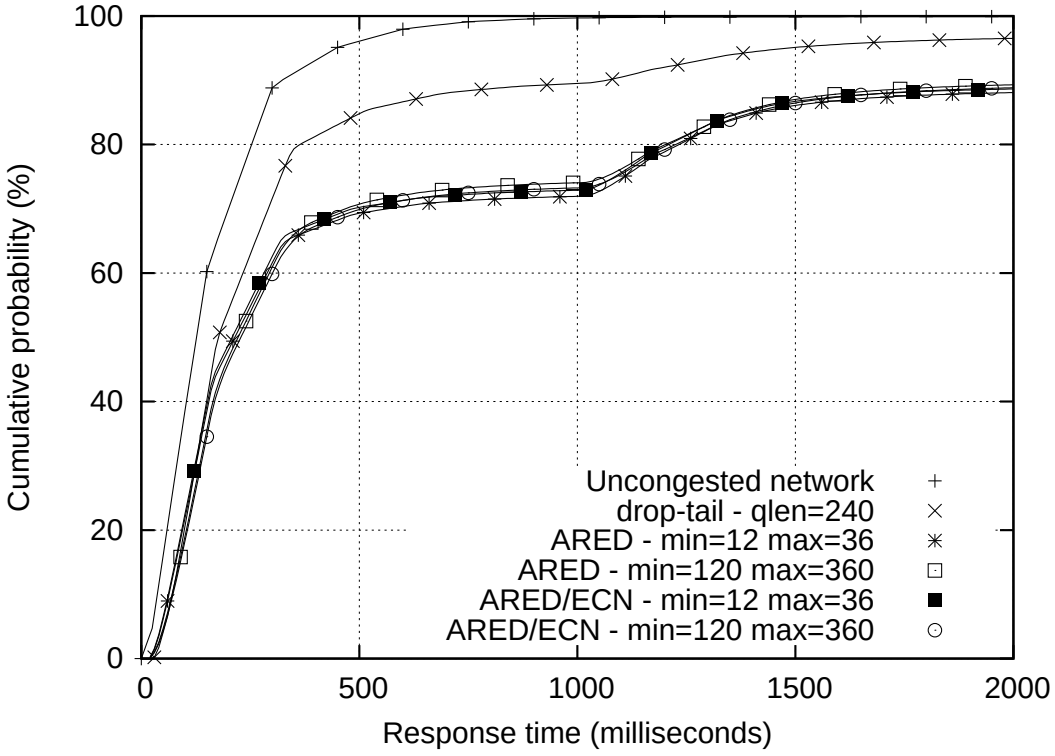

Figure 4.38: ARED/ECN performance at 90% load

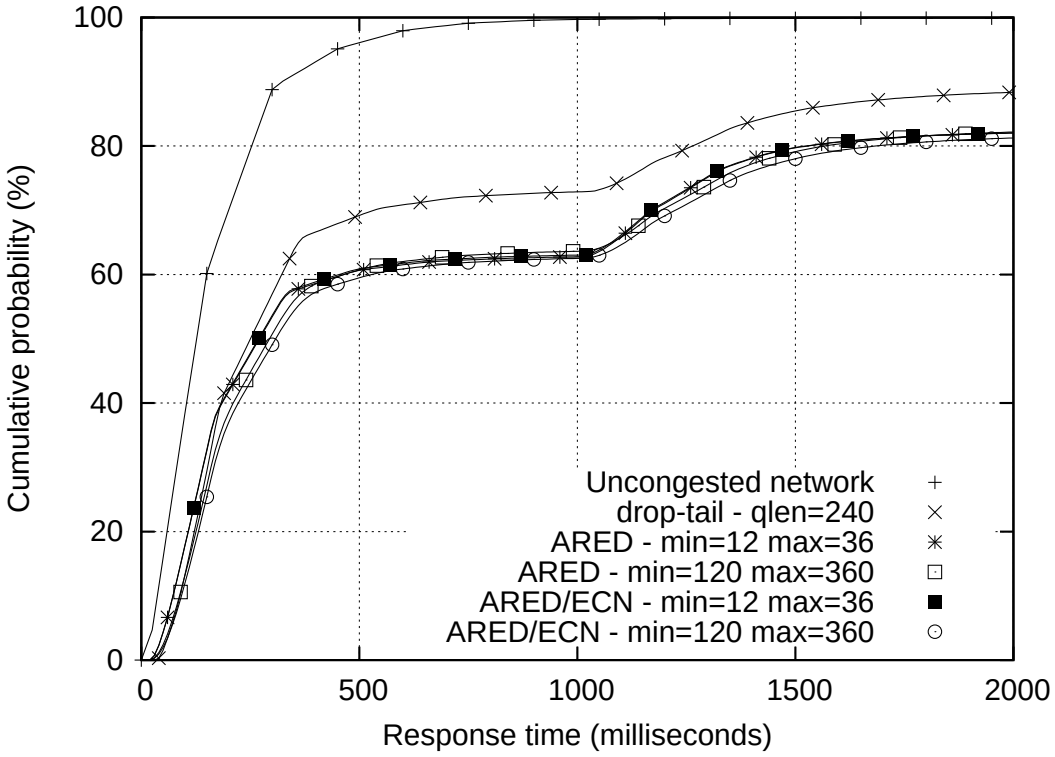

Figure 4.39: ARED/ECN performance at 98% load

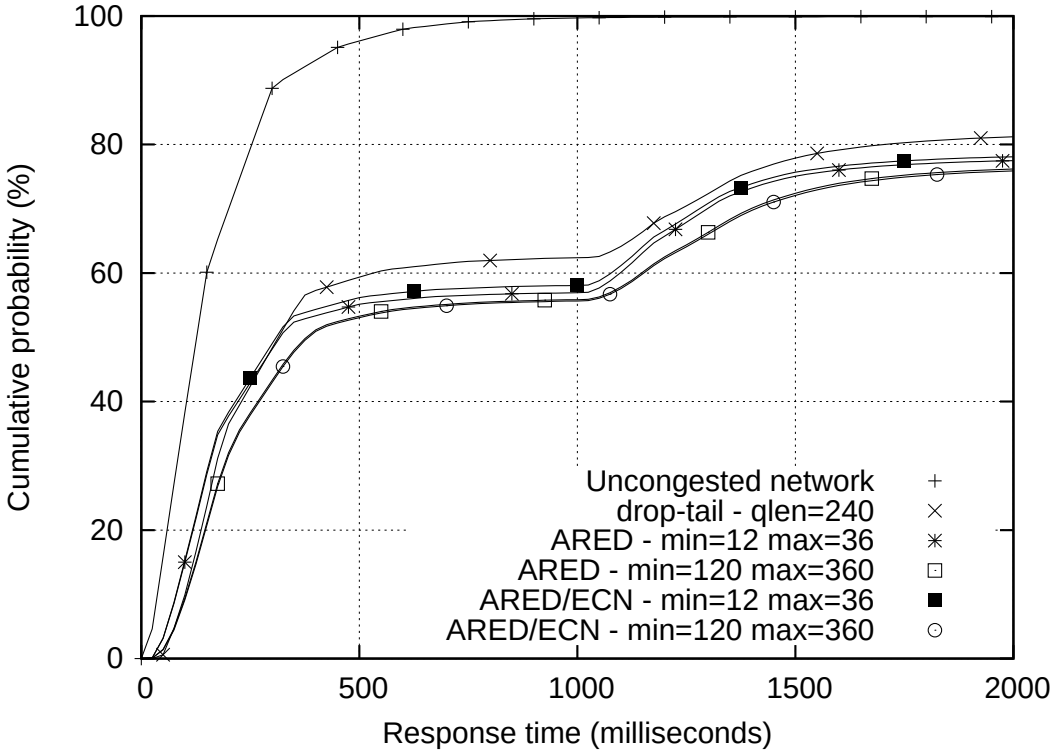

Figure 4.40: ARED/ECN performance at 105% load

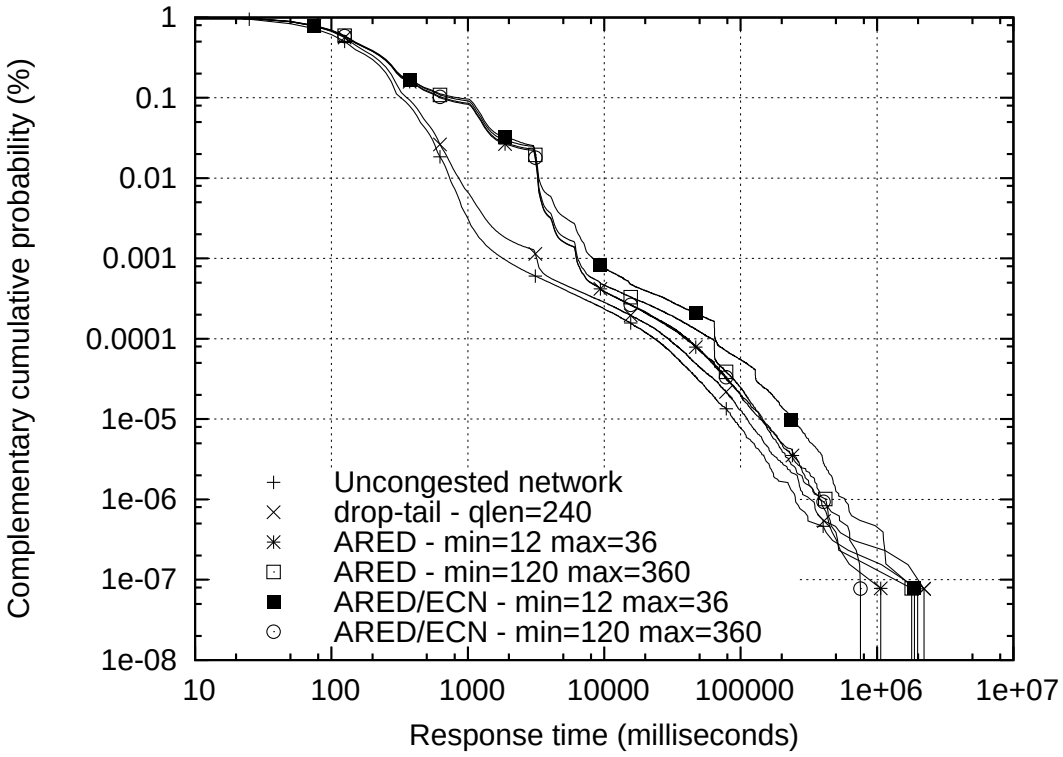

Figure 4.41: ARED/ECN performance at 80% load (CCDF)

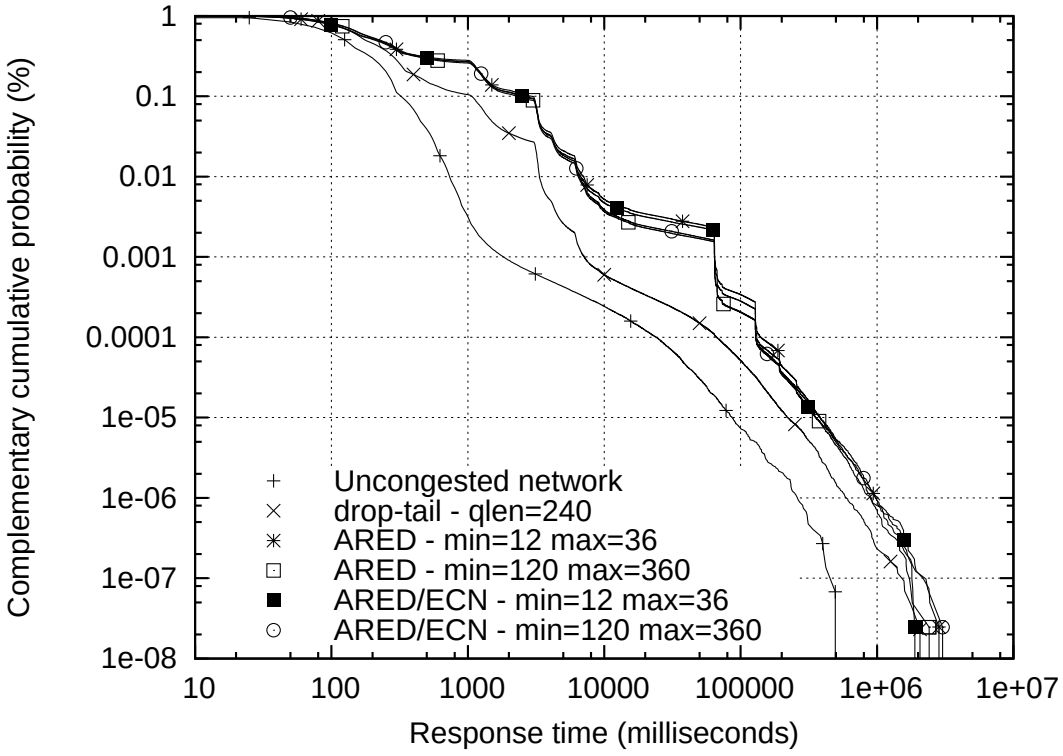

Figure 4.42: ARED/ECN performance at 90% load (CCDF)

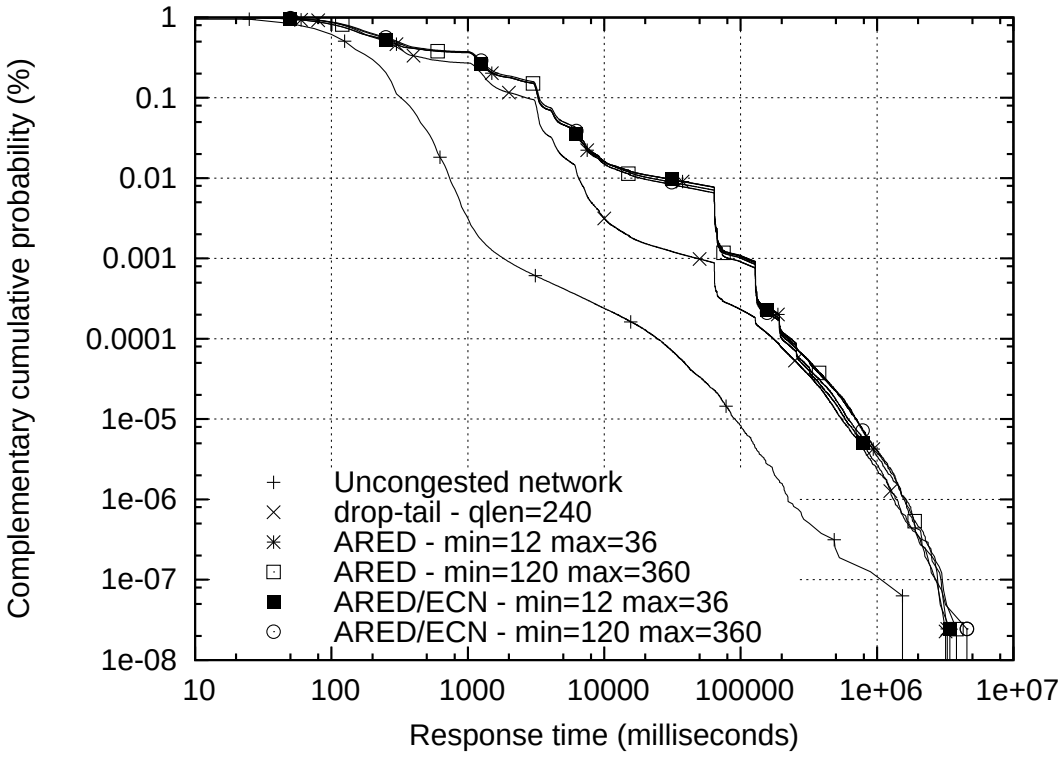

Figure 4.43: ARED/ECN performance at 98% load (CCDF)

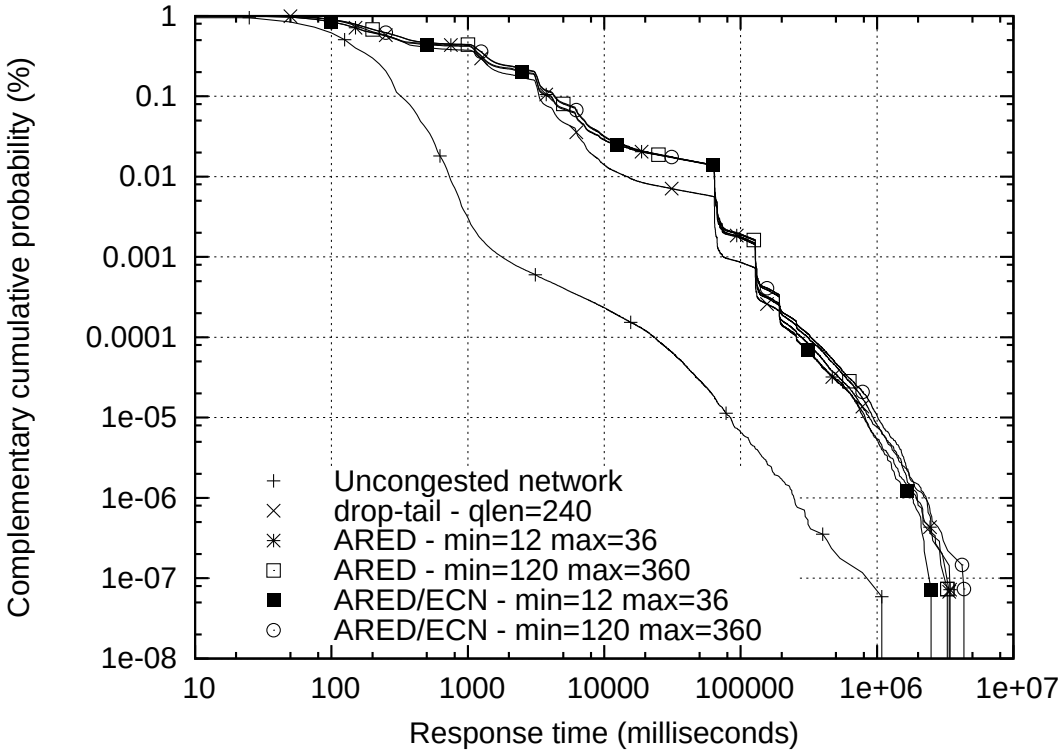

Figure 4.44: ARED/ECN performance at 105% load (CCDF)

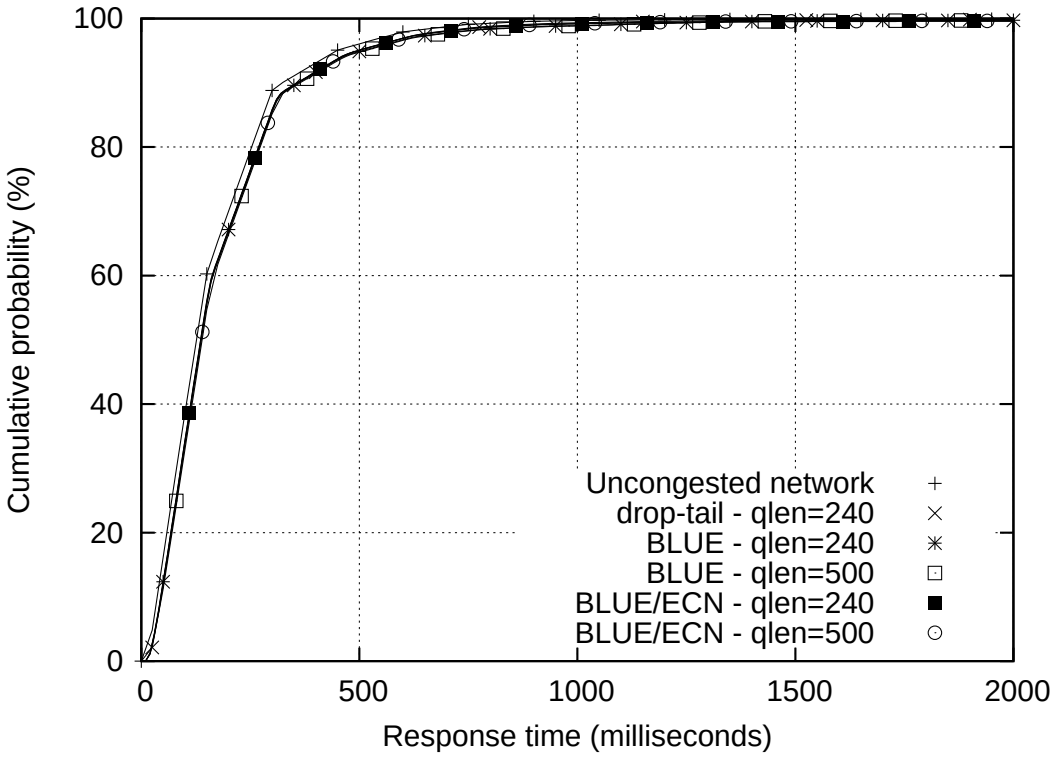

Figure 4.45: BLUE/ECN performance at 80% load

4.5.4 Results for BLUE/ECN

Figures 4.45, 4.46, 4.47, and 4.48 show the performance of BLUE with and without ECN and compare it with the performance of drop-tail at 80%, 90%, 98%, and 105% loads. The performance for BLUE was obtained with a router's queue size of 240 and 500 packets.

At 80% load, BLUE obtained comparable performance with drop-tail and closely approximated the performance of the uncongested network. The addition of ECN did not improve the performance for BLUE at this load.

At 90% offered load, BLUE without ECN and with a queue length of 240 packets performed slightly better than drop-tail. However, BLUE without ECN and with a queue length of 500 packets delivered slightly worse performance than drop-tail. When BLUE was used with ECN, it obtained a small performance improvement with both queue lengths and gave better performance than drop-tail.

As the offered load increased to 98% and 105%, BLUE without ECN and with a queue length of 240 and 500 packets gave poorer performance than drop-tail. However, when BLUE was used with ECN, it obtained significant performance improvement with both queue lengths and outperformed drop-tail.

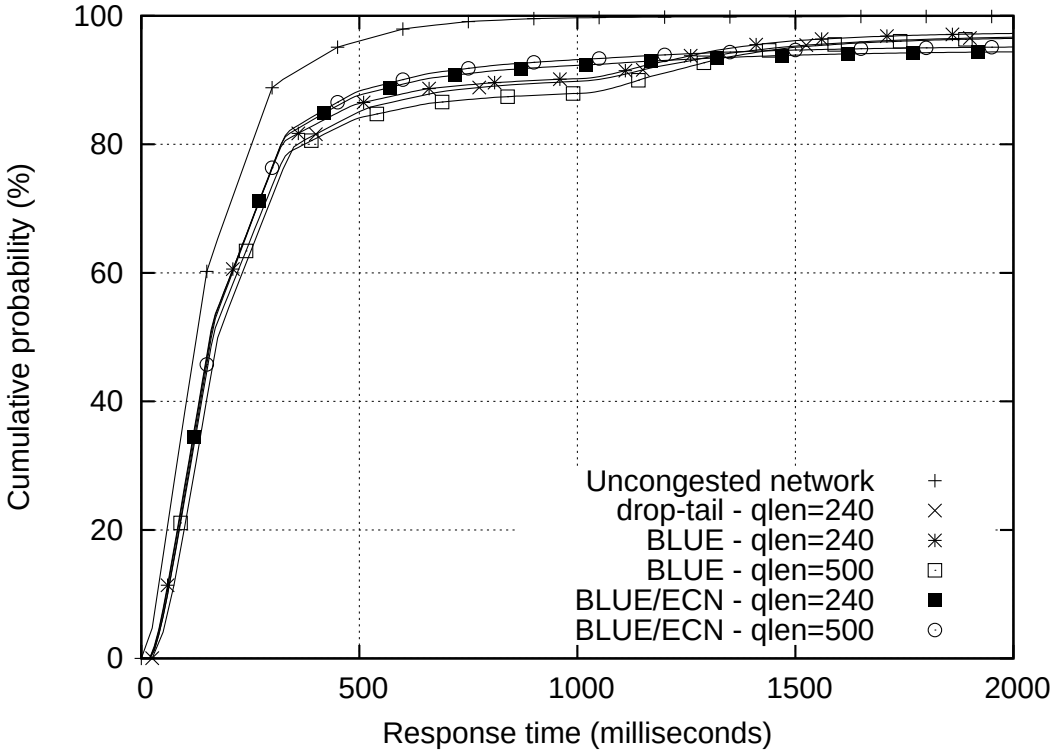

Figure 4.46: BLUE/ECN performance at 90% load

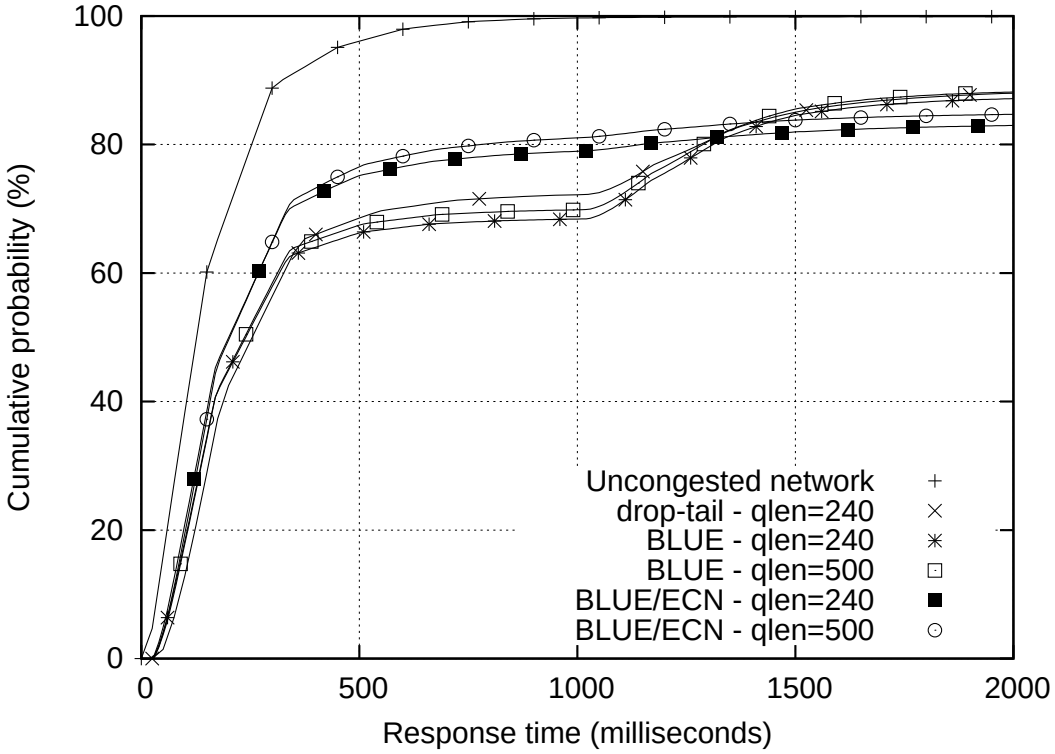

Figure 4.47: BLUE/ECN performance at 98% load

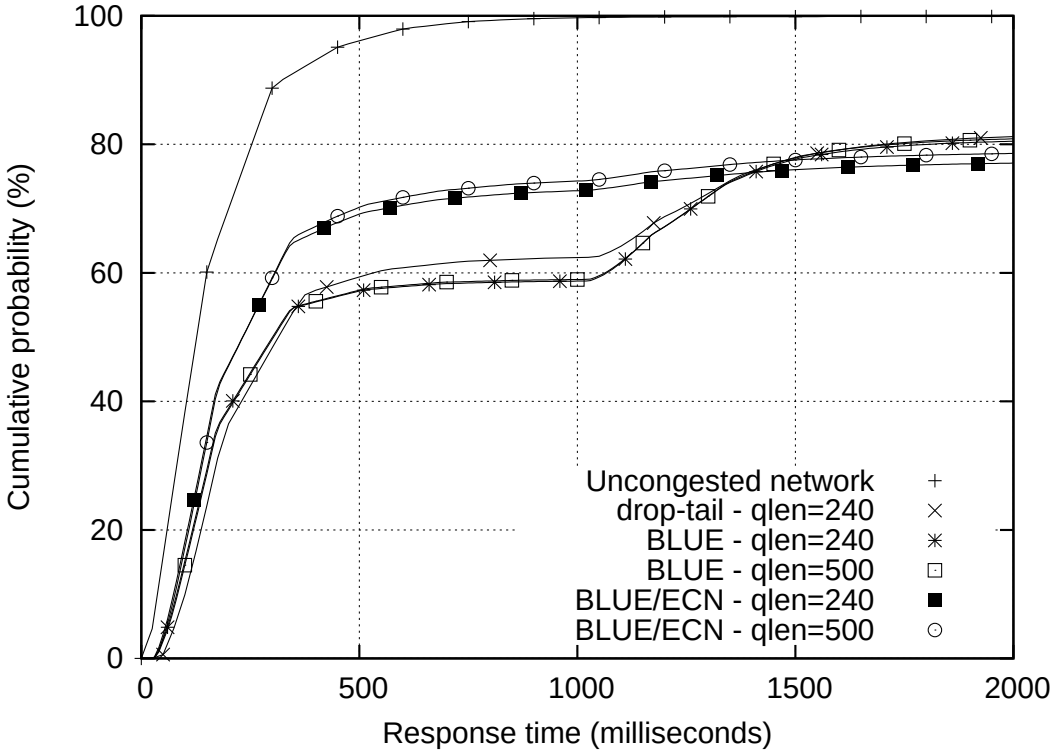

Figure 4.48: BLUE/ECN performance at 105% load

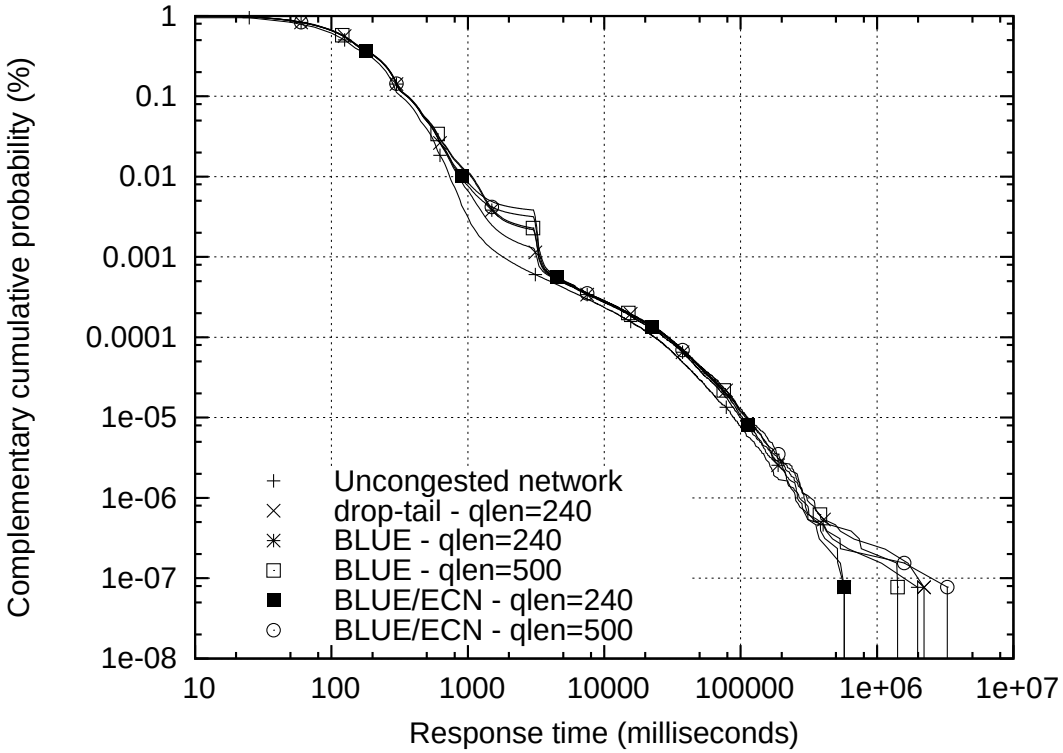

Figure 4.49: BLUE/ECN performance at 80% load (CCDF)

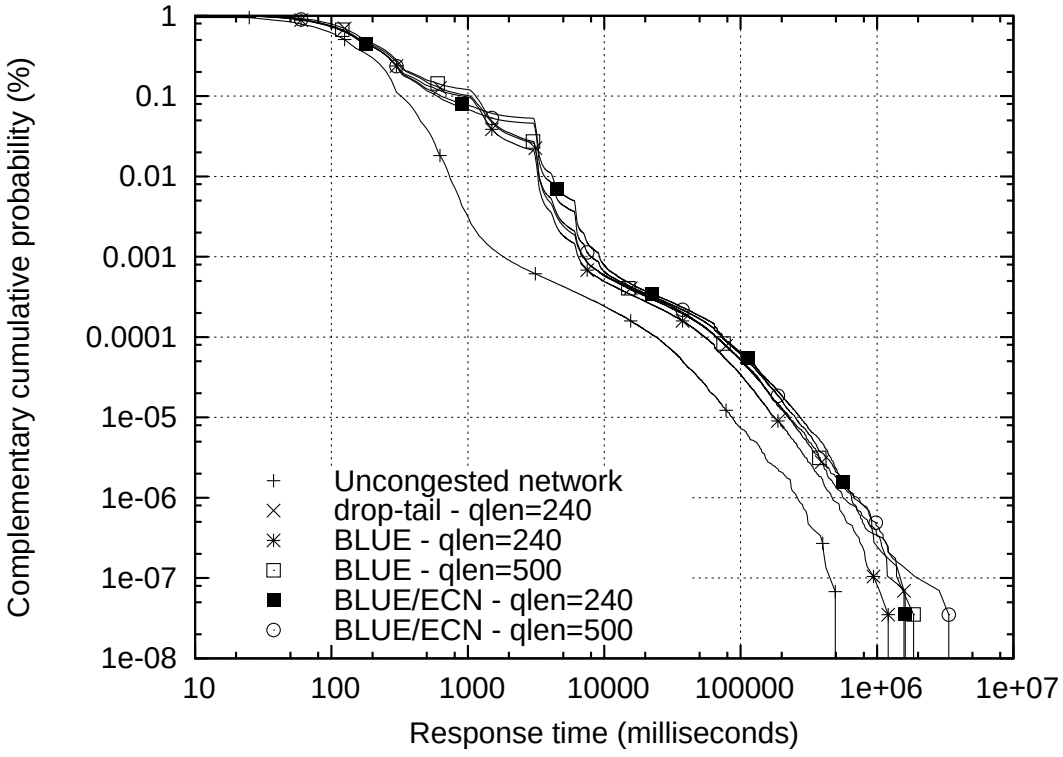

Figure 4.50: BLUE/ECN performance at 90% load (CCDF)

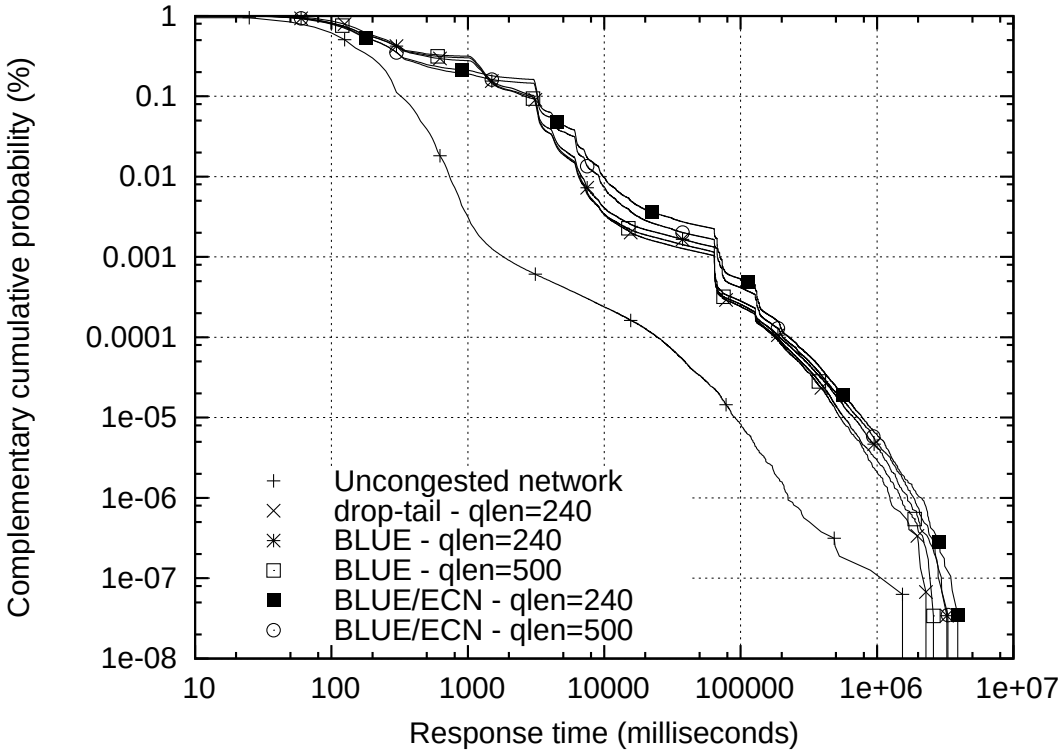

Figure 4.51: BLUE/ECN performance at 98% load (CCDF)

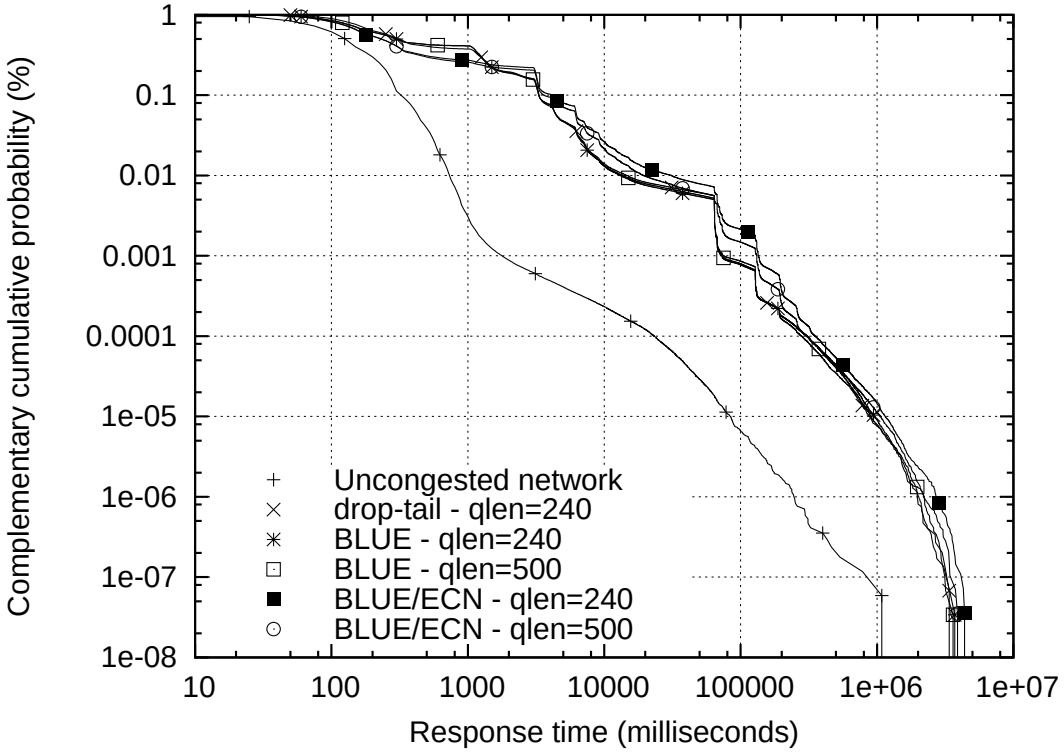

Figure 4.52: BLUE/ECN performance at 105% load (CCDF)

4.5.5 Results for AVQ/ECN

Figures 4.53, 4.54, 4.55, and 4.56 present the performance of AVQ with and without ECN and compare it with the performance of drop-tail at 80%, 90%, 98%, and 105% loads. The performance for AVQ was obtained with a virtual queue length of 240 and 500 packets.

At 80% load, AVQ without ECN and with a virtual queue length of 240 and 500 packets gave slightly worse performance than drop-tail. However, when AVQ was used with ECN, it obtained considerable performance improvement and delivered comparable performance with both virtual queue lengths as drop-tail.

At 90% load, AVQ without ECN gave comparable performance with both virtual queue lengths. Without ECN, AVQ delivered the same performance as drop-tail for approximately 75% of flows but gave poorer performance than drop-tail for the rest 25% of flows. When AVQ was used with ECN, it obtained dramatic performance improvement with both virtual queue lengths and significantly outperformed drop-tail at this load.

At 98% load, AVQ without ECN again obtained comparable performance with both virtual queue lengths. Without ECN, AVQ gave better performance than drop-tail for approximately 70% of flows and obtained comparable performance as drop-tail for the rest 30% of flows. Interestingly, when AVQ was used with ECN, it suffered significant perfor-

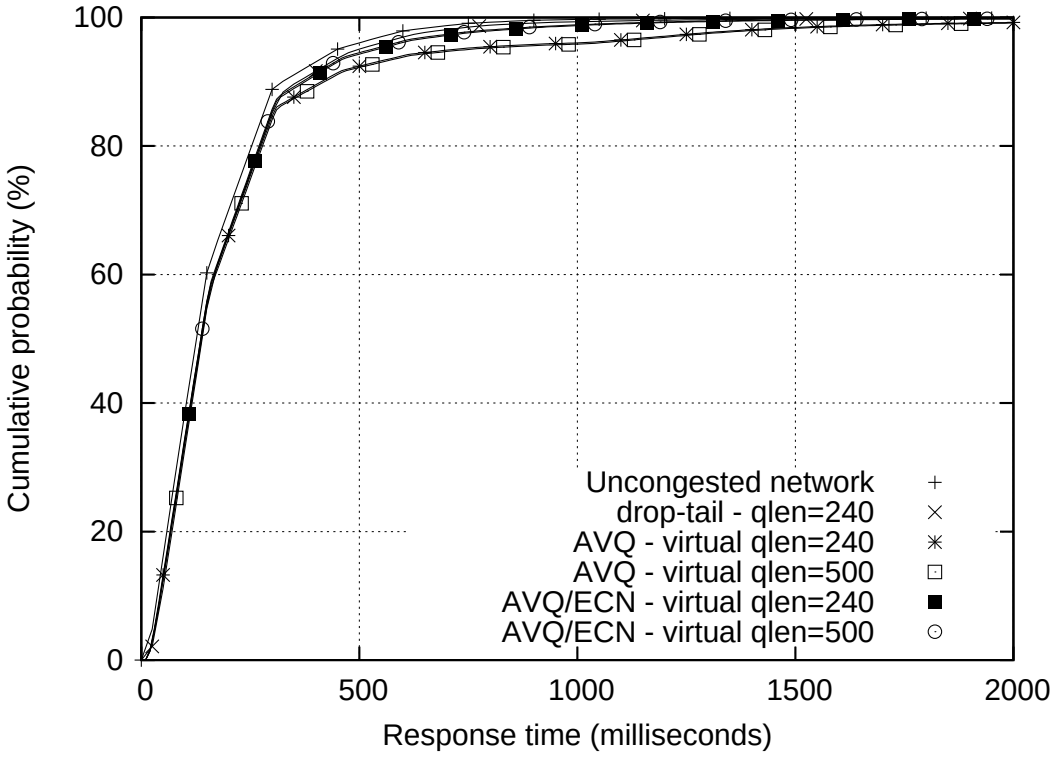

Figure 4.53: AVQ/ECN performance at 80% load

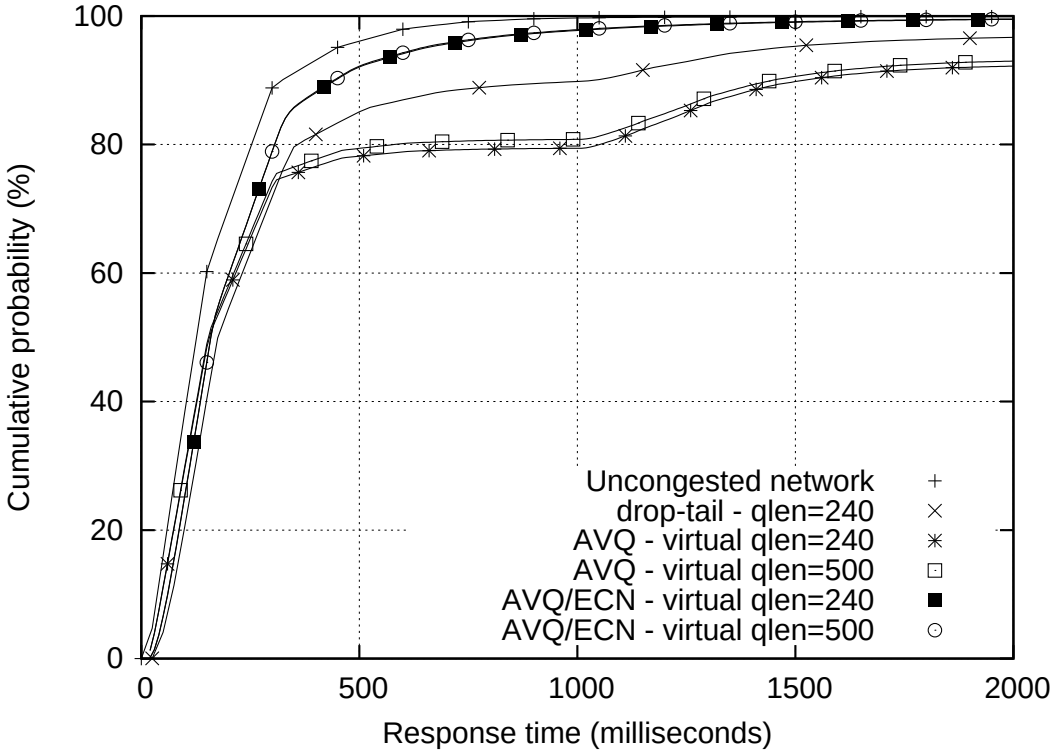

Figure 4.54: AVQ/ECN performance at 90% load

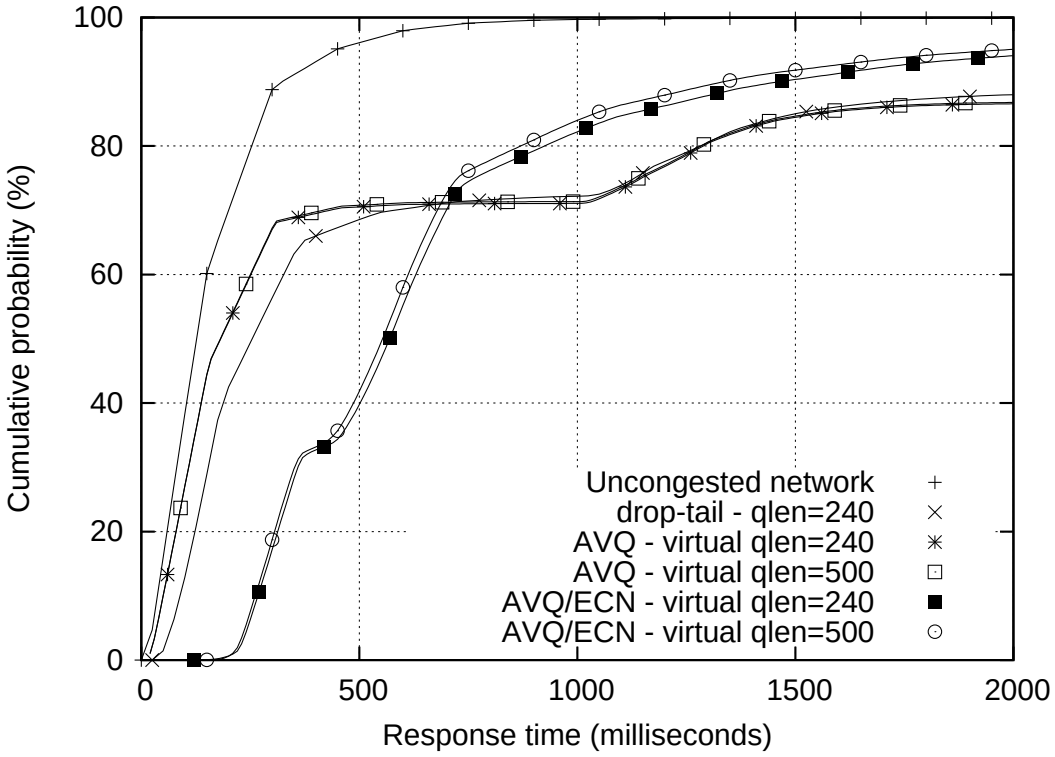

Figure 4.55: AVQ/ECN performance at 98% load

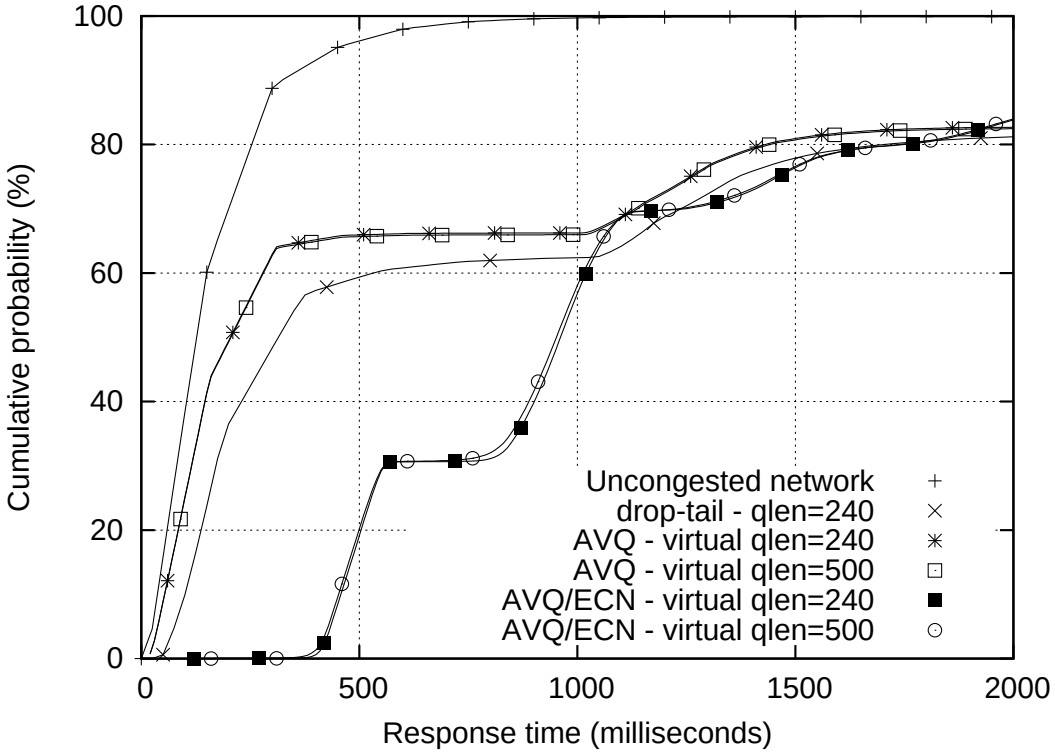

Figure 4.56: AVQ/ECN performance at 105% load

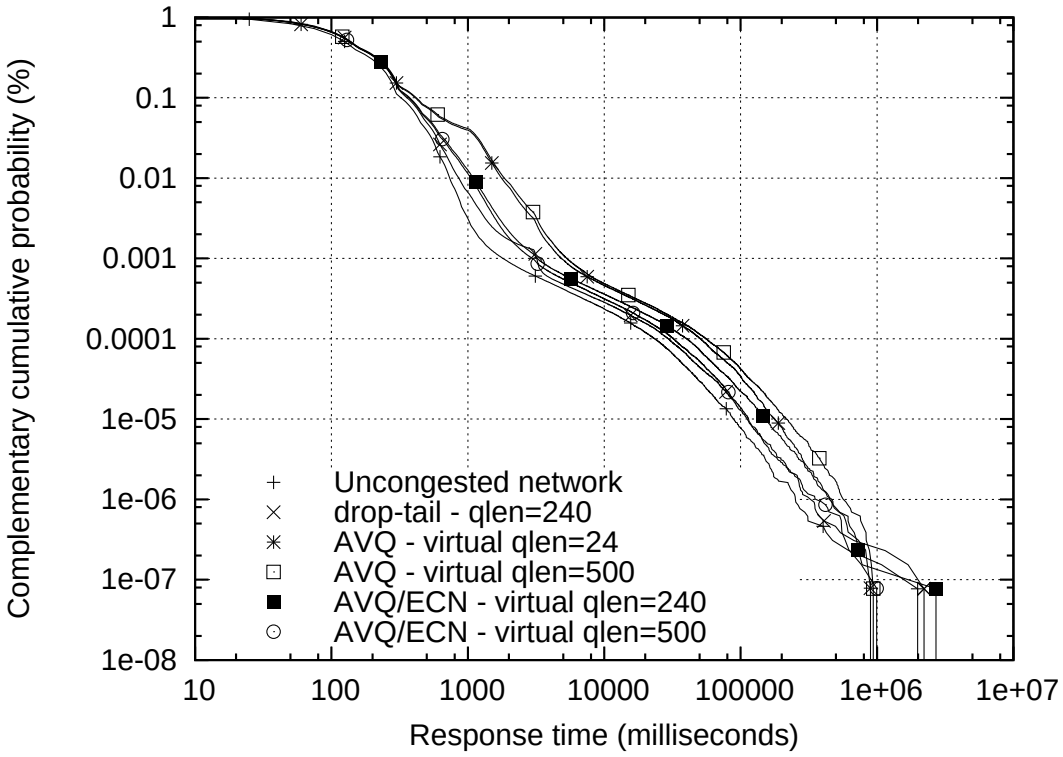

Figure 4.57: AVQ/ECN performance at 80% load (CCDF)

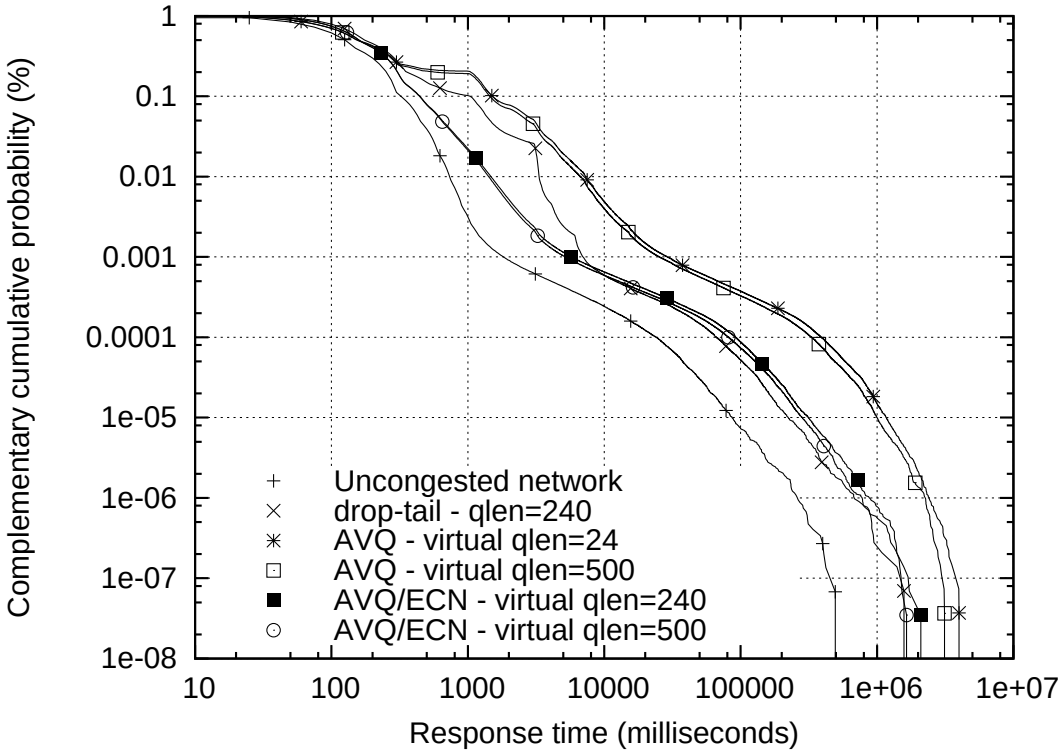

Figure 4.58: AVQ/ECN performance at 90% load (CCDF)

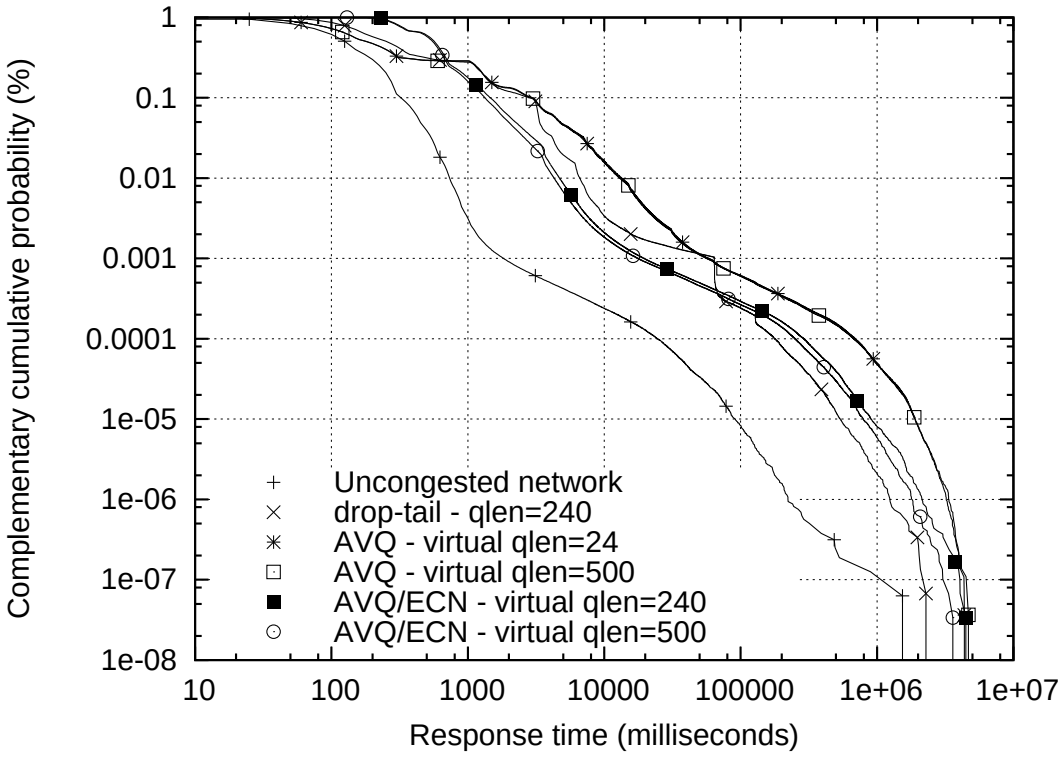

Figure 4.59: AVQ/ECN performance at 98% load (CCDF)

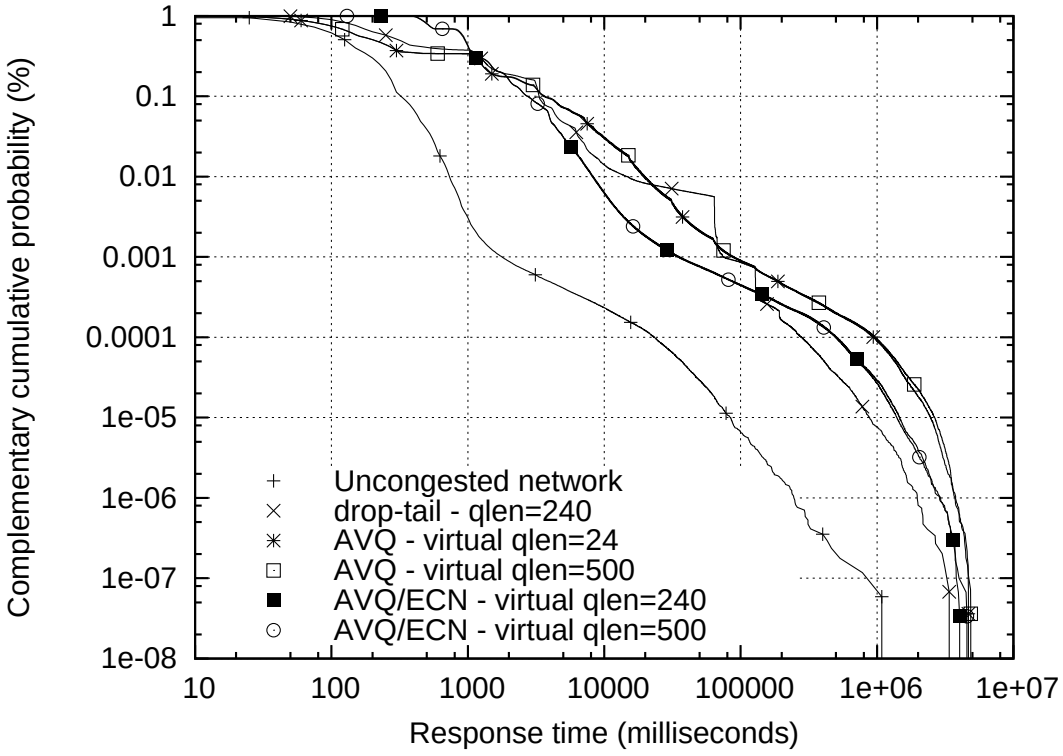

Figure 4.60: AVQ/ECN performance at 105% load (CCDF)

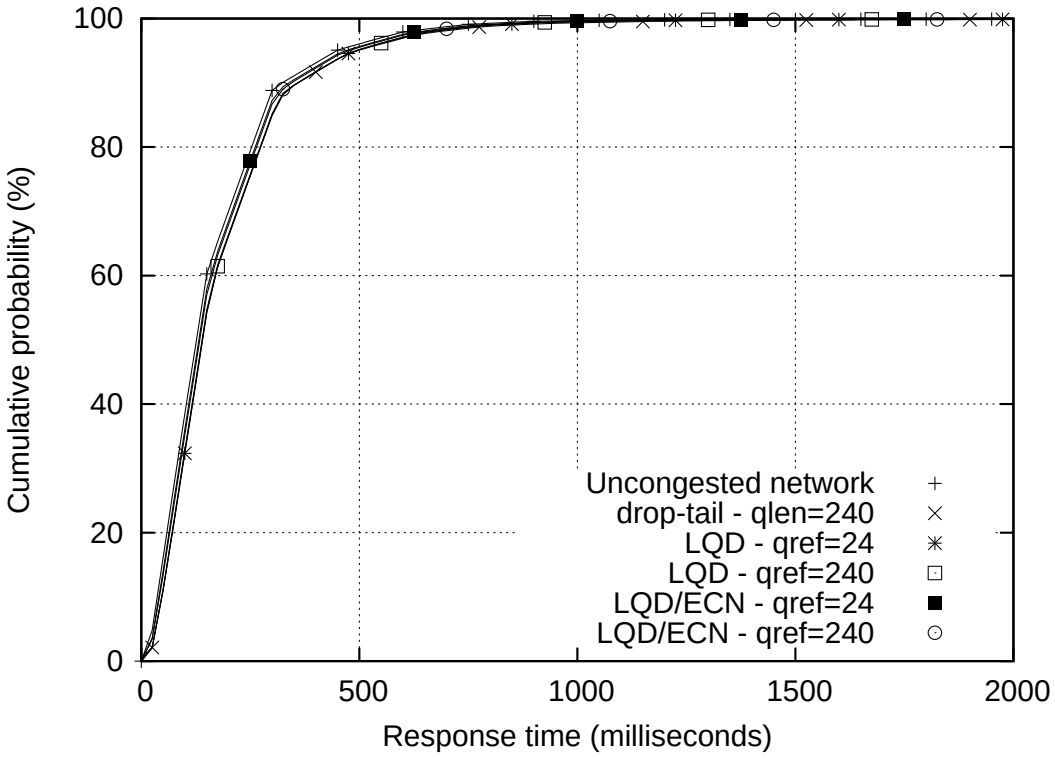

Figure 4.61: LQD/ECN performance at 80% load

mance degradation for 70% of flows but obtained considerable performance improvement for the rest 30% of flows.

At 105% offered load, AVQ without ECN gave comparable performance with both virtual queue lengths. Without ECN, AVQ obtained better performance than drop-tail at this extreme load. However, the addition of ECN significantly degraded the performance for AVQ at this load. When AVQ was used with ECN, it gave significantly worse performance than drop-tail at 105% load.

It is interesting to note that AVQ with ECN exhibited similar behavior as a drop-tail queue with 2,400 packets at 98% and 105% loads in Figures 4.3 and 4.4. Thus, it can be inferred that AVQ with ECN behaved like a large drop-tail queue at these loads and did not mark or drop packets aggressively enough. This behavior caused excessive queuing delay and significantly increased response times for exchanges of requests and responses.

4.5.6 Results for LQD/ECN

Figures 4.61, 4.62, 4.63, and 4.64 show results for LQD with and without ECN when experiments were performed with Web traffic and a uniform RTT distribution between 10 and 150 milliseconds. These results were obtained for LQD with a queue reference of 24 and 240 packets.

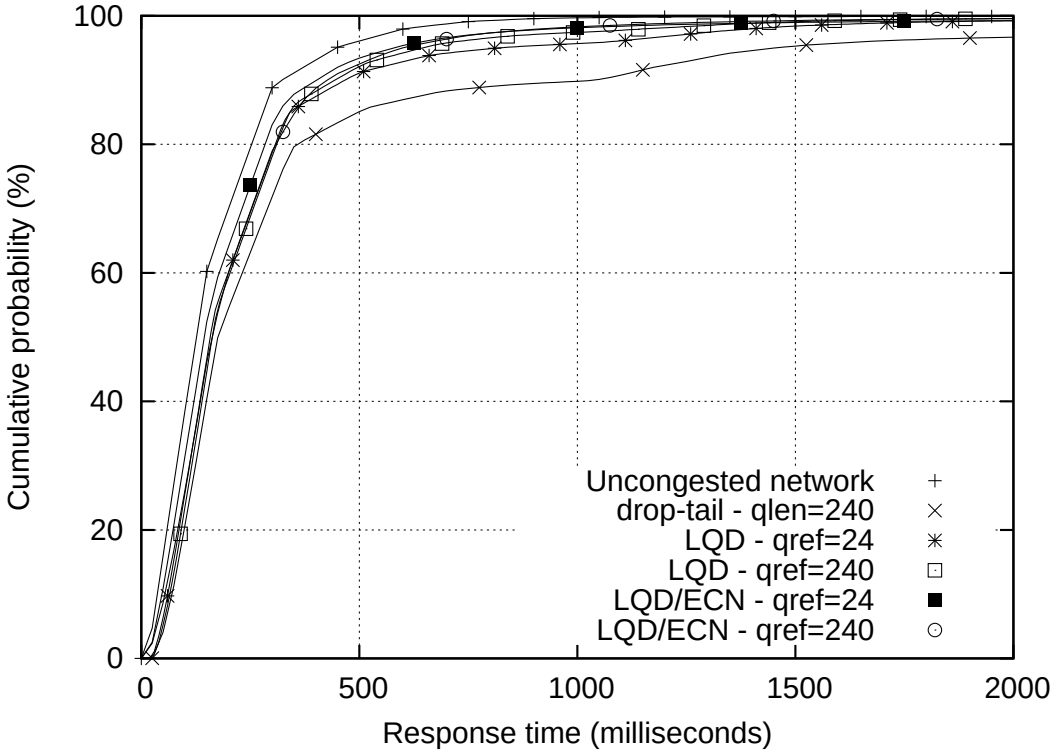

Figure 4.62: LQD/ECN performance at 90% load

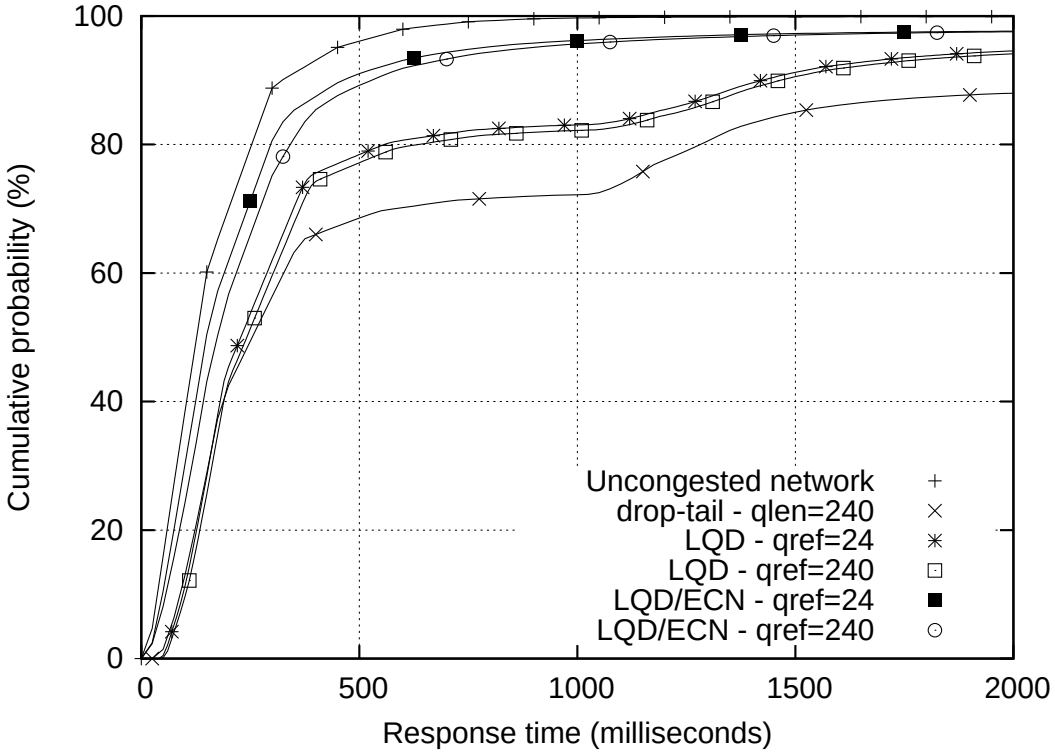

Figure 4.63: LQD/ECN performance at 98% load

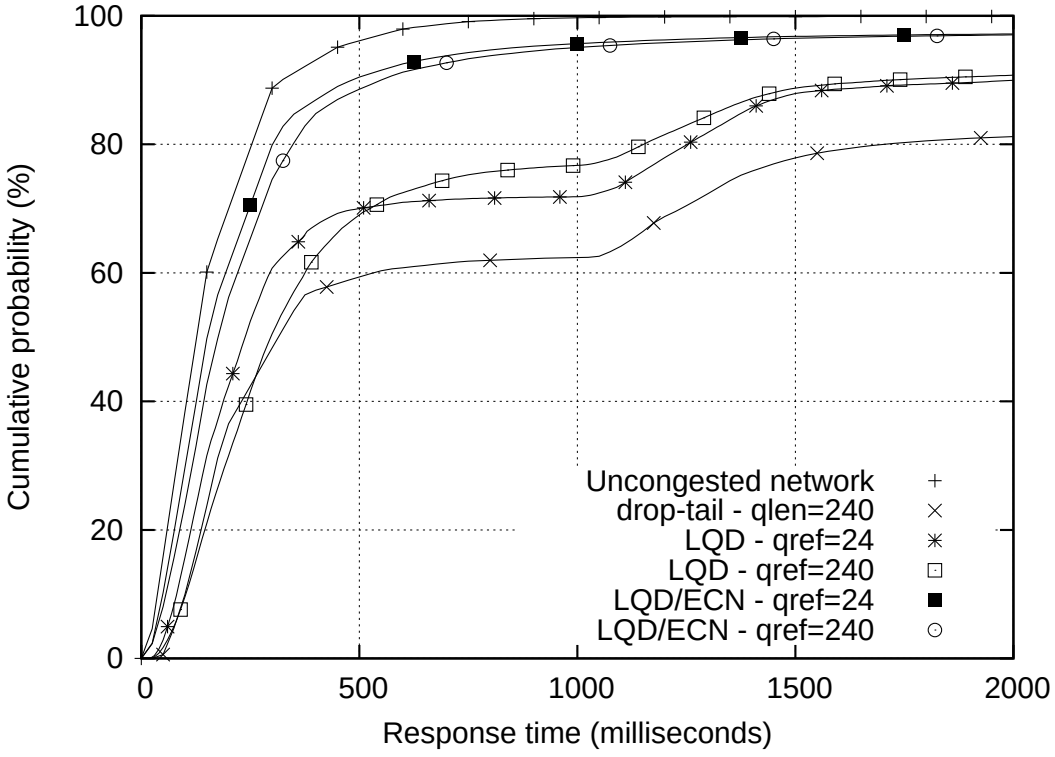

Figure 4.64: LQD/ECN performance at 105% load

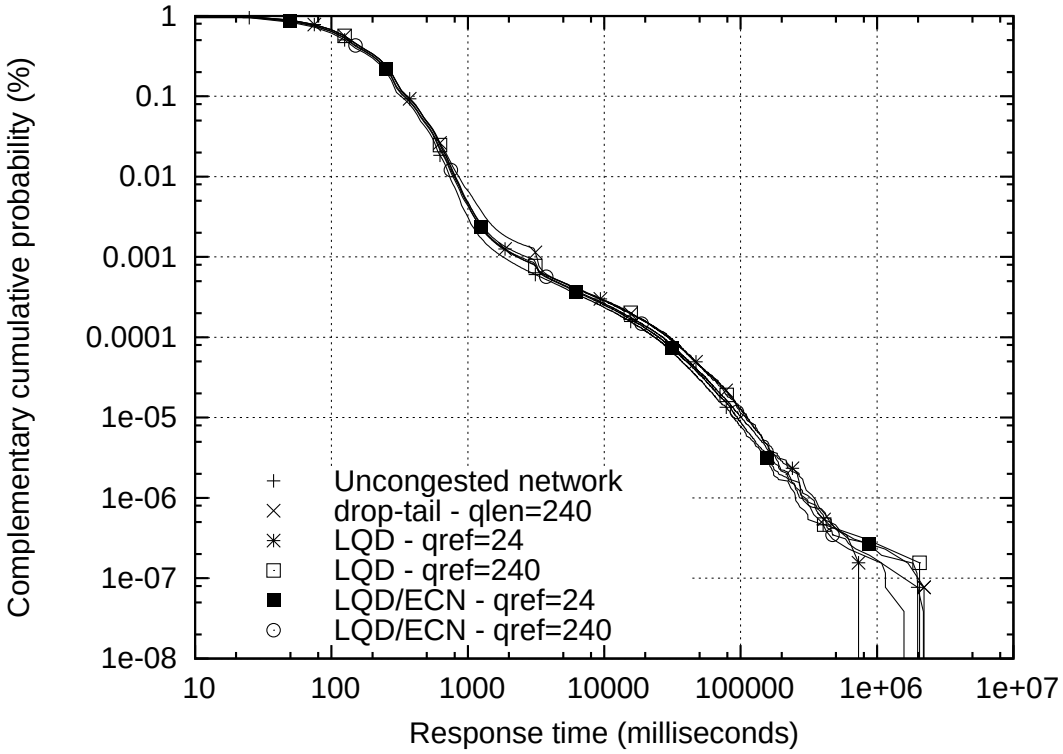

Figure 4.65: LQD/ECN performance at 80% load (CCDF)

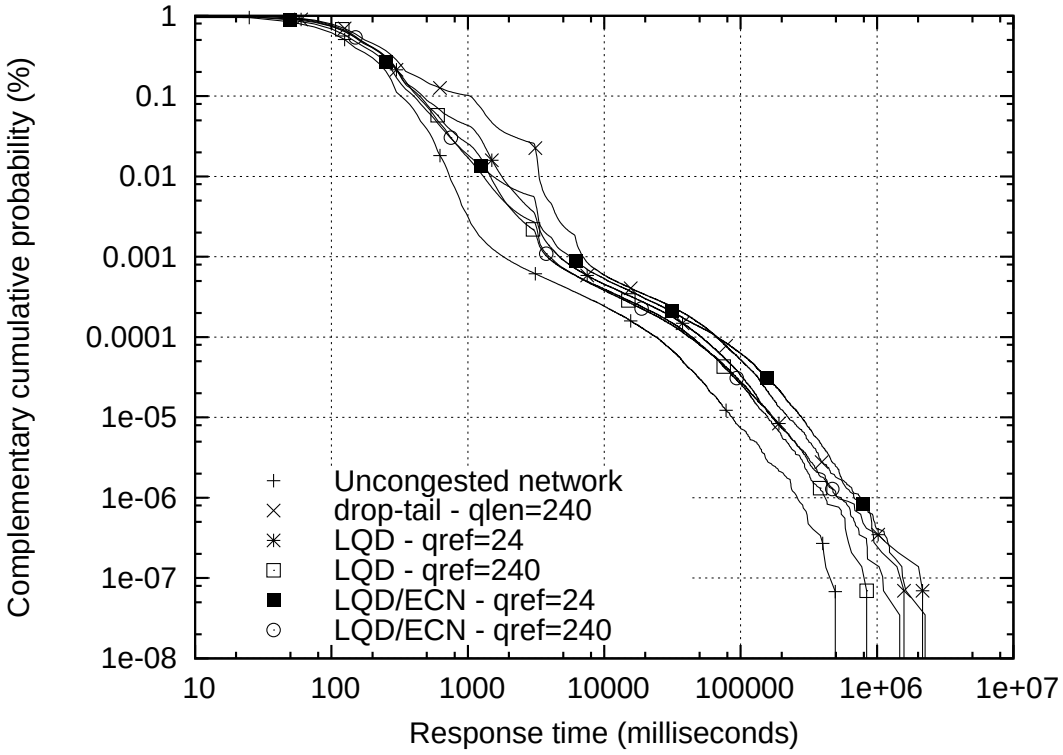

Figure 4.66: LQD/ECN performance at 90% load (CCDF)

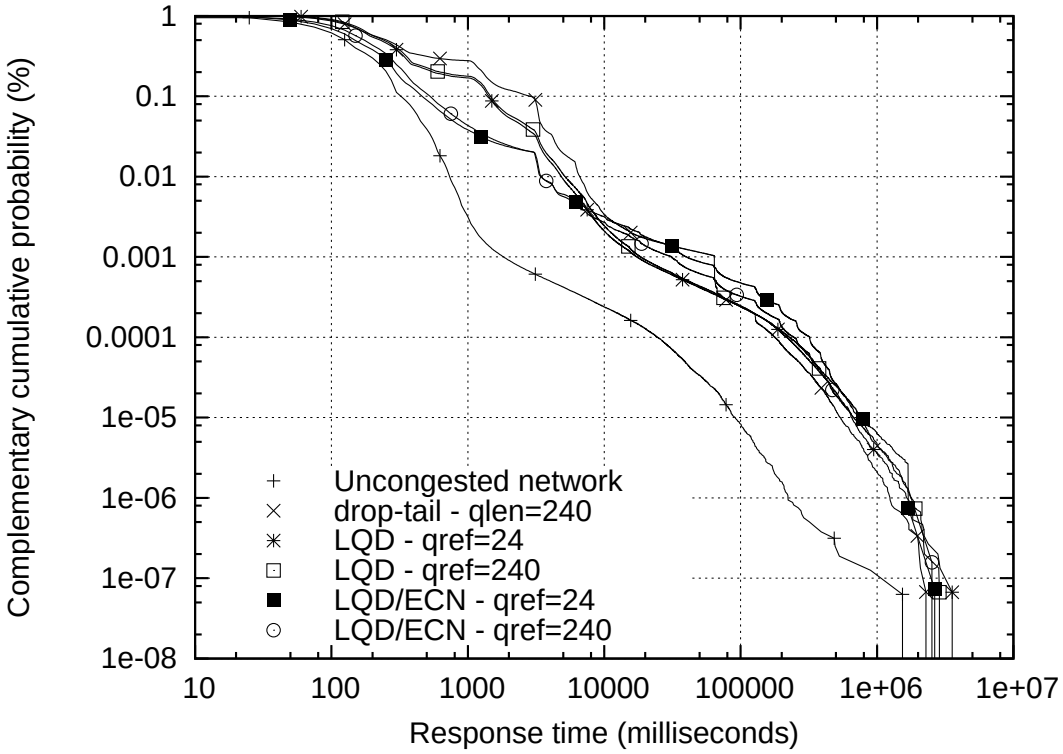

Figure 4.67: LQD/ECN performance at 98% load (CCDF)

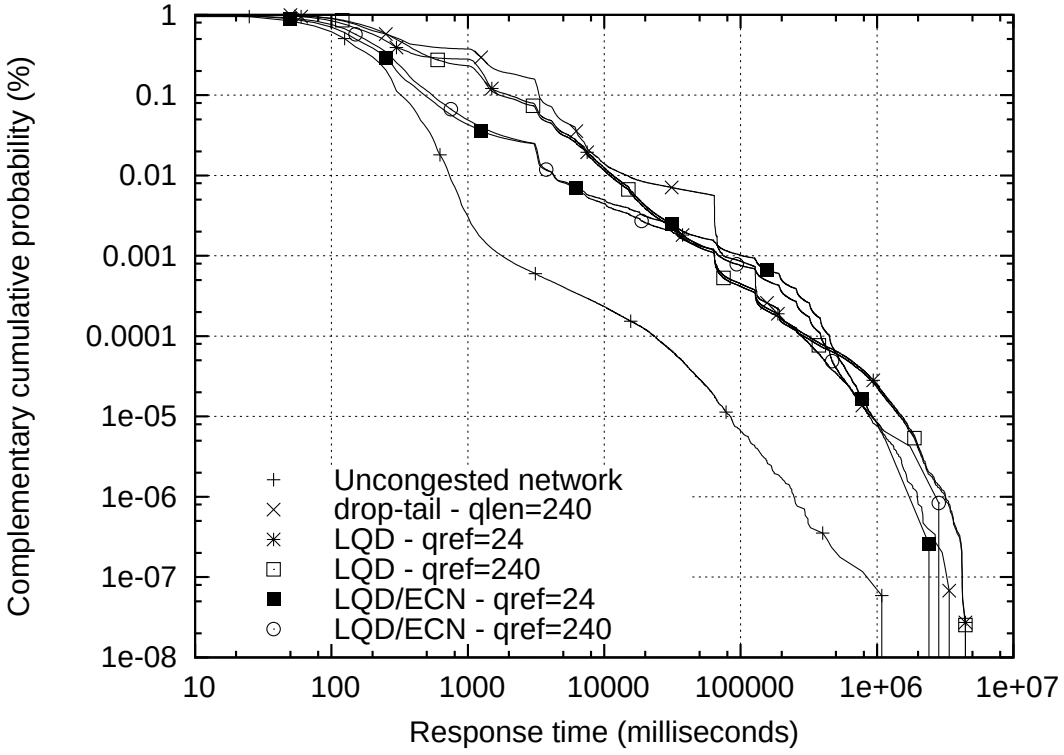

Figure 4.68: LQD/ECN performance at 105% load (CCDF)

At 80% offered load, LQD obtained the same performance when it was used with and without ECN. With or without ECN, LQD gave similar performance with both queue reference of 24 and 240 packets. The performance of LQD at this load was undistinguishable from that of the uncongested network and of drop-tail with a queue length of 240 packets.

At 90% offered load, LQD obtained a small performance improvement with ECN over packet dropping when LQD was operated with a queue reference of 24 packets. When LQD was used with a queue reference of 240 packets, it delivered essentially the same performance with and without ECN. With or without ECN, the performance for LQD at this load was clearly better than that of drop-tail and came relatively close to the performance of the uncongested network.

As the offered load increased to 98% and 105%, ECN help LQD obtain significant performance improvement over packet dropping. While LQD without ECN outperformed drop-tail at these loads, LQD gave performance that was significantly lower than that of the uncongested network. However, when LQD was used with ECN, the performance for LQD came reasonbly close to that of the uncongested network at these extreme loads.

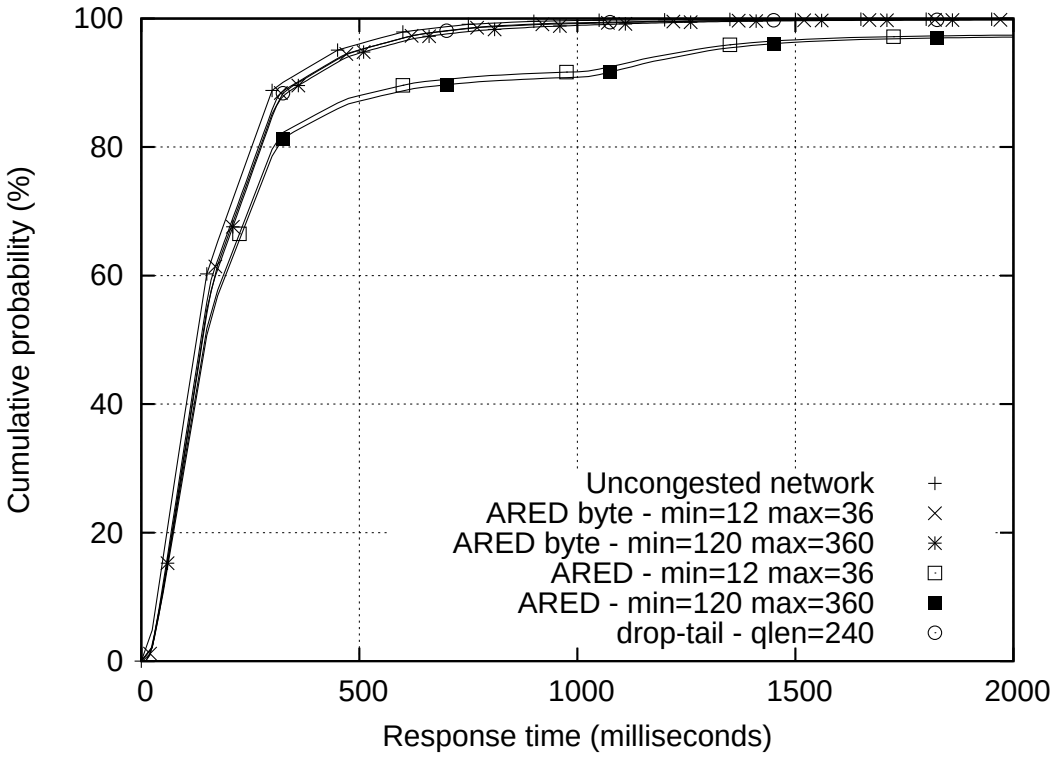

Figure 4.69: Performance of ARED byte mode at 80% load

4.6 The Effects of Byte Mode

Details of the ARED algorithm suggest that a possible reason for the consistent poor performance of ARED shown in section 4.2 is the "packet mode" that ARED uses to measure the router's queue. In contrast to ARED, PI and REM use the "byte mode" and measure the router's queue in bytes. The "byte mode" enables fine grain in measurements of the router's queue. Furthermore, the side effect of the "byte mode" is that the drop probability for an arriving packet is scaled by the size of that packet.

The side effect of the "byte mode" results in a lower drop probability for control packets such as SYNs and pure ACKs because of their small packet sizes and effectively gives a mechanism for protecting these packets. Since these control packets potentially have an important impact on application performance, "byte mode" could give significant improvement for application performance. While ARED also has a "byte mode", this mode was not recommended by the inventors of ARED.

Experimental results were obtained for ARED in "byte mode" and compared with results for ARED in "packet mode" to explore the potential effects of "byte mode".

Figures 4.69, 4.70, 4.71, and 4.72 show the performance of ARED in "packet mode" and "byte mode" at 80%, 90%, 98%, and 105% loads. The performance of ARED in both modes were obtained with a target queue reference of 24 and 240 packets by setting th_{min} and th_{max}

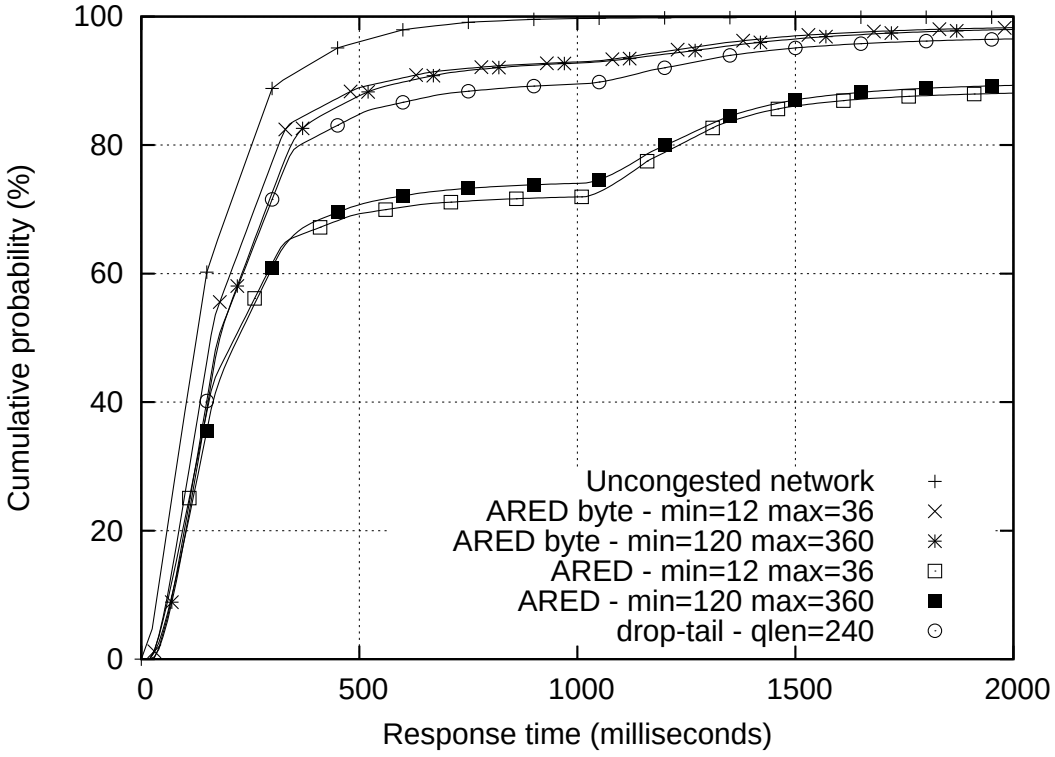

Figure 4.70: Performance of ARED byte mode at 90% load

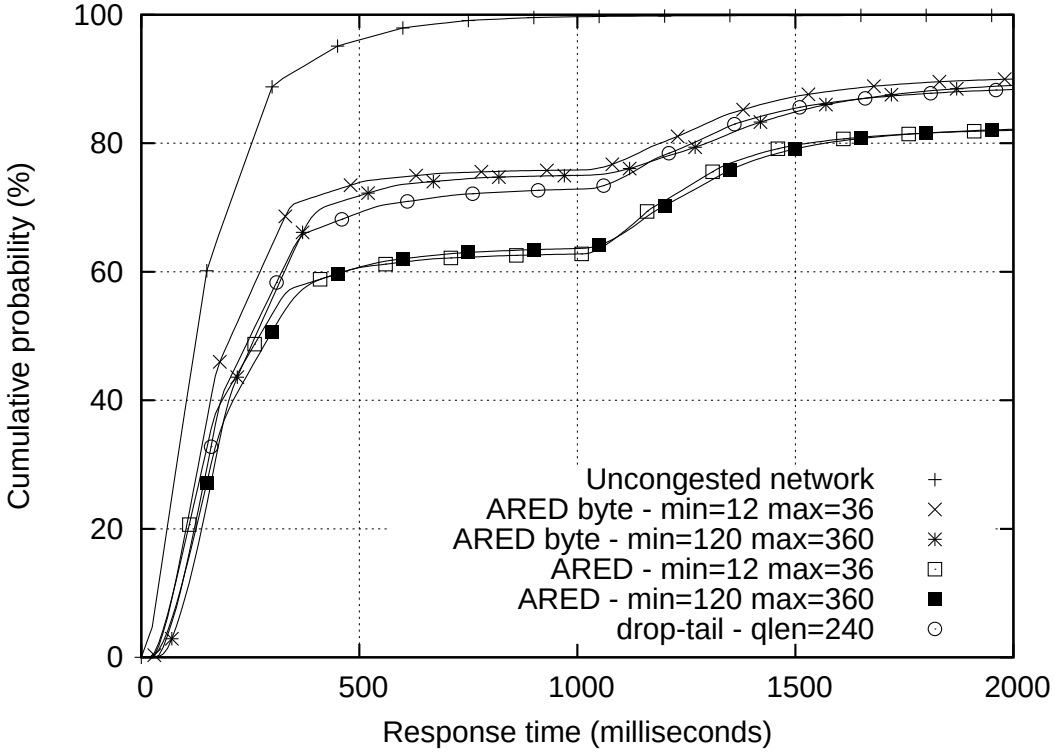

Figure 4.71: Performance of ARED byte mode at 98% load

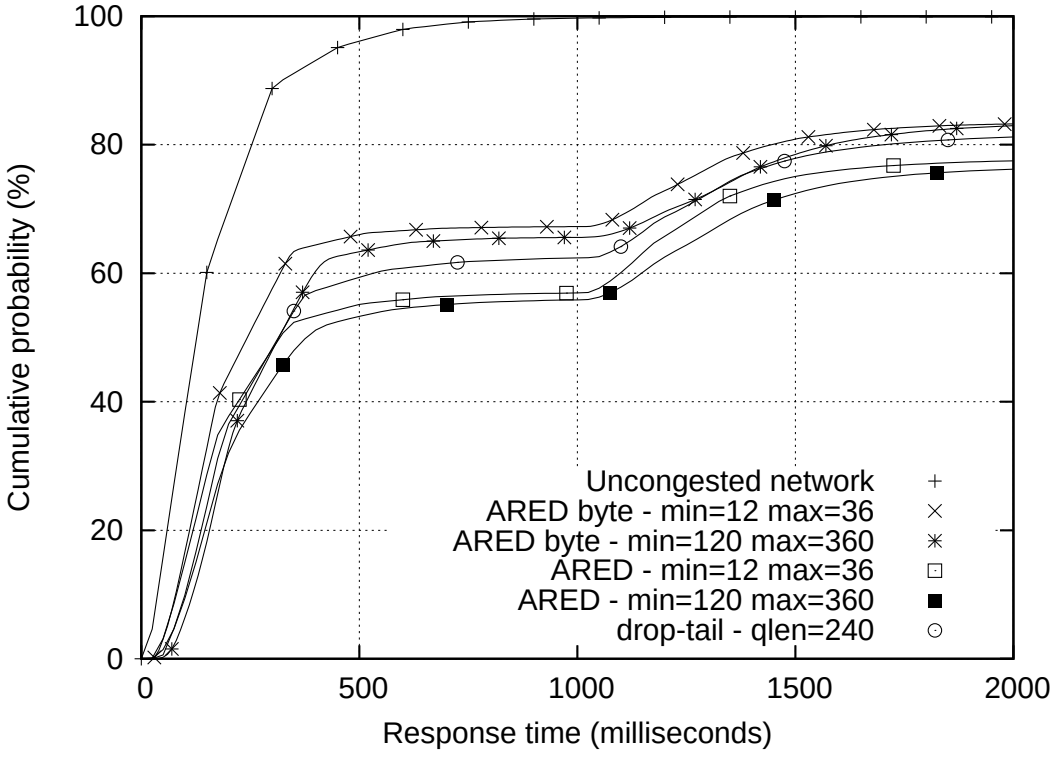

Figure 4.72: Performance of ARED byte mode at 105% load

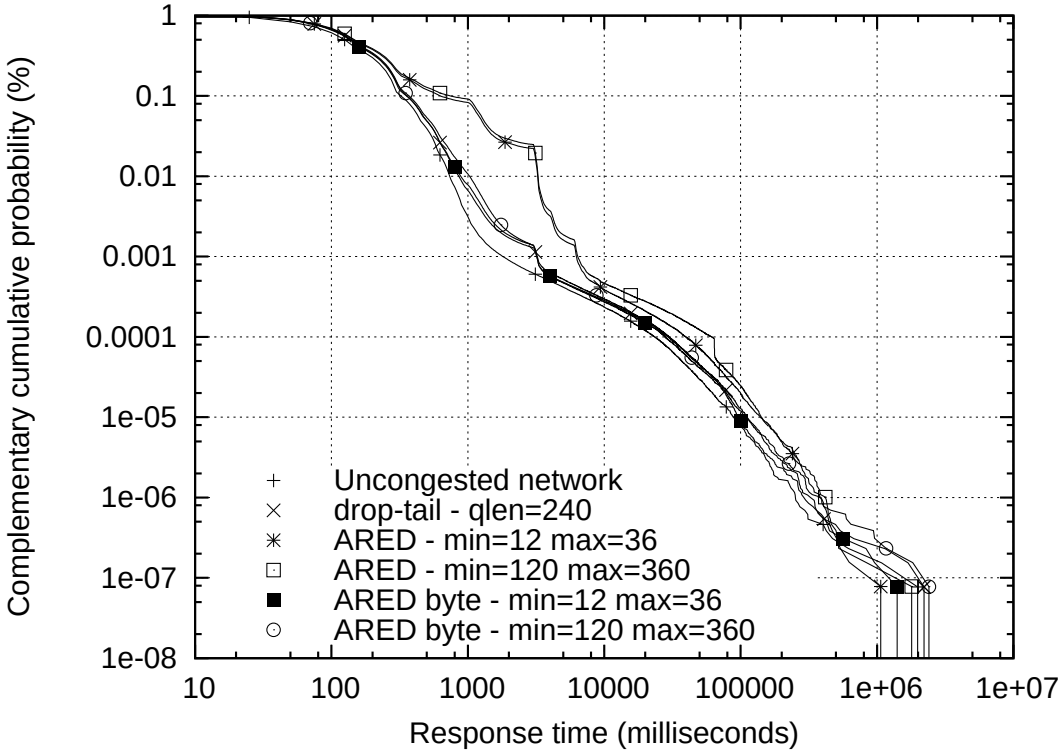

Figure 4.73: Performance of ARED byte mode at 80% load (CCDF)

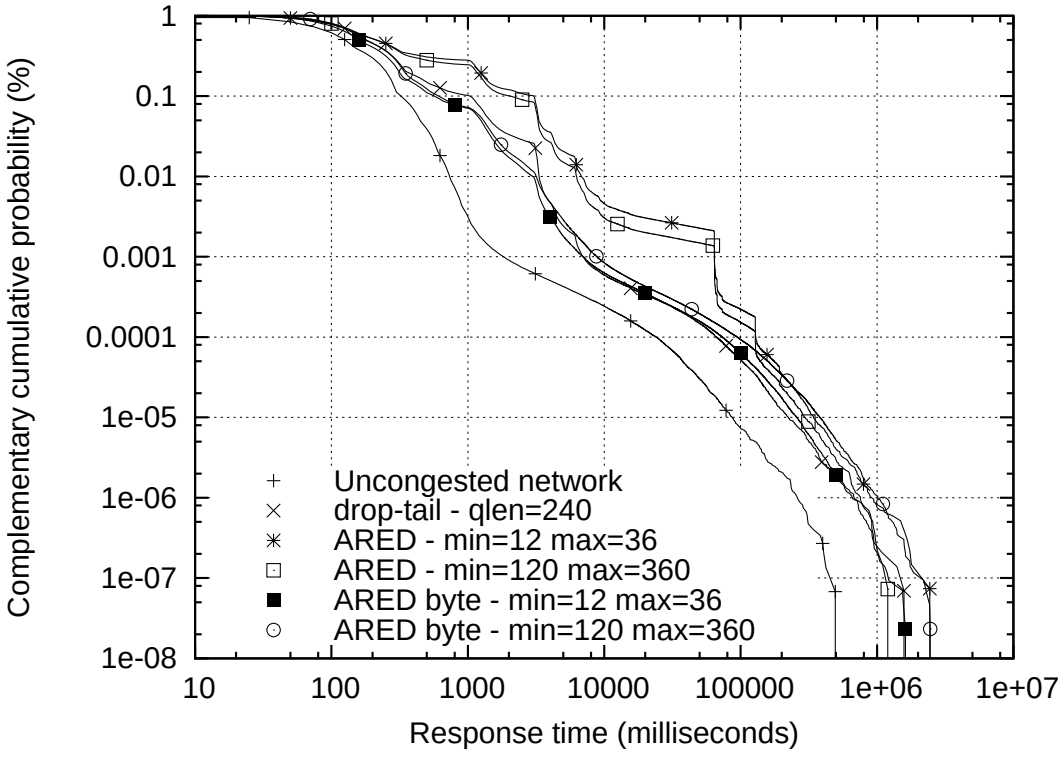

Figure 4.74: Performance of ARED byte mode at 90% load (CCDF)

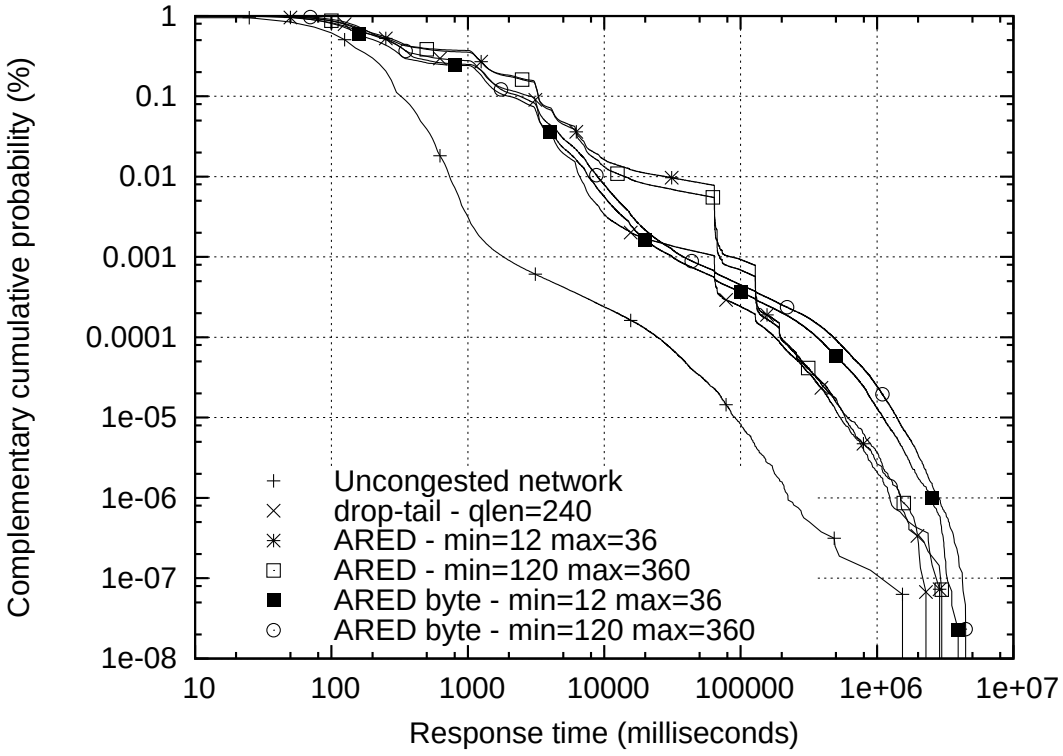

Figure 4.75: Performance of ARED byte mode at 98% load (CCDF)

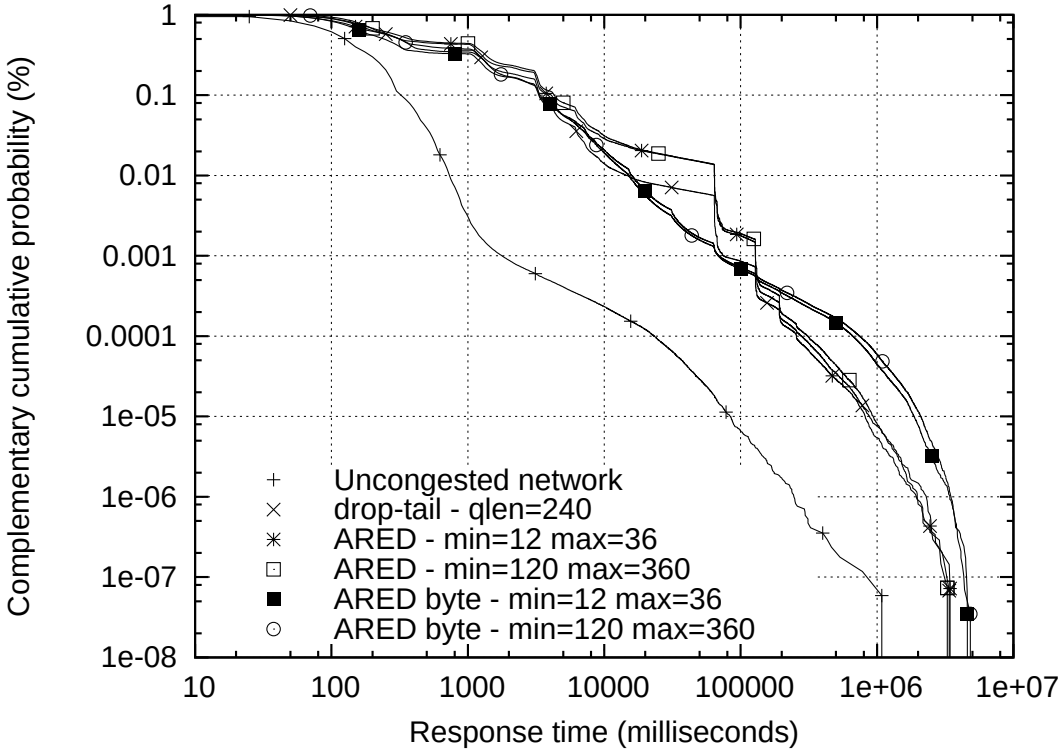

Figure 4.76: Performance of ARED byte mode at 105% load (CCDF)

to appropriate values. Figures 4.69, 4.70, 4.71, and 4.72 also compare the performance of ARED in both modes with the performance of drop-tail.

At 80% load, ARED in "packet mode" gave poorer performance than drop-tail. However, ARED in "byte mode" significantly outperformed ARED in "packet mode", gave comparable performance with drop-tail and came close to the performance of uncongested network.

At 90% load, ARED in "packet mode" continued to deliver poorer performance than drop-tail. However, the "byte mode" helped ARED obtain significant performance improvement and outperform drop-tail. When used in "byte mode", ARED with ($th_{min} = 12$ packets, $th_{max} = 36$ packets) gave slightly better performance than with ($th_{min} = 120$ packets, $th_{max} = 360$ packets) for about 90% of flows. For the rest 10% of flows, ARED in "byte mode" delivered the same performance with both parameter settings.

At 98% and 105% offered loads, ARED performance was also improved significantly with "byte mode". While the performance of ARED in "packet mode" was poorer than that of drop-tail, ARED in "byte mode" outperformed drop-tail at these loads. When ARED was used in "byte mode", it delivered slightly better performance with ($th_{min} = 12$ packets, $th_{max} = 36$ packets) than with ($th_{min} = 120$ packets, $th_{max} = 360$ packets) at 98% and 105% loads.

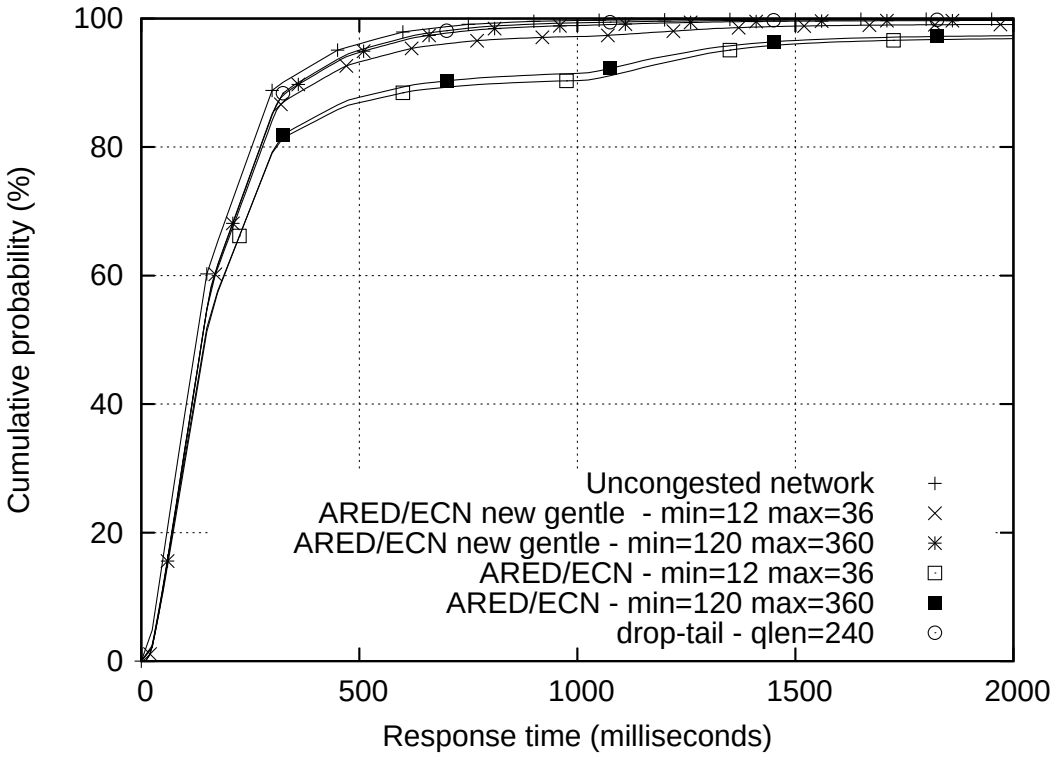

Figure 4.77: Performance of ARED/ECN new gentle mode at 80% load

4.7 The Effects of Dropping Packets in ECN Mode

As shown in section 4.5, the addition of ECN did not help ARED improve its performance at all. Details of the ARED algorithm in ECN mode suggest that a possible reason for the unimproved performance of ARED with ECN is a recommendation in the IETF's RFC 3168 that ARED chooses to follow. This recommendation states that an ECN-capable router should drop packets when its average queue size exceeds a certain threshold [RFB01]. Following this recommendation, ARED drops packets probabilistically when the average queue is between the thresholds max_{th} and $2max_{th}$ even when ARED operates in ECN mode and the dropped packets could have been marked.

I proposed a modification for ARED call "new gentle" that marks ECN-capable packets (instead of dropping them as in the original ARED algorithm) when the average queue size is between the thresholds max_{th} and $2max_{th}$. The name "new gentle" stems from the fact that the operational region for ARED between max_{th} and $2max_{th}$ is called the gentle mode.

Experimental results were obtained with ARED "new gentle" (henceforth called ARED/ECN "new gentle") and compared with the results of the original ARED in ECN (henceforth called ARED/ECN) mode to explore the effects of dropping packets in ECN mode.

Figures 4.77, 4.78, 4.79, and 4.80 show the results of ARED the original and new "gentle mode" when ECN is enabled at the routers and end systems. Experimental results for

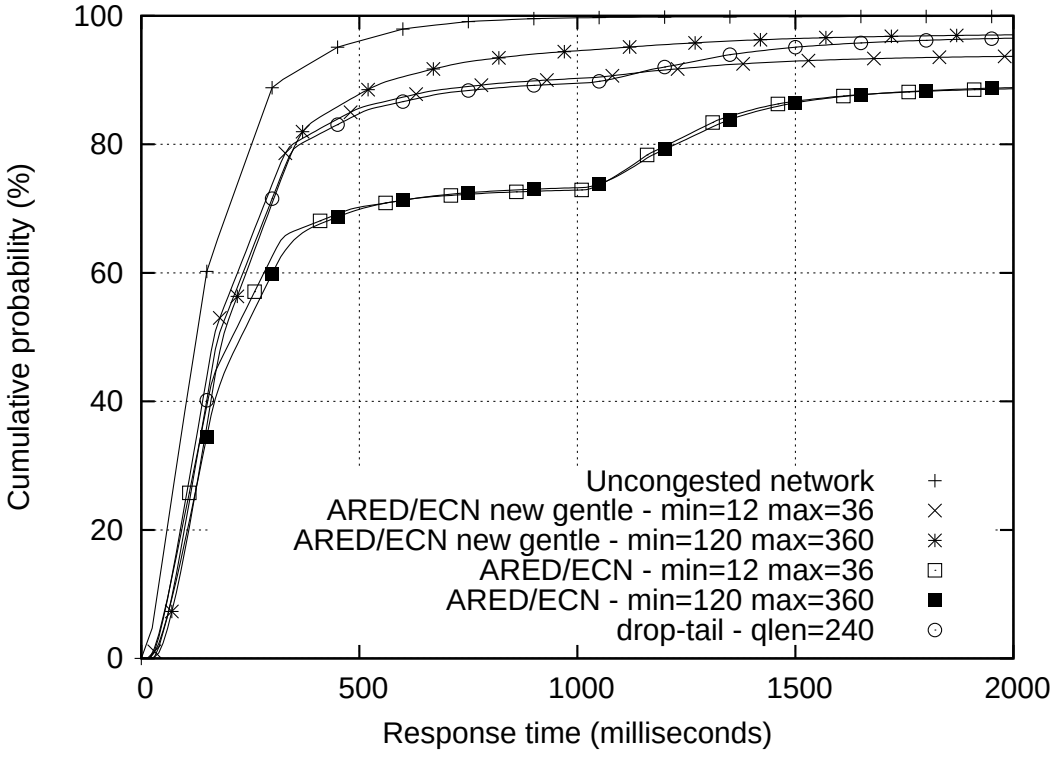

Figure 4.78: Performance of ARED/ECN new gentle mode at 90% load

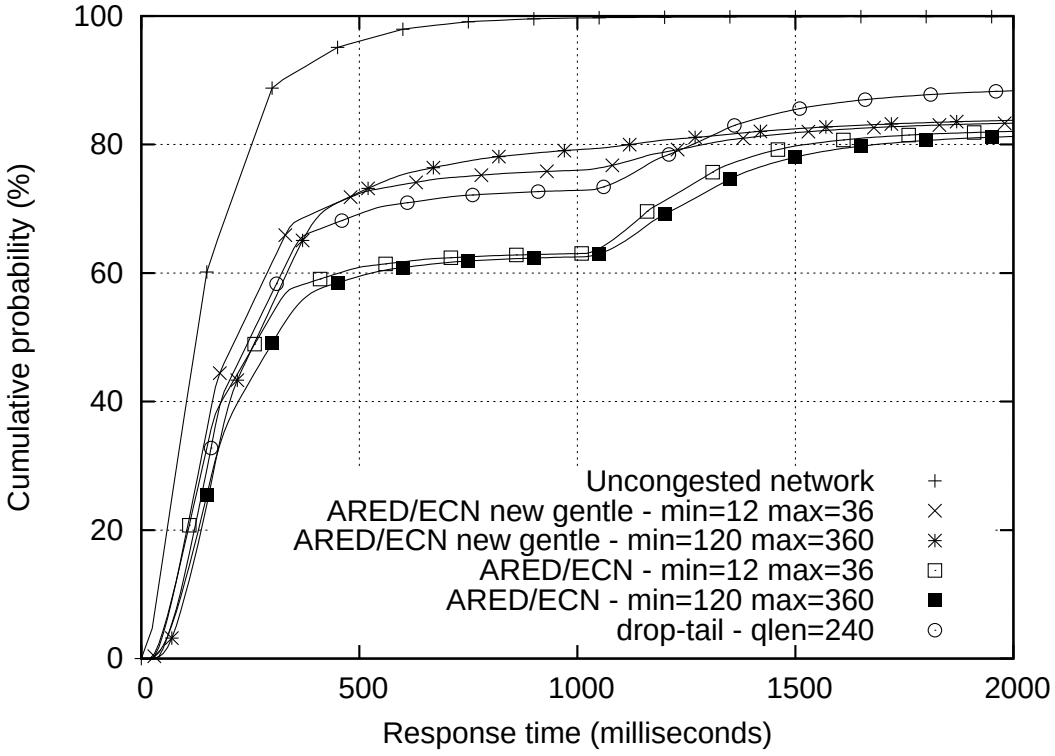

Figure 4.79: Performance of ARED/ECN new gentle mode at 98% load

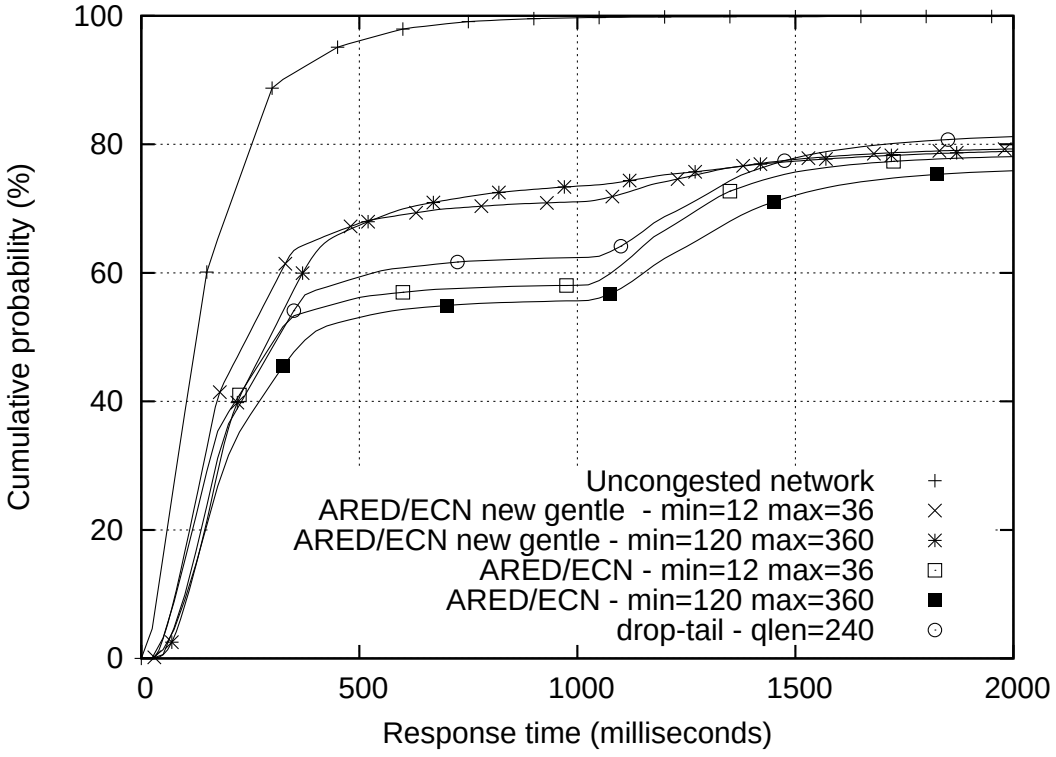

Figure 4.80: Performance of ARED/ECN new gentle mode at 105% load

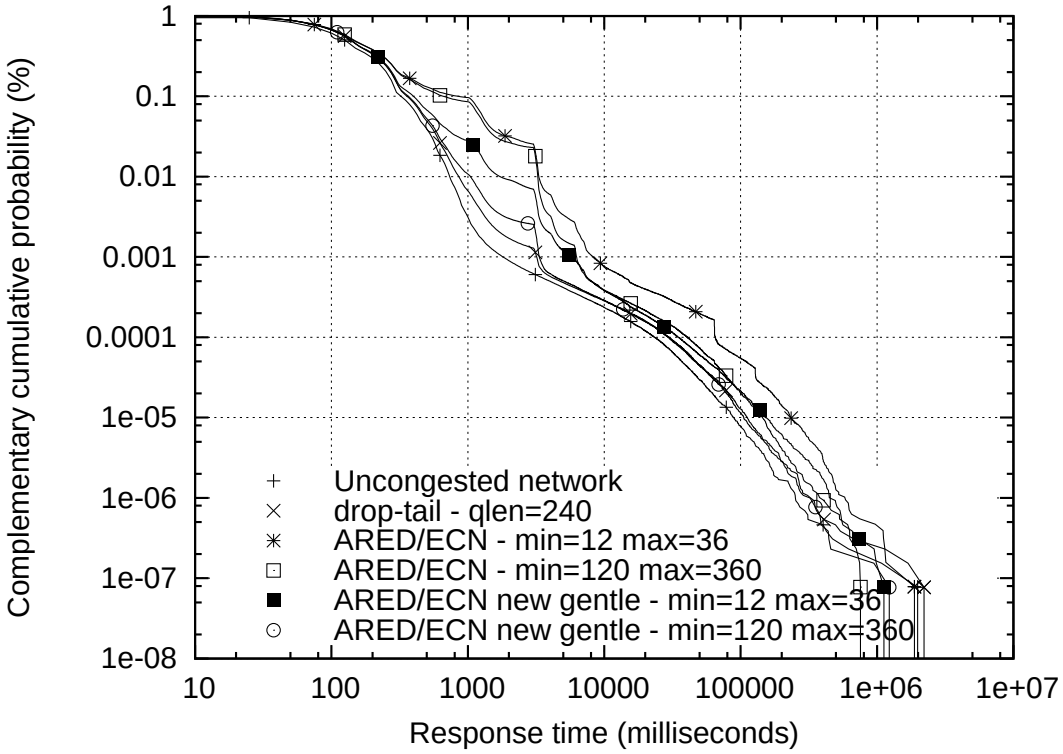

Figure 4.81: Performance of ARED/ECN new gentle mode at 80% load (CCDF)

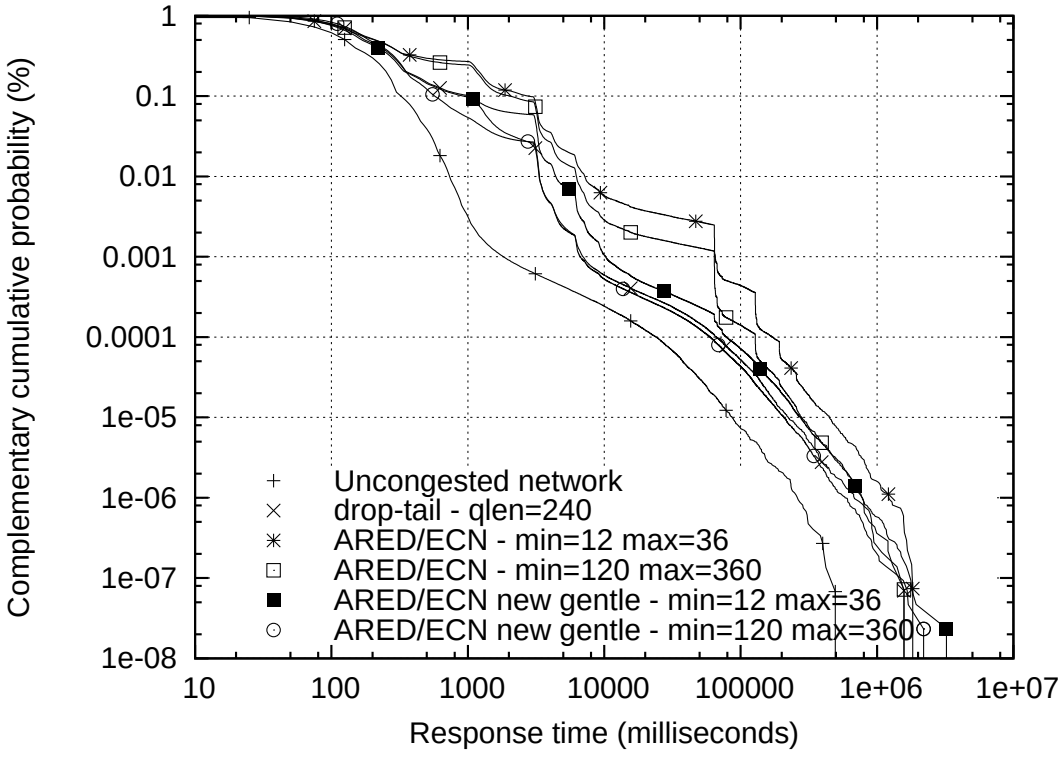

Figure 4.82: Performance of ARED/ECN new gentle mode at 90% load (CCDF)

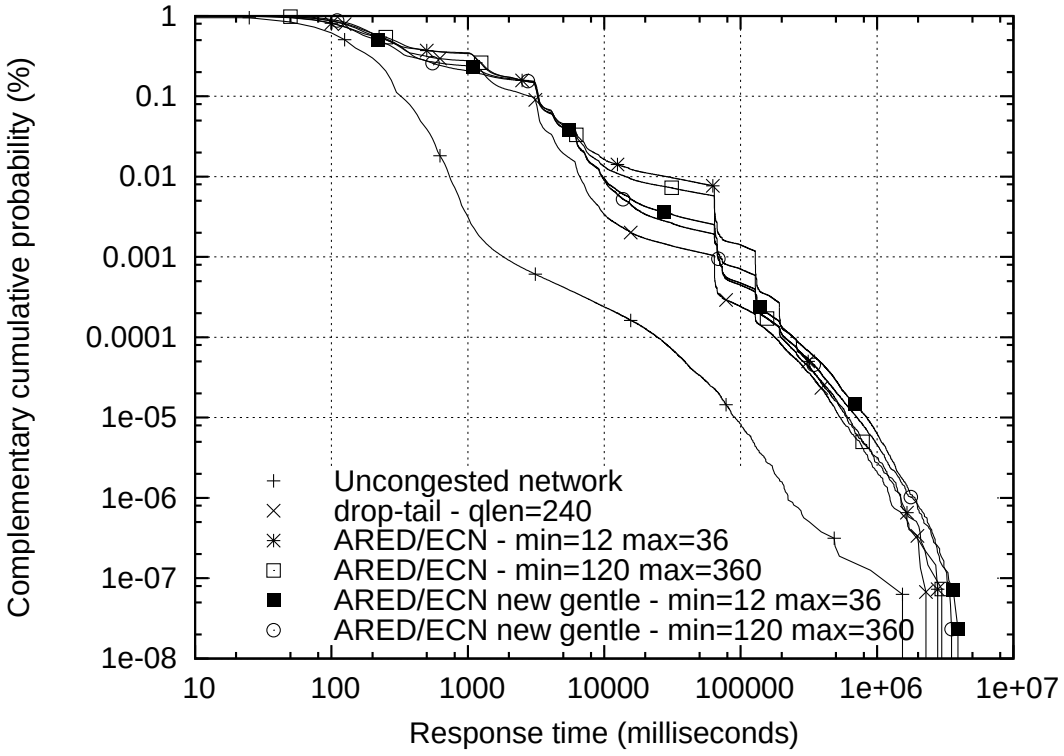

Figure 4.83: Performance of ARED/ECN new gentle mode at 98% load (CCDF)

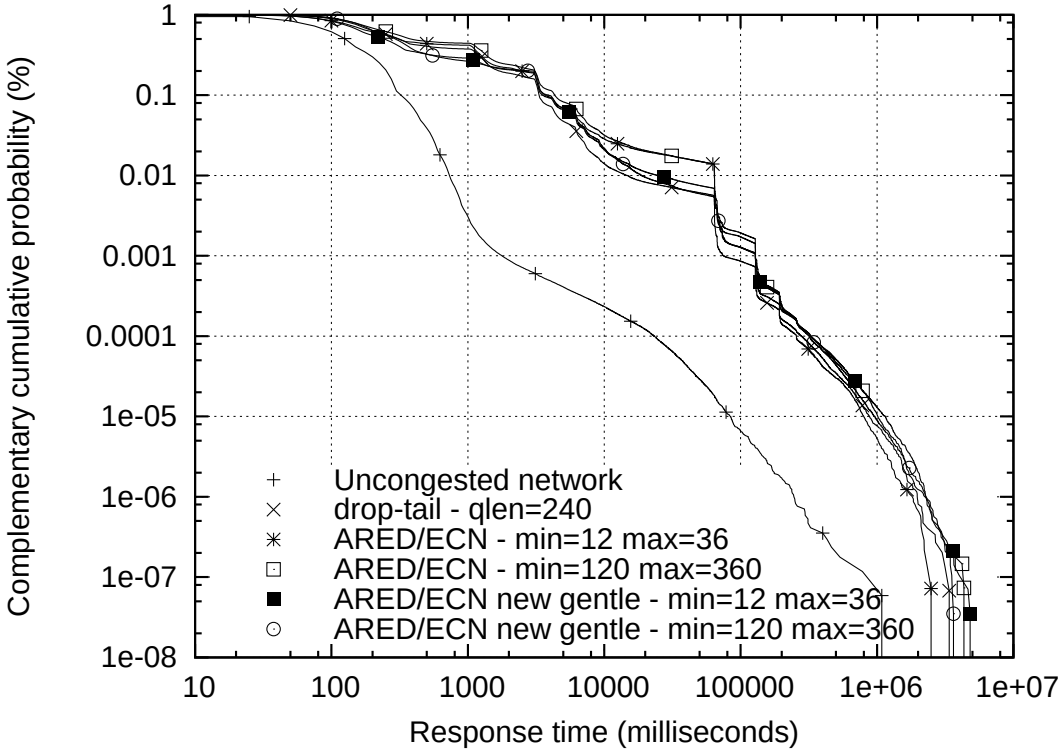

Figure 4.84: Performance of ARED/ECN new gentle mode at 105% load (CCDF)

ARED and ARED "new gentle" were obtained with a target queue reference of 24 and 240 packets.

At 80% load, the original ARED algorithm in ECN mode provided poorer performance than drop-tail. However, ARED "new gentle" obtained significant performance improvement over the original ARED algorithm and gave better performance than drop-tail.

At 90% load, the original ARED algorithm in ECN mode again delivered poorer performance than drop-tail for all flows. The ARED "new gentle" algorithm benefited from the addition of ECN and significantly outperformed the original ARED algorithm. When used with ($th_{min} = 12$ packets, $th_{max} = 36$ packets), ARED "new gentle" delivered comparable response times as drop-tail for 90% of flows but provided slightly poorer performance than drop-tail for the rest 10% of flows. However, ARED "new gentle" with ($th_{min} = 120$ packets, $th_{max} = 360$ packets) gave better response times than drop-tail for all flows.

At 98% and 105% loads, the original ARED algorithm in ECN mode continued to gave poor performance that was not competitive with drop-tail. ARED "new gentle" with ($th_{min} = 12$ packets, $th_{max} = 36$ packets) obtained better performance for approximately 70% of flows but worse performance for the rest 30% of flows than ARED "new gentle" with ($th_{min} = 120$ packets, $th_{max} = 360$ packets). Further, ARED "new gentle" with both parameter settings obtained performance that was better than that of drop-tail for

approximately 80% of flows. However, the rest 20% of flows experienced better response times under drop-tail.

4.8 The Effects of Differential Treatment of Flows

Experimental results in previous sections demonstrated the potential benefits of AQM in improving both network and application performance. However, this positive result was dampened by the fact that AQM algorithms evaluated in previous sections required the signaling protocol ECN to deliver good performance at high loads. This presents a serious hindrance for AQM because ECN is not currently deployed or enabled by most end systems.

In a study of TCP behavior in 2001, Padhye and Floyd found that less than 10% of the 24,030 web servers tested had ECN enabled, of which less than 1% had a compliant implementation or configuration of ECN [PF01]. More recent ECN testing in 2004 showed that only 2.1% of web servers on the Internet had correctly deployed ECN [MAF05]. This clearly points to obvious difficulties in deploying and properly using ECN on the end-systems.

Because of the lack of ECN deployment, some AQM algorithms proposed in recent years attempted to improve network and application performance without ECN by taking advantage of traffic characteristics [GM01, FKSS01b, PBPS03, LAJS04]. These AQM algorithms distinguish between flows when they decide whether or not to mark or drop arriving packets (in contrast to AQM algorithms evaluated in previous sections that treat all flows identically).

This section presents experimental results that were obtained to evaluate the effects of differential treatment of flows by AQM algorithms. Experiments were performed for three algorithms: Approximate Fairness through Differential Dropping (AFD) [PBPS03], RED with In and Out with Preferential treatment to Short flows (RIO-PS) [GM01], and Differential Congestion Notification (DCN) [LAJS04]. Attempts to perform experiments for the Stochastic Fair BLUE algorithm [FKSS01b] failed because of this algorithm's high demand for CPU resources on the routers in the laboratory network.

4.8.1 Results for AFD

Figure 4.85, 4.86, 4.87, and 4.88 present the experimental results for AFD with and without ECN at 80%, 90%, 98%, and 105% offered loads. These results were obtained with a queue reference of 24 and 240 packets.

At 80% load, AFD without ECN and with a queue reference of 24 packets obtained slightly worse performance than drop-tail. However, AFD without ECN and with a queue reference of 240 packets delivered good response times that were comparable with the performance of drop-tail. Further, when AFD was used with ECN, AFD with both queue

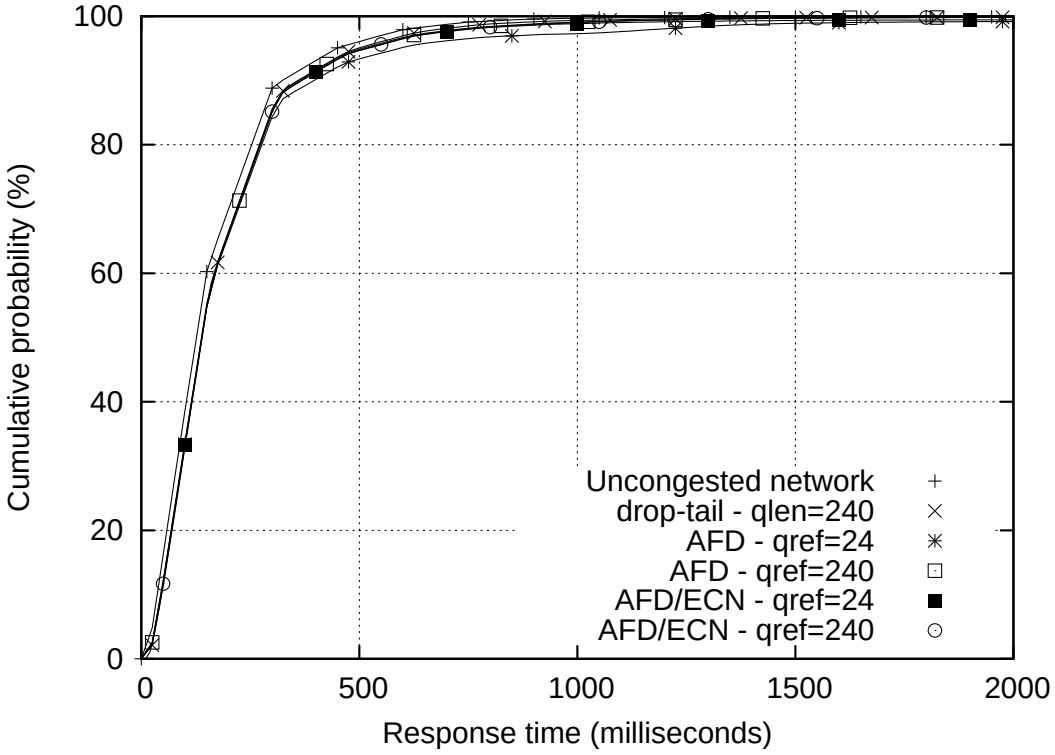

Figure 4.85: Performance of AFD with and without ECN at 80% load

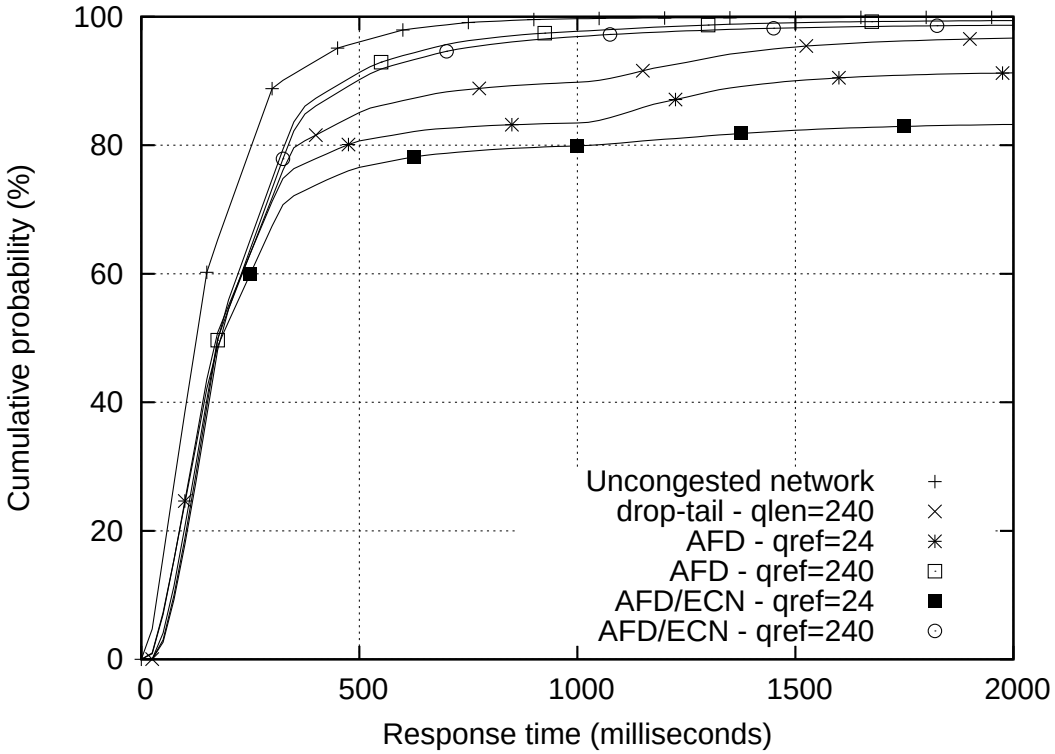

Figure 4.86: Performance of AFD with and without ECN at 90% load

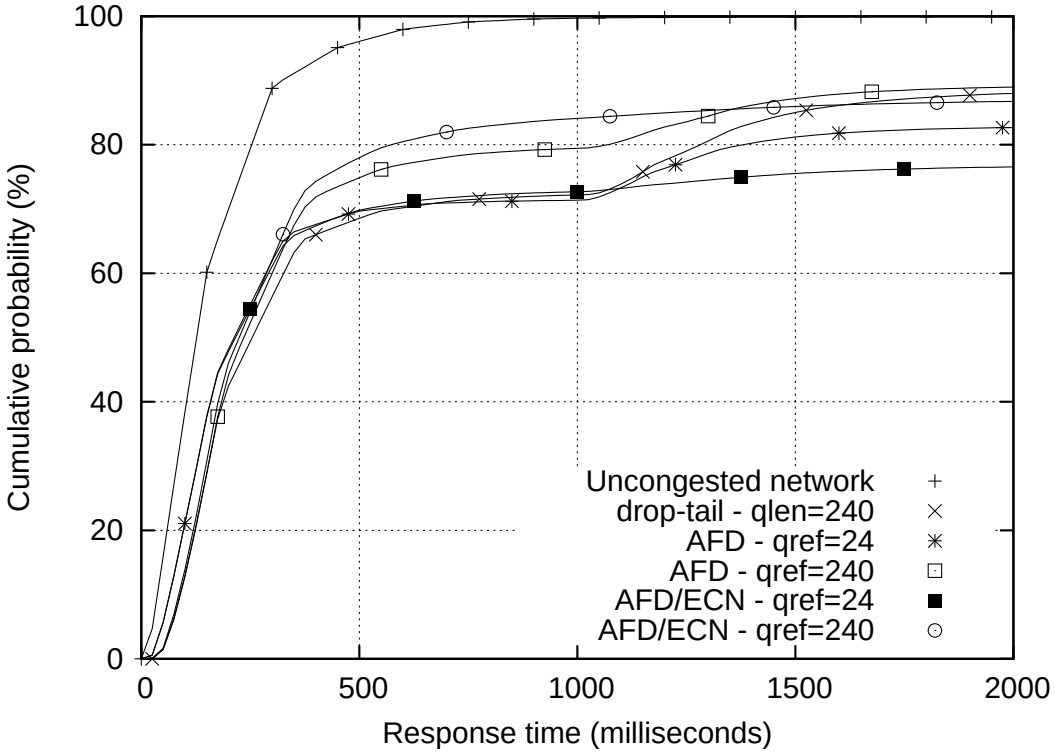

Figure 4.87: Performance of AFD with and without ECN at 98% load

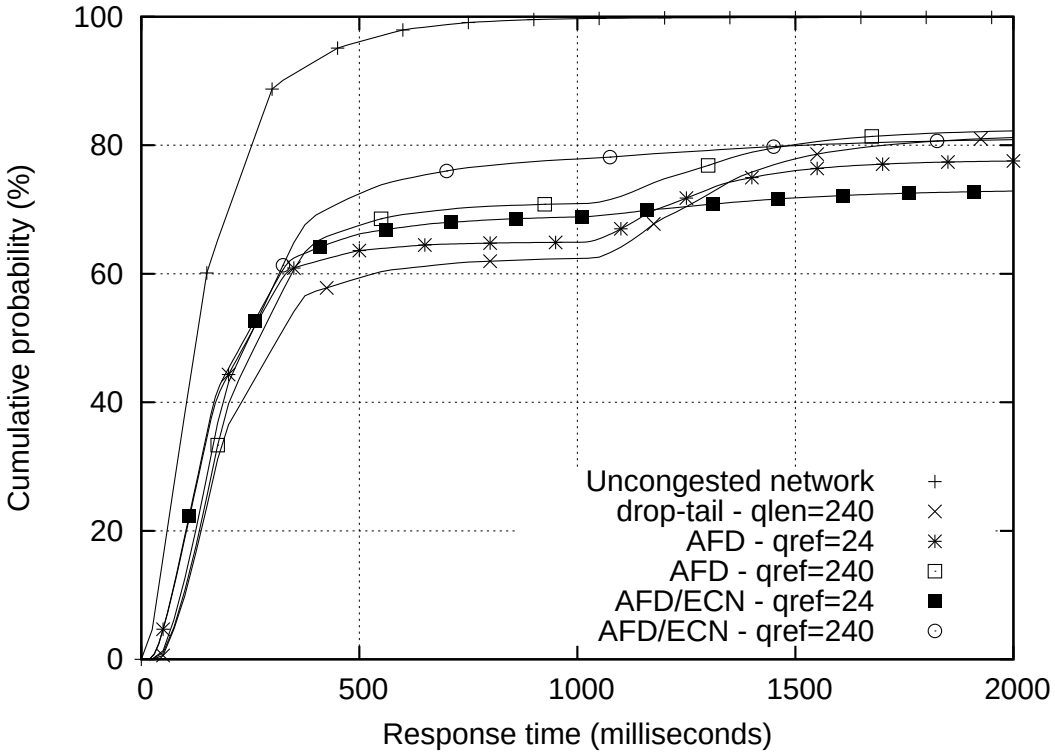

Figure 4.88: Performance of AFD with and without ECN at 105% load

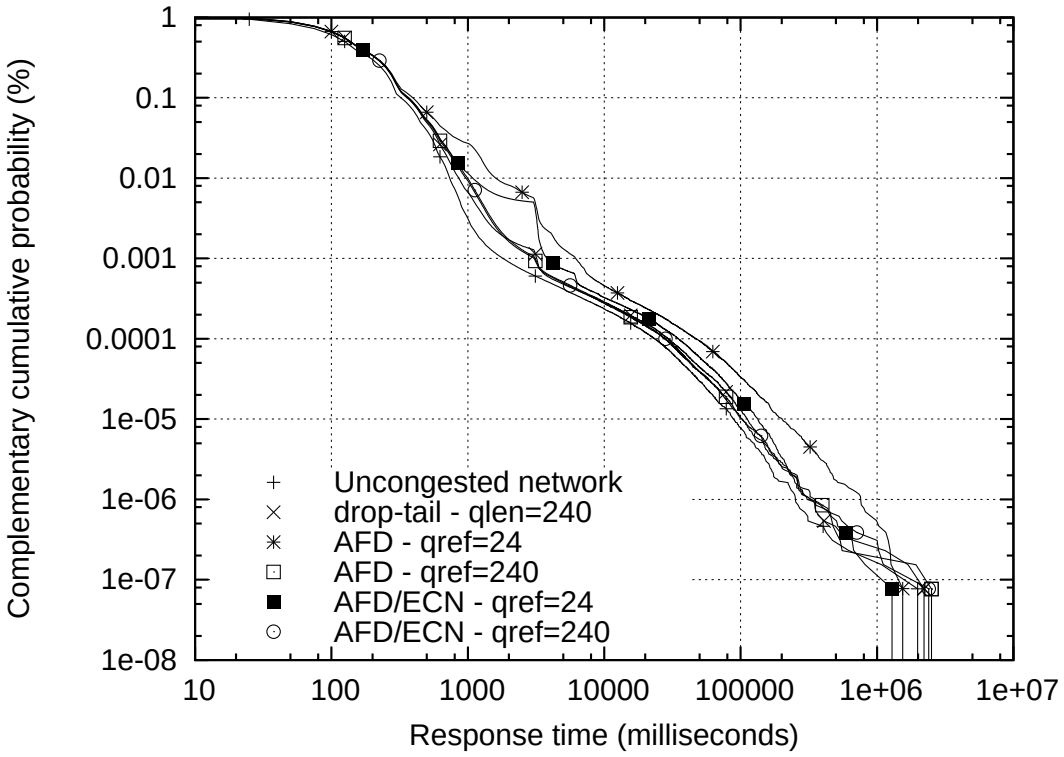

Figure 4.89: Performance of AFD with and without ECN at 80% load (CCDF)

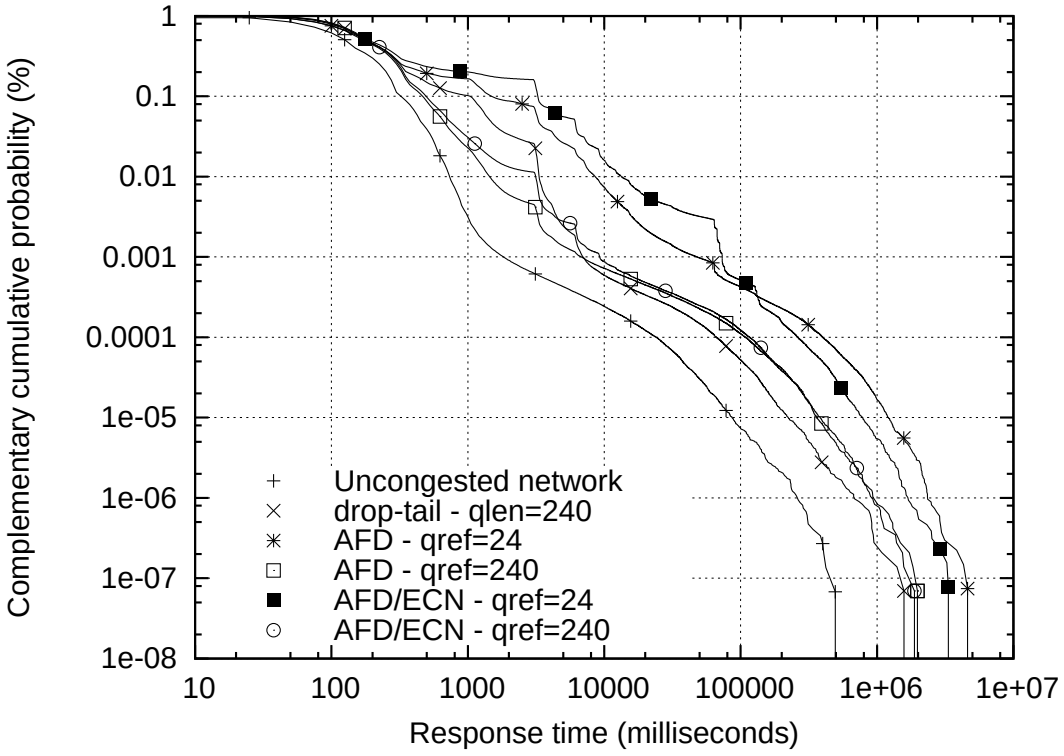

Figure 4.90: Performance of AFD with and without ECN at 90% load (CCDF)

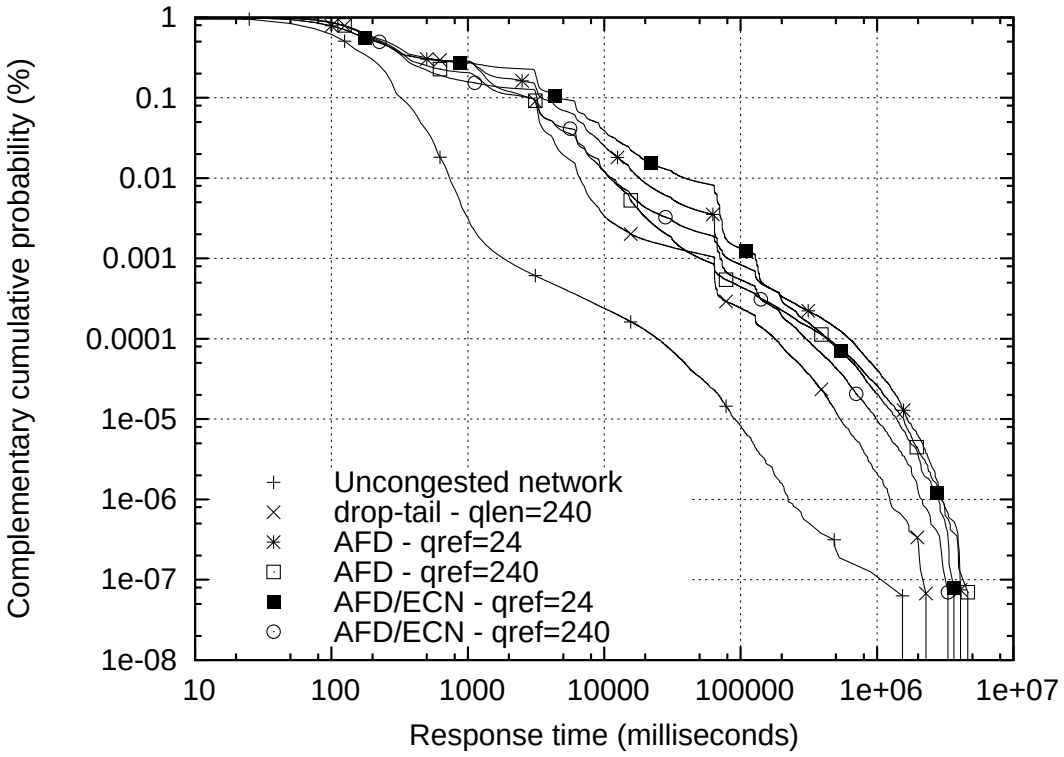

Figure 4.91: Performance of AFD with and without ECN at 98% load (CCDF)

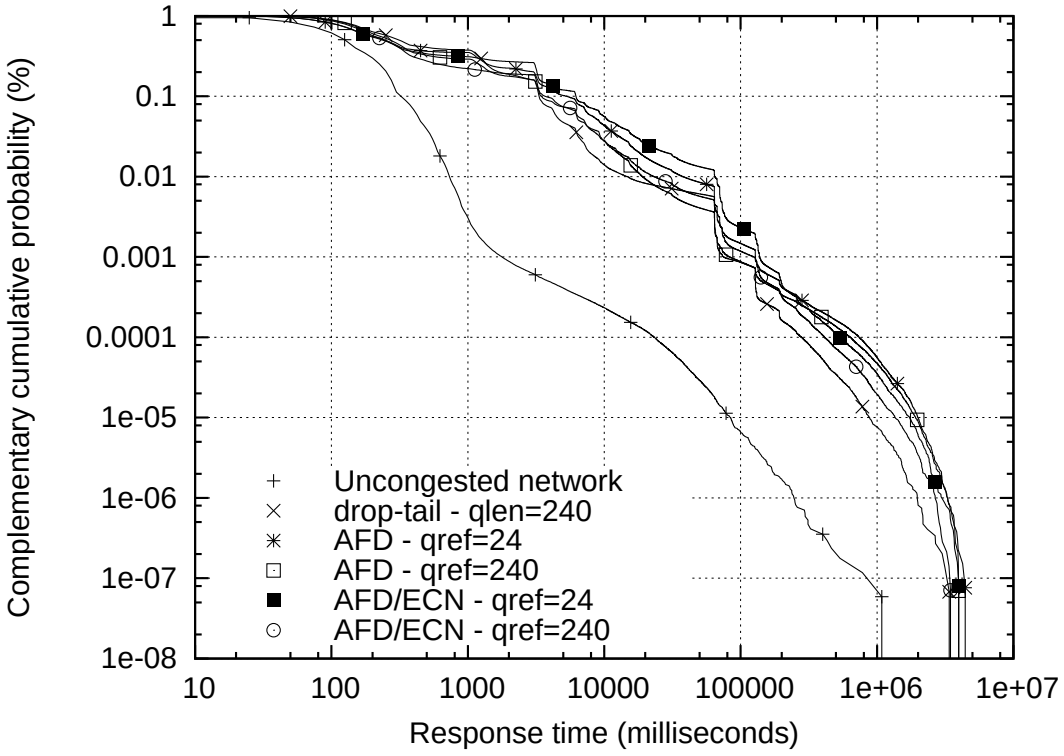

Figure 4.92: Performance of AFD with and without ECN at 105% load (CCDF)

reference of 24 and 240 packets also achieved as good performance as drop-tail and competitive to that of uncongested network.

When AFD operated with a queue reference of 24 packets at 90% offered load, AFD with and without ECN delivered poor performance that was worse than the performance of drop-tail However, when AFD operated with a queue reference of 240 packets, AFD with and without ECN gave good response times and outperformed drop-tail at this load.

At 98% load, AFD with a queue reference of 24 packets (with and without ECN) gave comparable performance with drop-tail for 70% of flows but delivered poorer performance than drop-tail for the rest 30% of flows. However, AFD with a queue reference of 240 packets (with and without ECN) achieved better performance than drop-tail.

At 105% load, AFD with a queue reference of 24 packets (with and without ECN) outperformed drop-tail for 70% of flows but gave worse performance than drop-tail for the rest 30% of flows. AFD with a queue reference of 240 packets (with and without ECN) continued to outperform drop-tail.

4.8.2 Results for RIO-PS

Figure 4.93, 4.94, 4.95, and 4.96 show the experimental results for RIO-PS at 80%, 90%, 98%, and 105% loads. These results were obtained with the default parameters for RIO-PS $th_{min} = 80$ packets, $th_{max} = 840$ packets, $max_p = 1/20$ for short-lived flows and $th_{min} = 80$ packets, $th_{max} = 1200$ packets, $max_p = 1/10$ for long-lived flows. These parameters were recommended so that packets from short-lived flows are marked or dropped with a lower probability than packets from long-lived flows.

At 80% offered load, RIO-PS with and without ECN obtained similar performance for all flows. RIO-PS gave comparable performance as drop-tail and approximated the performance of the uncongested network at this load.

At 90% offered load, RIO-PS delivered good performance both with and without ECN and outperformed drop-tail. Since RIO-PS already obtained good performance without ECN at this load, the addition of ECN only improved the performance for RIO-PS marginally.

As the offered load increased to 98% and 105%, RIO-PS continued to obtain acceptable performance even without ECN and outperformed drop-tail. The addition of ECN only improved the performance for RIO-PS slightly at these loads. However, the performance for RIO-PS both with and without ECN degraded noticeably at these high loads.

4.8.3 Results for DCN

Sections 4.8.1 and 4.8.2 demonstrated the difficult trade-off between performance and algorithm complexity in designing AQM algorithms for differential treatment of flows. While the RIO-PS algorithm could achieve good performance even without ECN, it maintained per-flow statistics for all flows and put a high demand for CPU and memory resources.

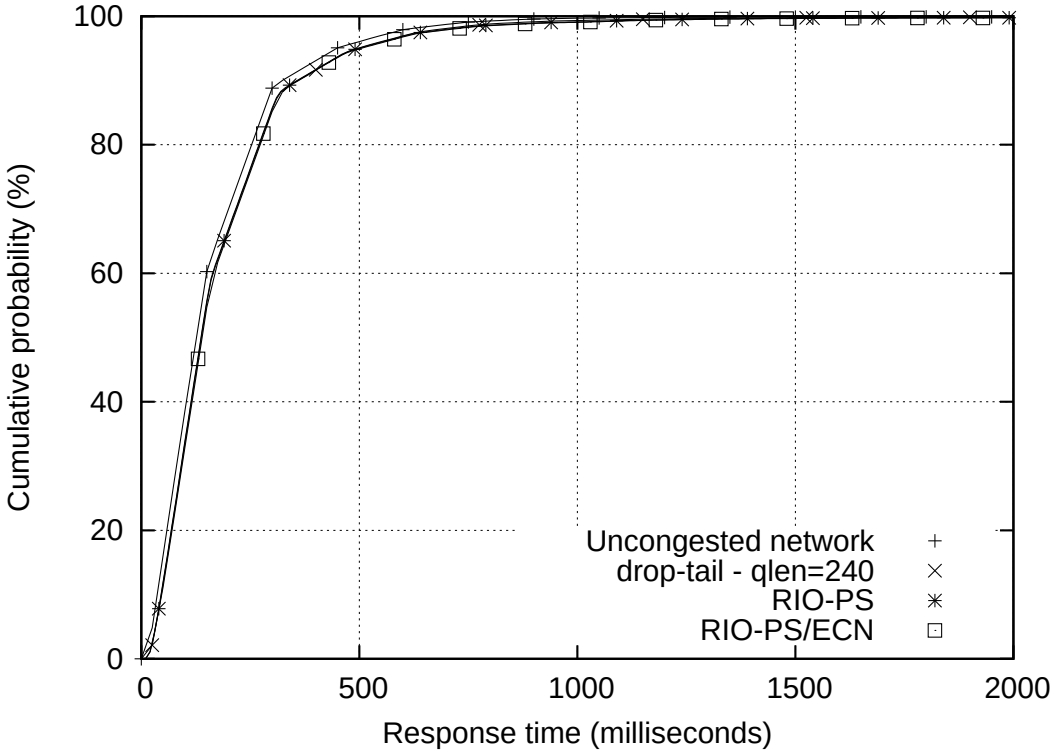

Figure 4.93: Performance of RIO-PS with and without ECN at 80% load

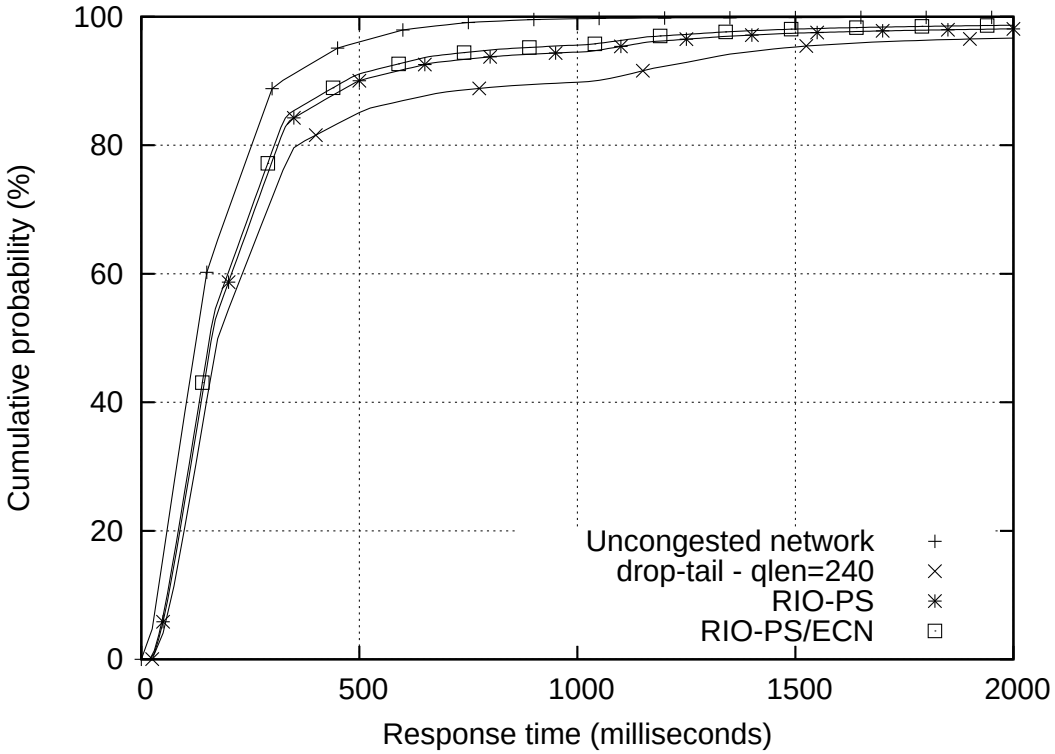

Figure 4.94: Performance of RIO-PS with and without ECN at 90% load

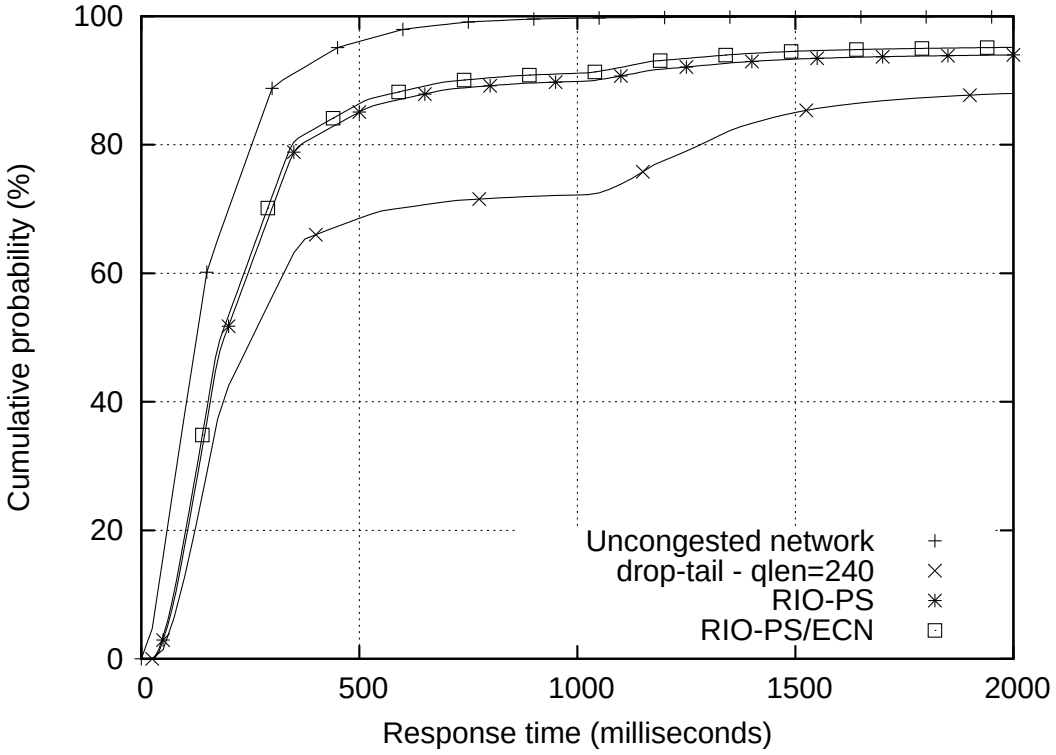

Figure 4.95: Performance of RIO-PS with and without ECN at 98% load

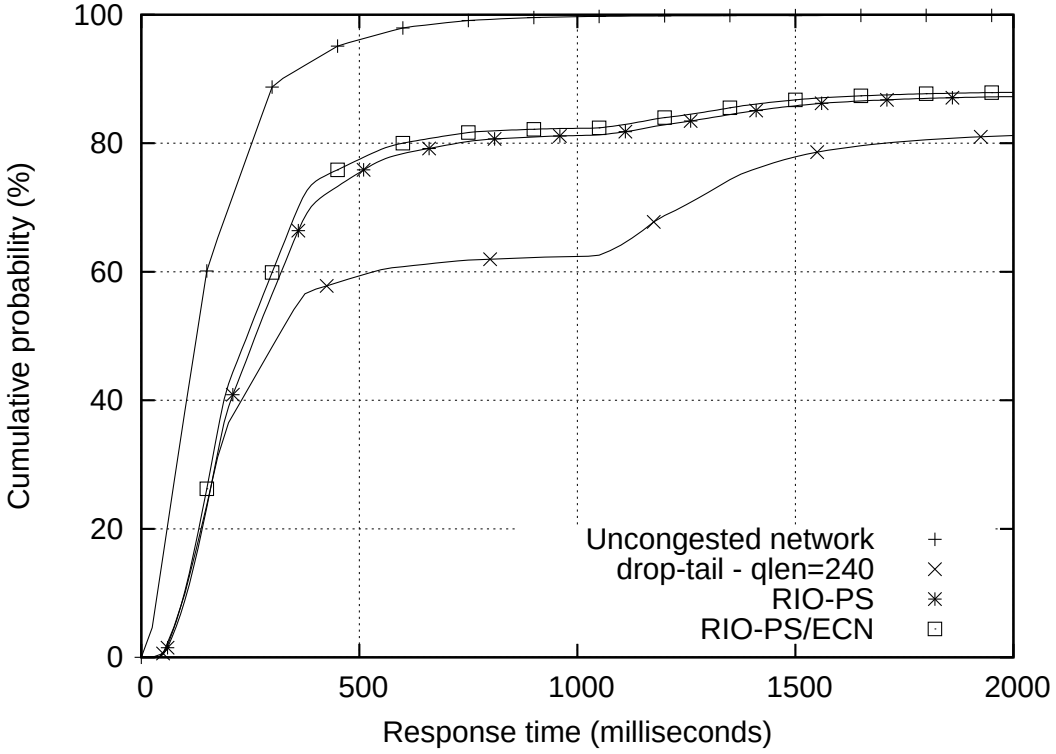

Figure 4.96: Performance of RIO-PS with and without ECN at 105% load

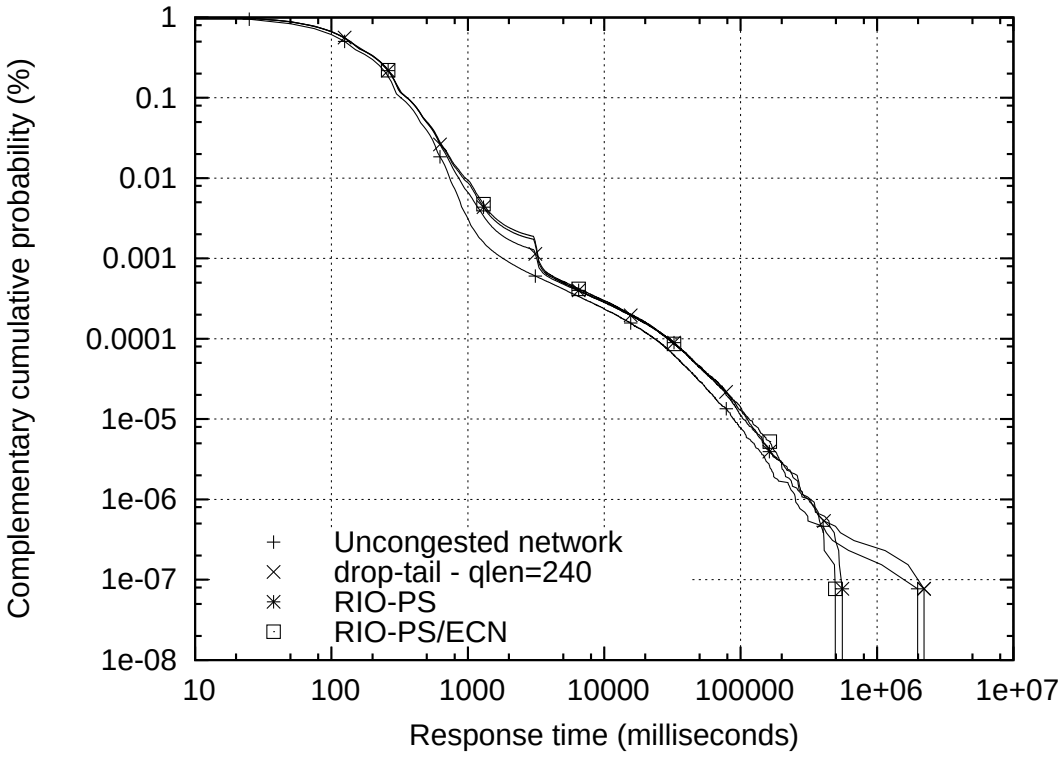

Figure 4.97: Performance of RIO-PS with and without ECN at 80% load (CCDF)

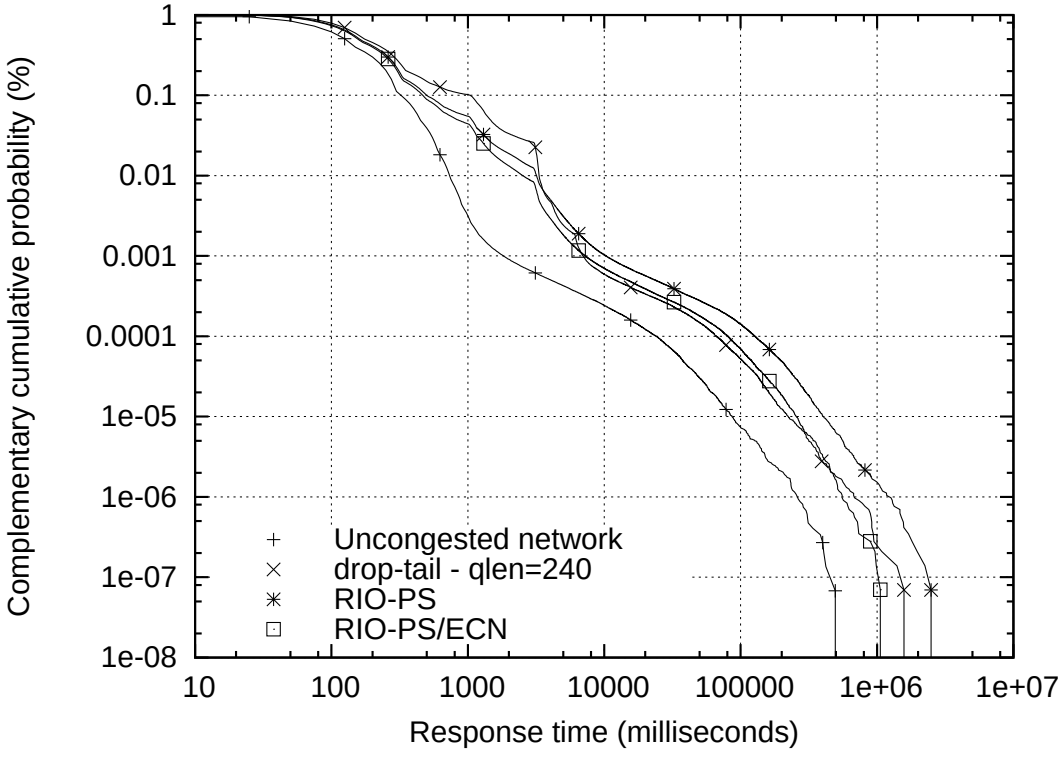

Figure 4.98: Performance of RIO-PS with and without ECN at 90% load (CCDF)

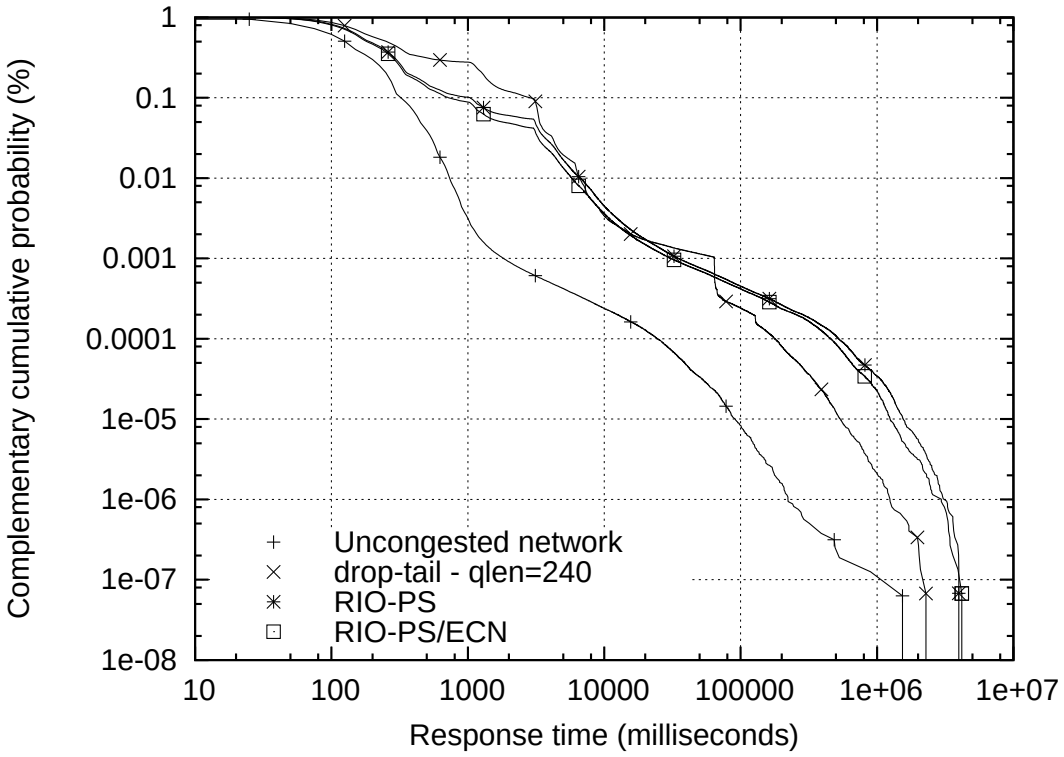

Figure 4.99: Performance of RIO-PS with and without ECN at 98% load (CCDF)

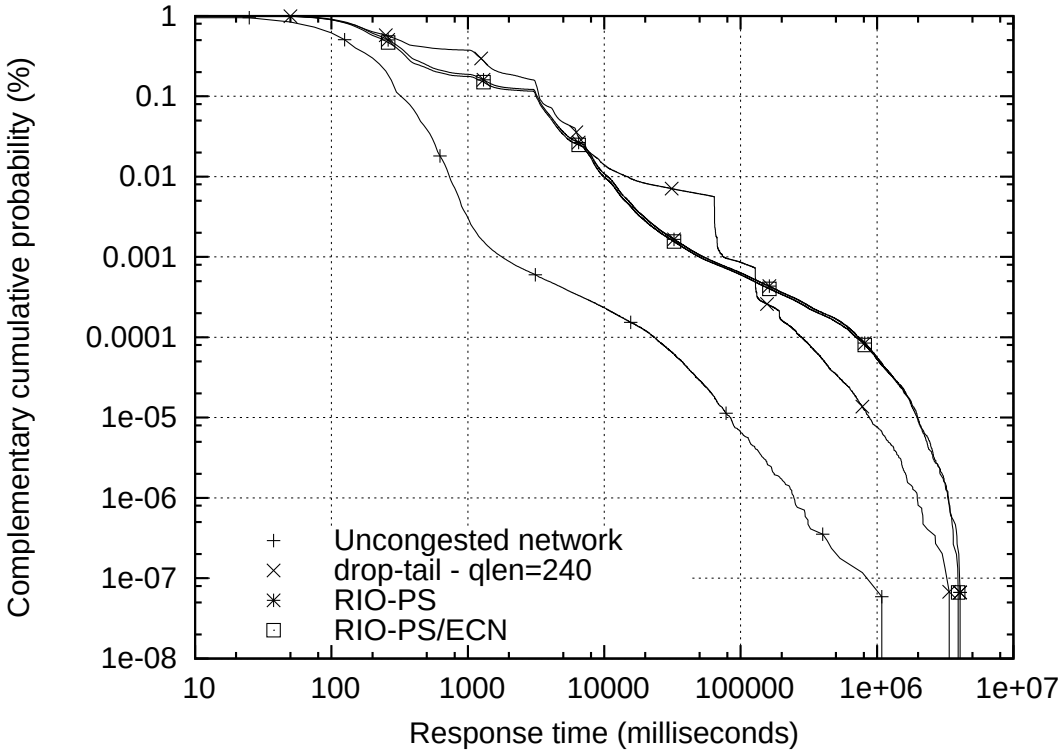

Figure 4.100: Performance of RIO-PS with and without ECN at 105% load (CCDF)

On the other hand, the AFD algorithm used a constant-size hash table and required less CPU and memory resources, but this algorithm did not obtain impressive performance improvement over drop-tail. This section investigates the Differential Congestion Notification (DCN) algorithm that attempts to achieve good performance with a relatively low algorithm complexity.

The DCN algorithm is based on identifying large (i.e., long-lived) and fast (i.e., high-bandwidth) flows and delivering congestion notification to these flows. The premise of DCN is the fact that while most flows on Internet links are small, a large fraction of bandwidth on an Internet link is consumed by a handful of large and high-bandwidth flows. For example, Zhang et al. found that small flows (100 KB or less) accounted for at least 84% of all flows, but carried less than 15% of all bytes [ZBPS02]. They also found that large flows accounted for a small fraction of the number of flows, but carried most of the bytes. Further, the flows that are large and fast (i.e., high-bandwidth, greater than 10 KB/sec) account for less than 10% of all flows, while carrying more than 80% of all the bytes.

After identifying large and high-bandwidth flows, DCN delivers congestion notification to these flows by either marking or dropping their packets. While marking is clearly preferred over dropping, experimental results demonstrated that DCN can obtain good performance at high offered loads without ECN (as will be shown later in this section and subsequent Chapters).

While the idea of DCN is simple, the challenge, however, is to design an algorithm that uses a small amount of state to identify the few large and high-bandwidth flows from a large aggregate of flows and provide them with a congestion signal when appropriate. An important dimension of this problem is that of all the flows carrying large responses, it is most effective to signal flows that are also transmitting at a high rate. These are the flows that are consuming the most bandwidth and hence will produce the greatest effect when they reduce their rate. DCN has two main components: identification of high-bandwidth flows and a decision procedure for determining when early congestion notification is in order.

Identifying High-Bandwidth Flows

DCN's approach to identifying large and high-bandwidth flows is based on the idea that packets of high-bandwidth flows are closely paced (i.e., their interarrival times are short) [EV02]. DCN tracks the number of packets that have been recently seen from each flow. If this count exceeds a threshold, the flow is considered to be a long-lived and high-bandwidth flow and the flow's rate is monitored and its packets are eligible for dropping or ECN-marking. As long as a flow remains classified as high-bandwidth, it remains eligible for dropping/marking. If a flow reduces its transmission rate, it is removed from the list of monitored flows and is no longer eligible for dropping/marking.

DCN uses two hash tables for classifying flows: HB (high bandwidth) and SB (score-

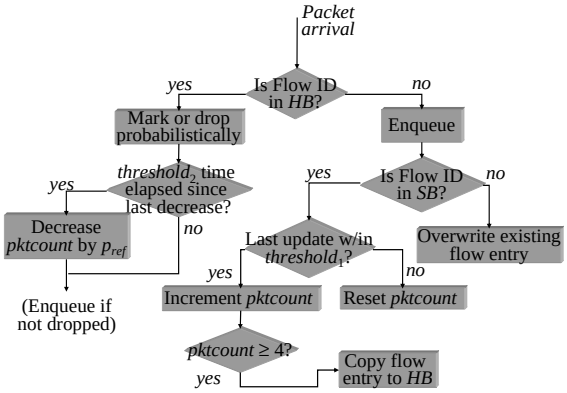

Figure 4.101: High-level flow chart of DCN

board). The HB table tracks flows that are considered high-bandwidth and stores each flow's flow ID (IP addressing 5-tuple) and forwarded packet count. The SB table tracks a fixed number of flows that are *potential* high-bandwidth flows. SB stores the flow ID and recent forwarded packet count of these potential high-bandwidth flows.

When a packet arrives at a router, HB is checked to see if this packet belongs to a high-bandwidth flow. If the packet's flow is found in HB, then it is handled as described below. If the packet's flow ID is not HB, then the packet is enqueued and its flow is tested to see if it should be entered into HB. This test is performed by looking for the flow's ID in SB. If the flow ID is not present, it is added to SB. If the flow ID is present in SB, its packet count is incremented.

A flow is classified as long-lived and high-bandwidth if the number of packets from the flow arriving within a clearing interval, exceeds a threshold. Once the flow's packet count in SB has been incremented, if the count exceeds the threshold, the flow's entry in SB is added to HB. If no packets have been received for the flow within a clearing interval, its packet count is reset to 0.

A high-level flow chart of the DCN algorithm is given in Figure 4.101. All operations on SB are performed in $O(1)$ time. Further, since the number of flows identified as high-bandwidth is small, hash collisions in HB are rare for a table size of a few thousand entries. Thus, operations on HB are also usually executed in $O(1)$ time.

Decision Procedure for Dropping/Marking Packets

Packets from a high-bandwidth flow are marked or dropped with a probability $(1 - p_{ref}/pktcount)$, where $pktcount$ is the number of packets from that flow that have arrived at the router within a period of T_{dec}, and p_{ref} is the current fair share of a flow on a congested link. When congestion is suspected in the router, DCN targets high-bandwidth flows for dropping in proportion to their deviation from their fair share (p_{ref} packets within

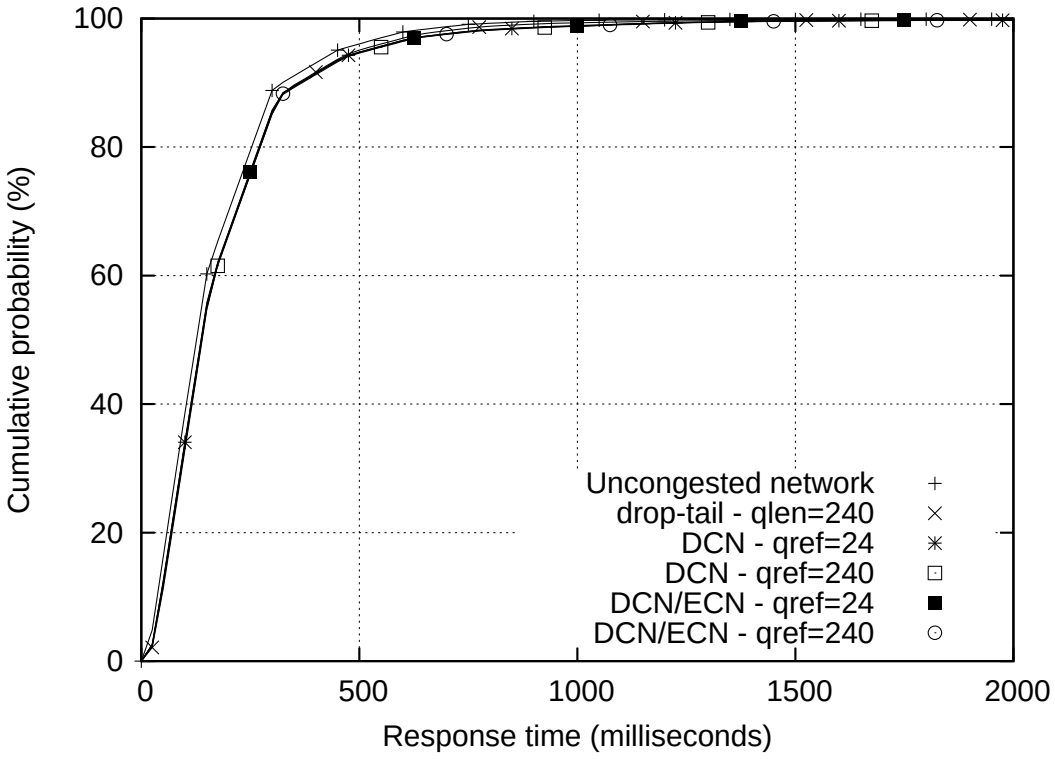

Figure 4.102: Performance of DCN with and without ECN at 80% load

an interval T_{dec}) [SSZ98, PBPS03].

DCN uses a simple control theoretic algorithm based on the well-known PI controller to compute p_{ref}. The instantaneous length of the queue in the router is periodically sampled with period T. A flow's fair share of the queue at the k^{th} sampling period is given by

$$p_{ref}(kT) = a(q(kT) - q_{ref}) - b(q((k-1)T) - q_{ref}) + p_{ref}((k-1)T) \tag{4.2}$$

where a and b, $a < b$, are control coefficients (constants), $q(kT)$ is the length of the queue at a given time kT and q_{ref} is a target queue length value for the controller. Since $a < b$, p_{ref} decreases when the queue length is larger than q_{ref} (an indication of congestion). Thus, packets from high-bandwidth flows are marked or dropped with a high probability. When congestion abates and the queue length drops below q_{ref}, p_{ref} increases and the probability of marking or dropping becomes low. Pan et al. and Misra et al. use the same equation in the design of AFD and PI respectively [PBPS03, HMTG01].

The flow ID of a high-bandwidth flow is kept in HB as long as its counter *pktcount* is positive. After each interval T_{dec}, the counter *pktcount* is decreased by p_{ref}. If the counter *pktcount* of a high-bandwidth flow is smaller than 0, the flow is released from HB.

Figure 4.102, 4.103, 4.104, and 4.105 show the experimental results for DCN with and without ECN at 80%, 90%, 98%, and 105% loads. These results for DCN were obtained

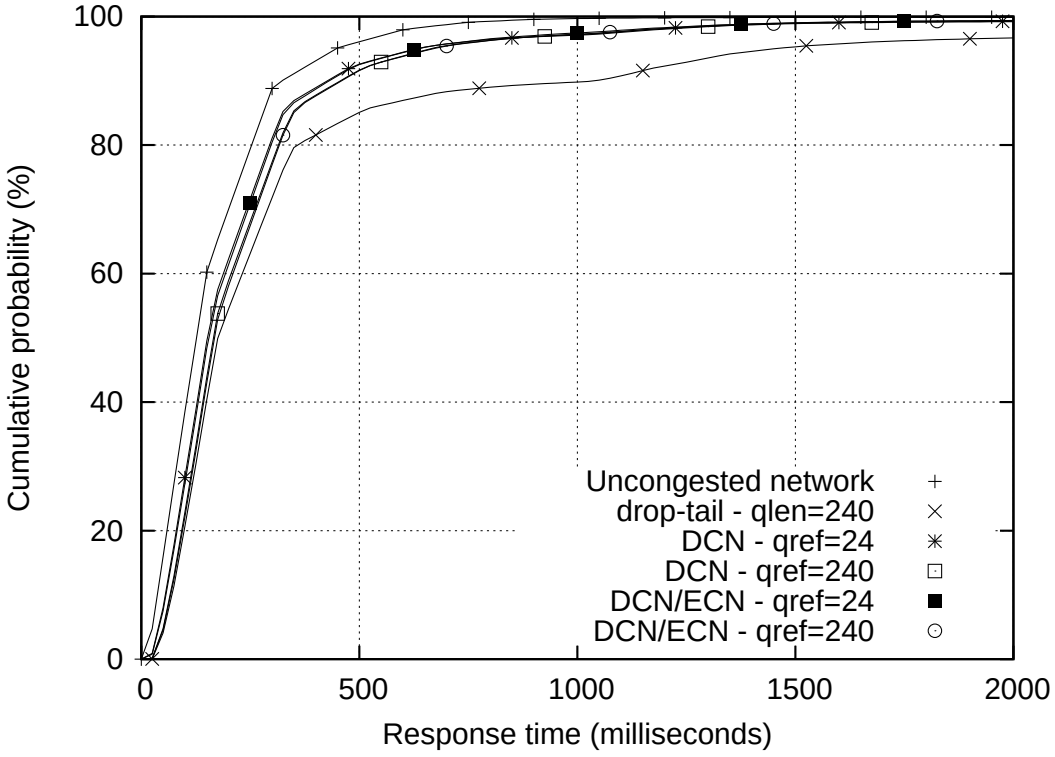

Figure 4.103: Performance of DCN with and without ECN at 90% load

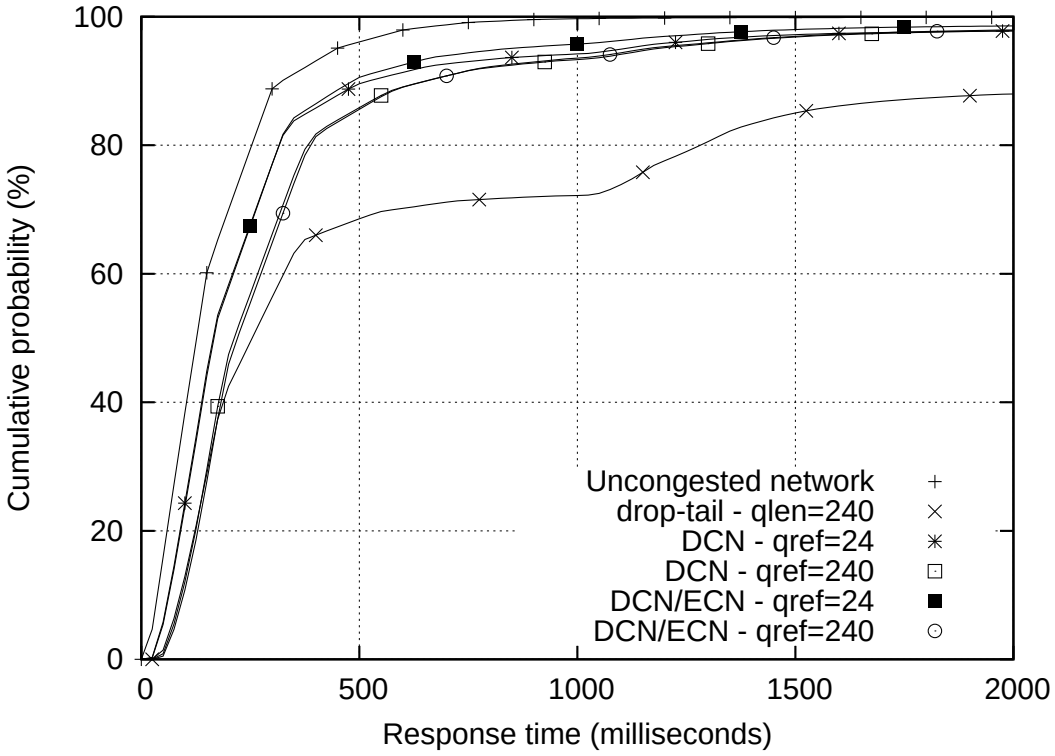

Figure 4.104: Performance of DCN with and without ECN at 98% load

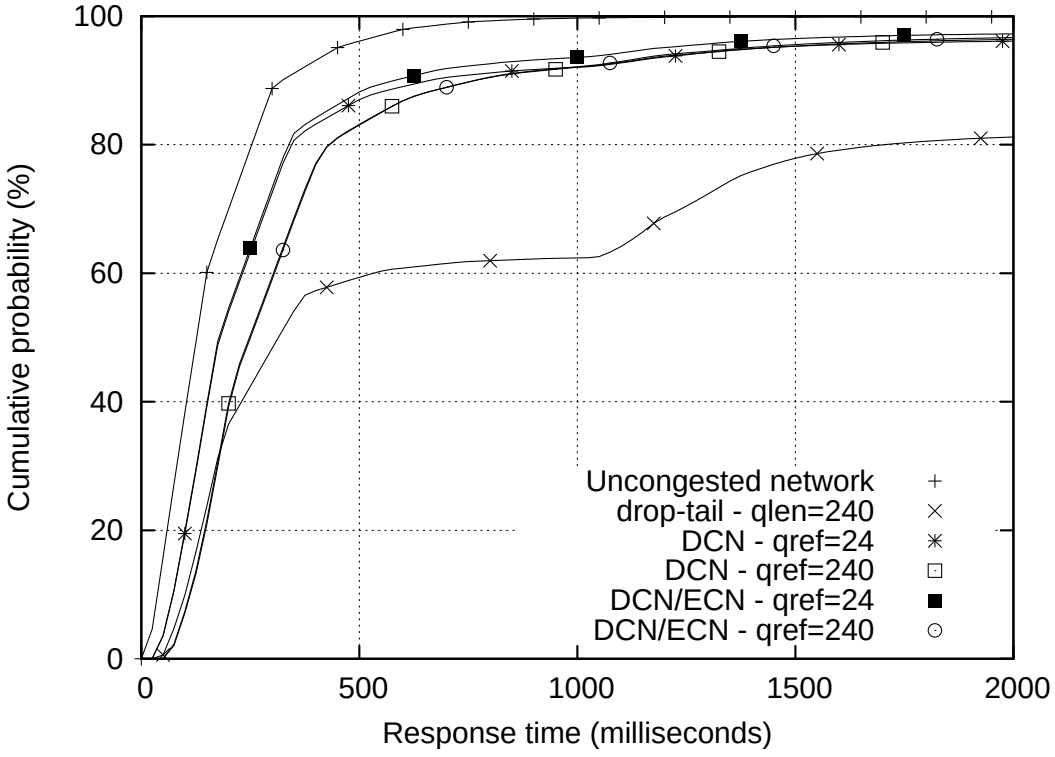

Figure 4.105: Performance of DCN with and without ECN at 105% load

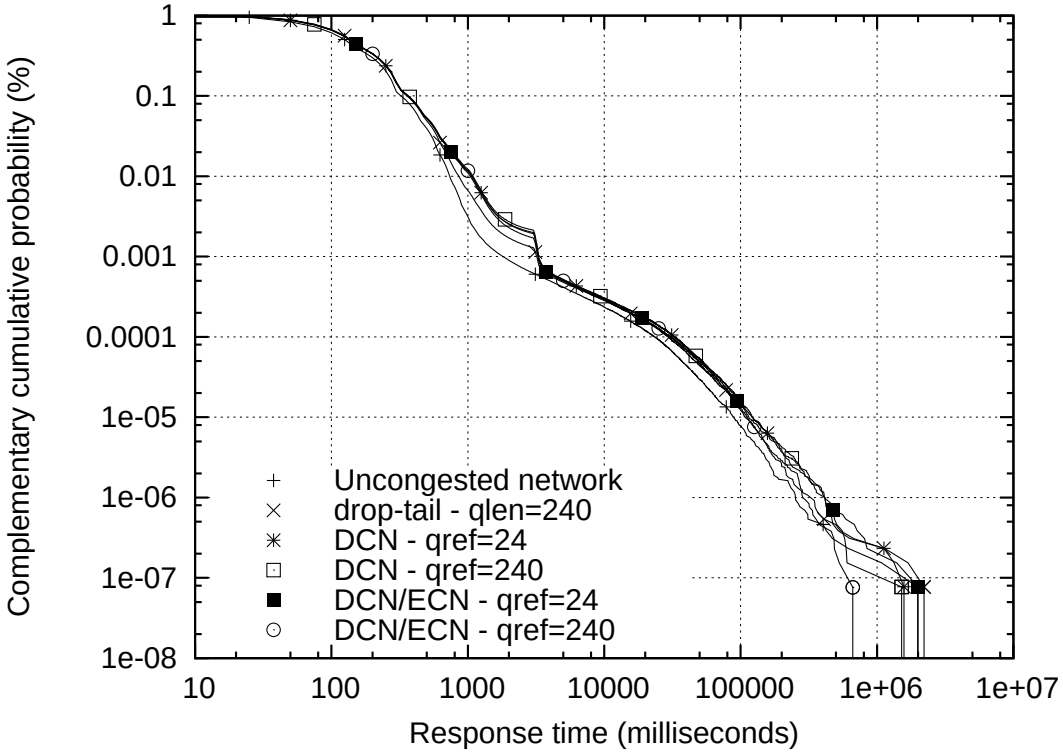

Figure 4.106: Performance of DCN with and without ECN at 80% load (CCDF)

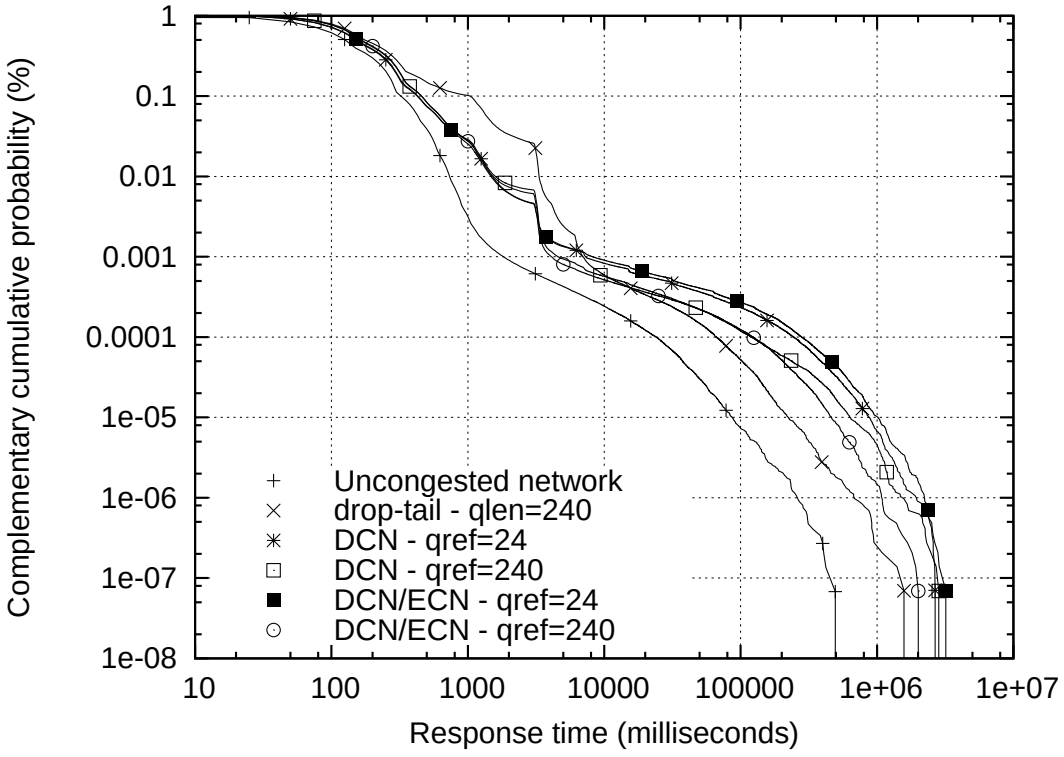

Figure 4.107: Performance of DCN with and without ECN at 90% load (CCDF)

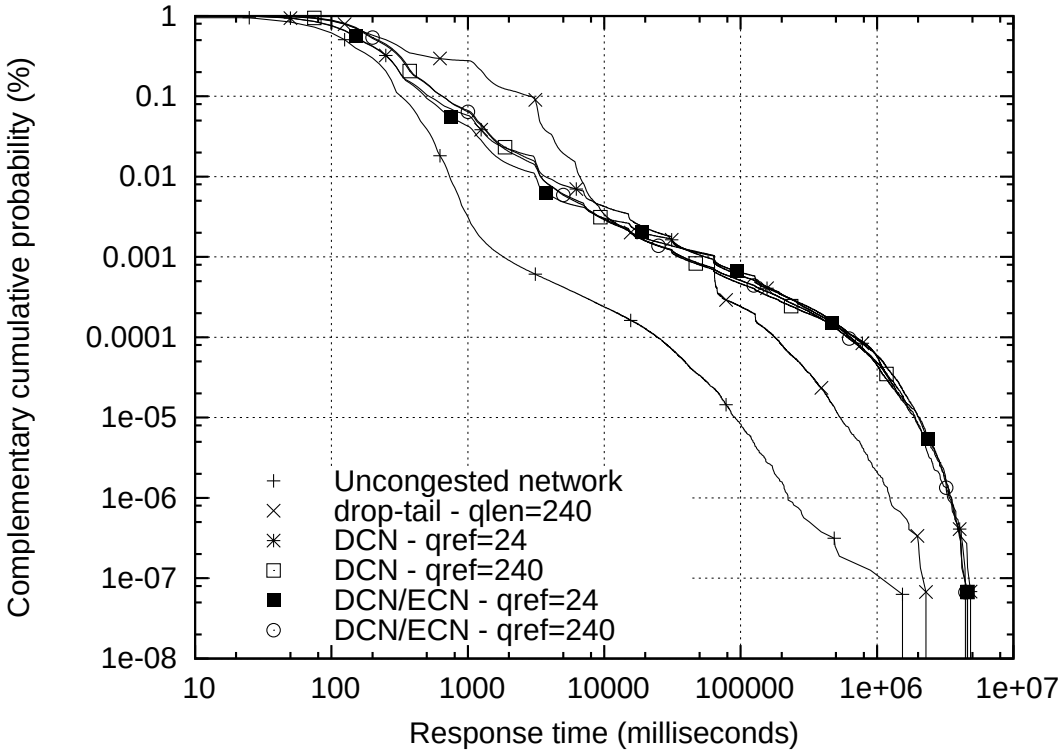

Figure 4.108: Performance of DCN with and without ECN at 98% load (CCDF)

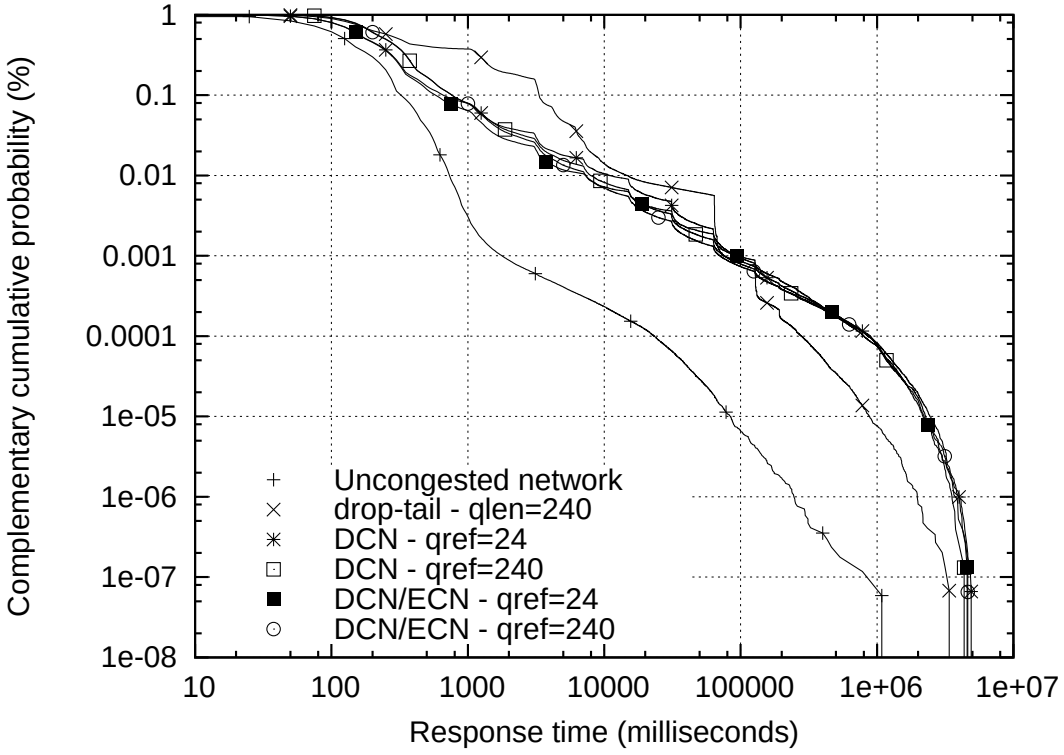

Figure 4.109: Performance of DCN with and without ECN at 105% load (CCDF)

with a queue reference of 24 and 240 packets.

At 80% load, DCN obtained good performance with both queue reference values of 24 and 240 packets. DCN with and without ECN obtained equally good performance at this load. The performance for DCN at this load was comparable with that of drop-tail and came close to the performance of the uncongested network.

At 90% load, DCN also delivered good performance with both queue reference values and significantly outperformed drop-tail. DCN obtained with both queue reference values good performance even without ECN and the addition of ECN did not improve the performance for DCN at this load.

As the offered load increased to 98% and 105%, DCN obtained good performance with both queue reference values at these loads but it gave slightly better performance with a queue reference of 24 packets. At these high loads, DCN continued to obtain good performance even without ECN and significantly outperformed drop-tail. This positive result demonstrated the benefits of differential treatment of flows.

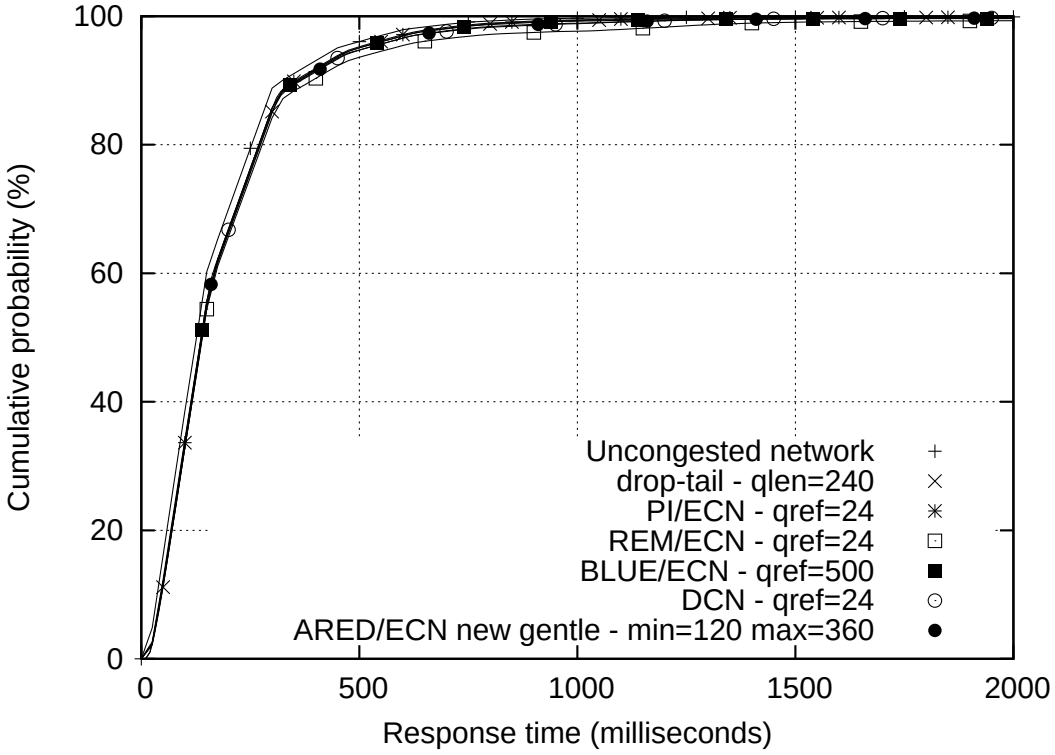

Figure 4.110: Comparison of all AQM algorithms at 80% load

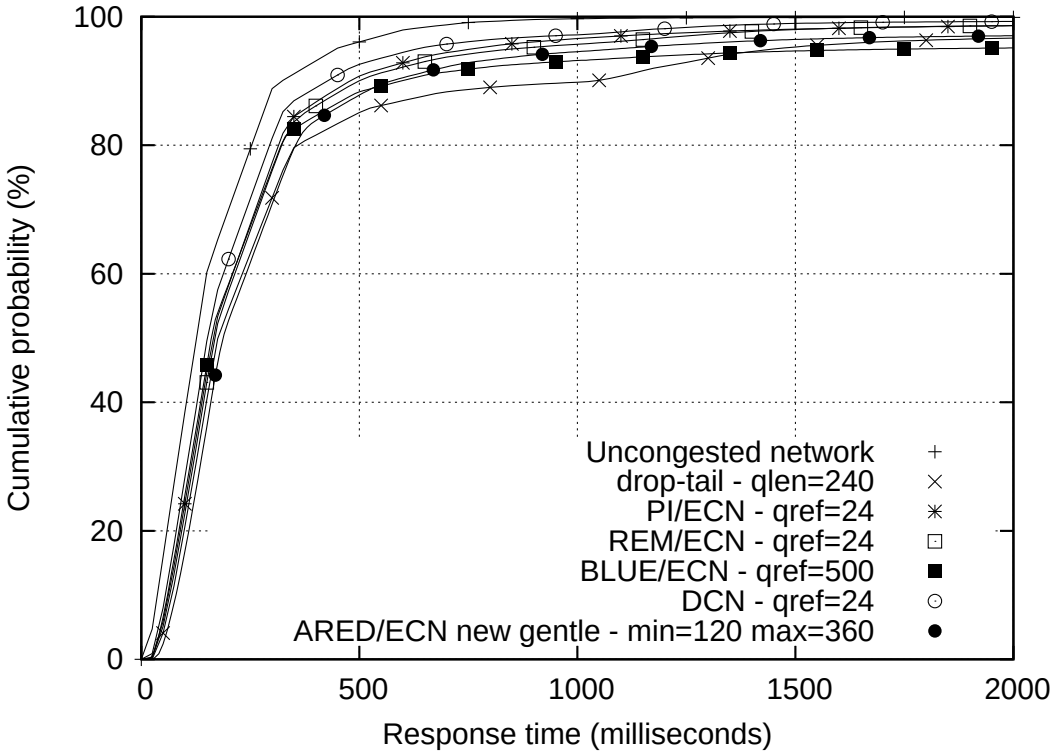

Figure 4.111: Comparison of all AQM algorithms at 90% load

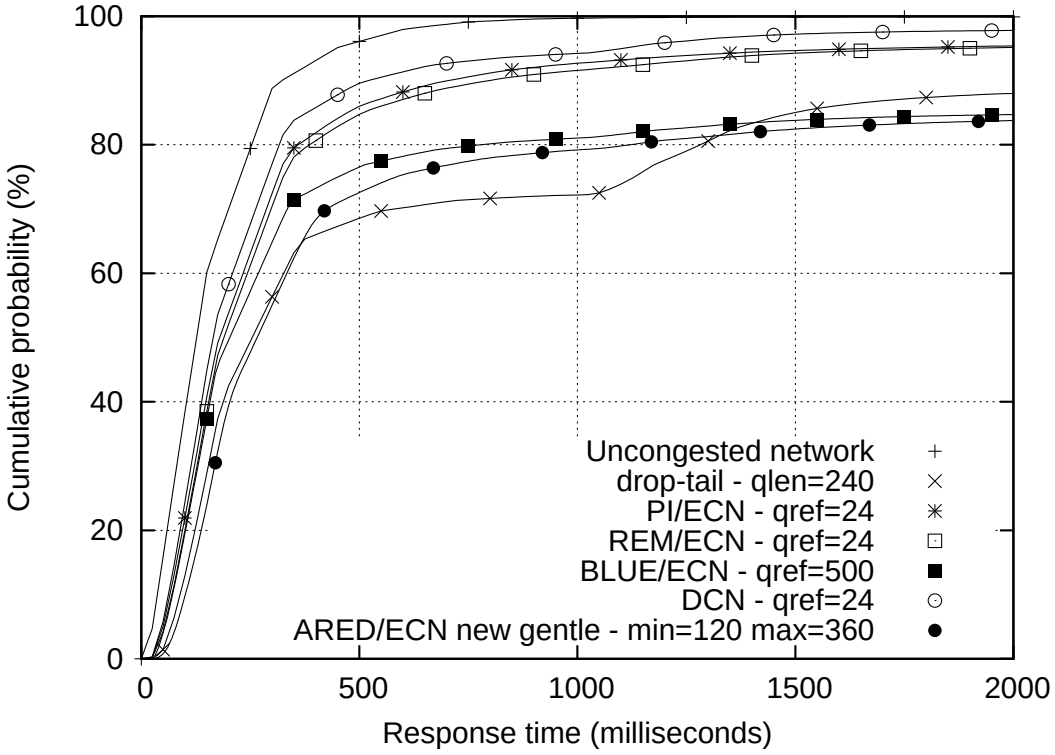

Figure 4.112: Comparison of all AQM algorithms at 98% load

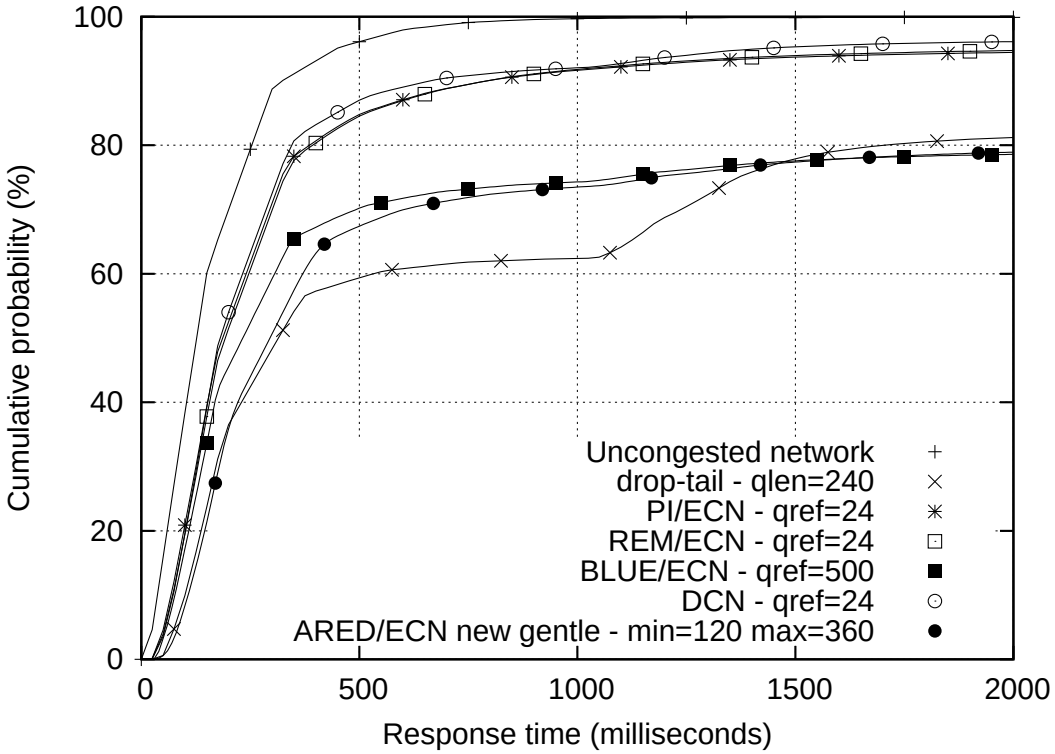

Figure 4.113: Comparison of all AQM algorithms at 105% load

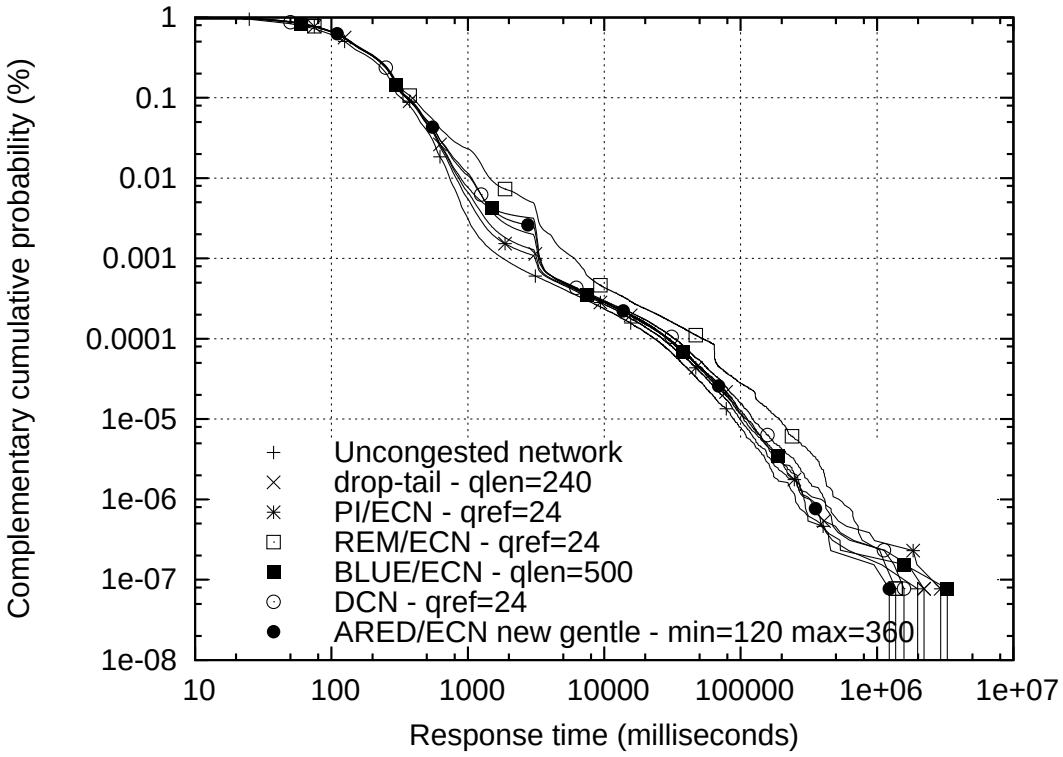

Figure 4.114: Comparison of all AQM algorithms at 80% load (CCDF)

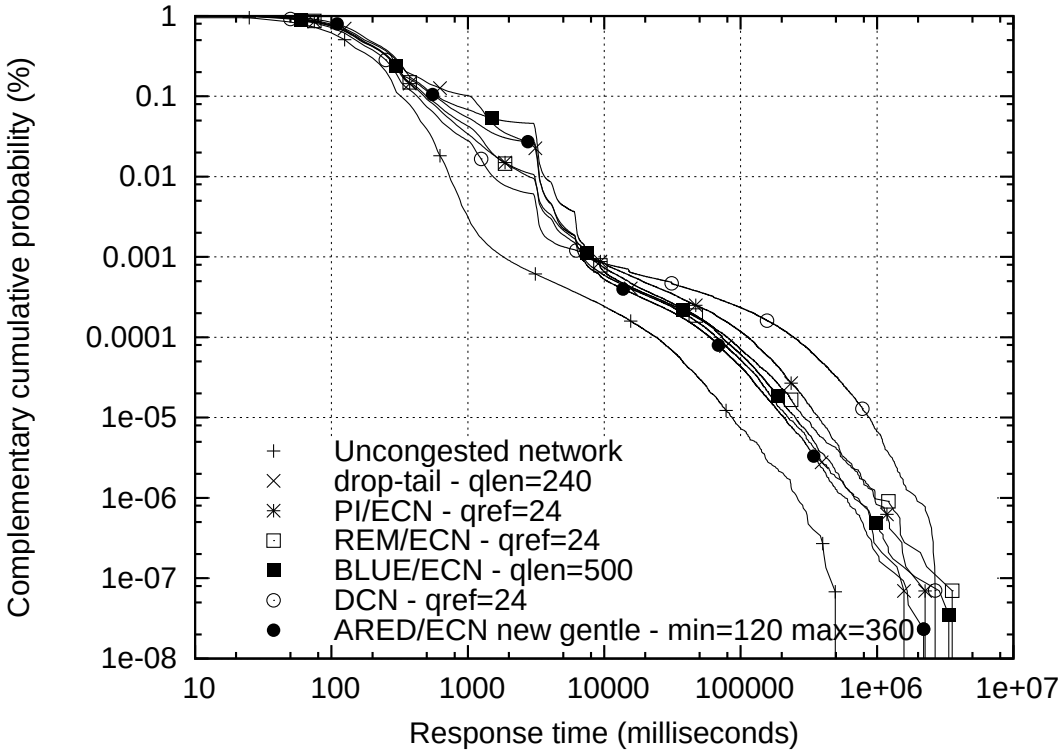

Figure 4.115: Comparison of all AQM algorithms at 90% load (CCDF)

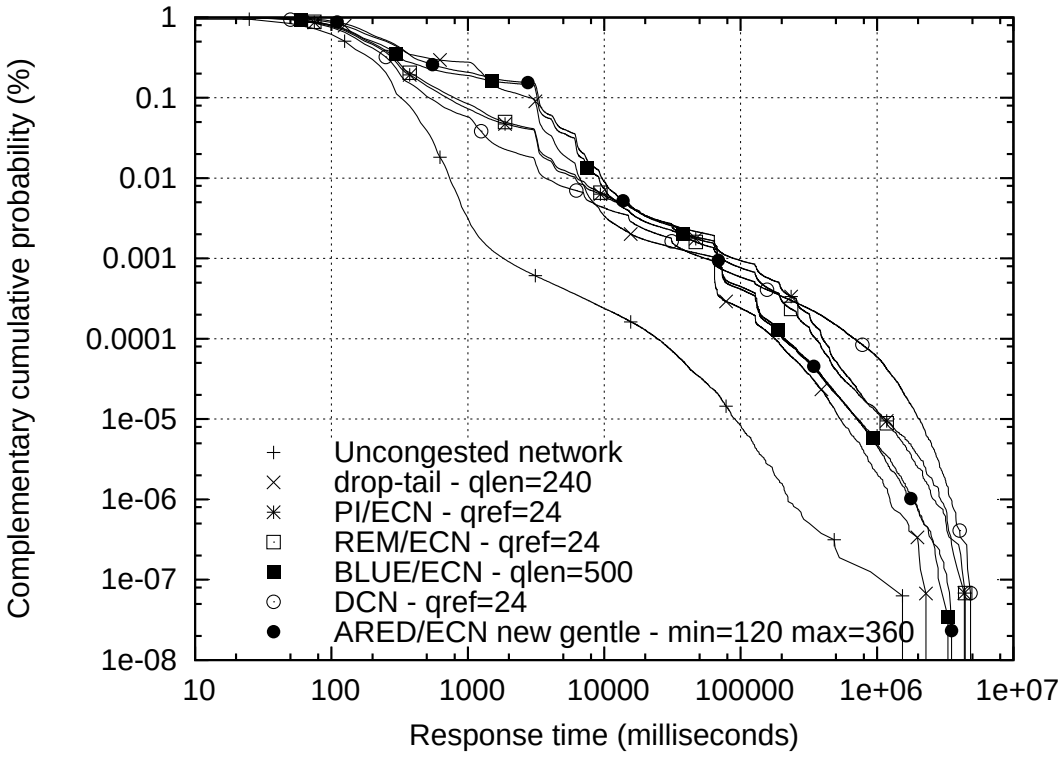

Figure 4.116: Comparison of all AQM algorithms at 98% load (CCDF)

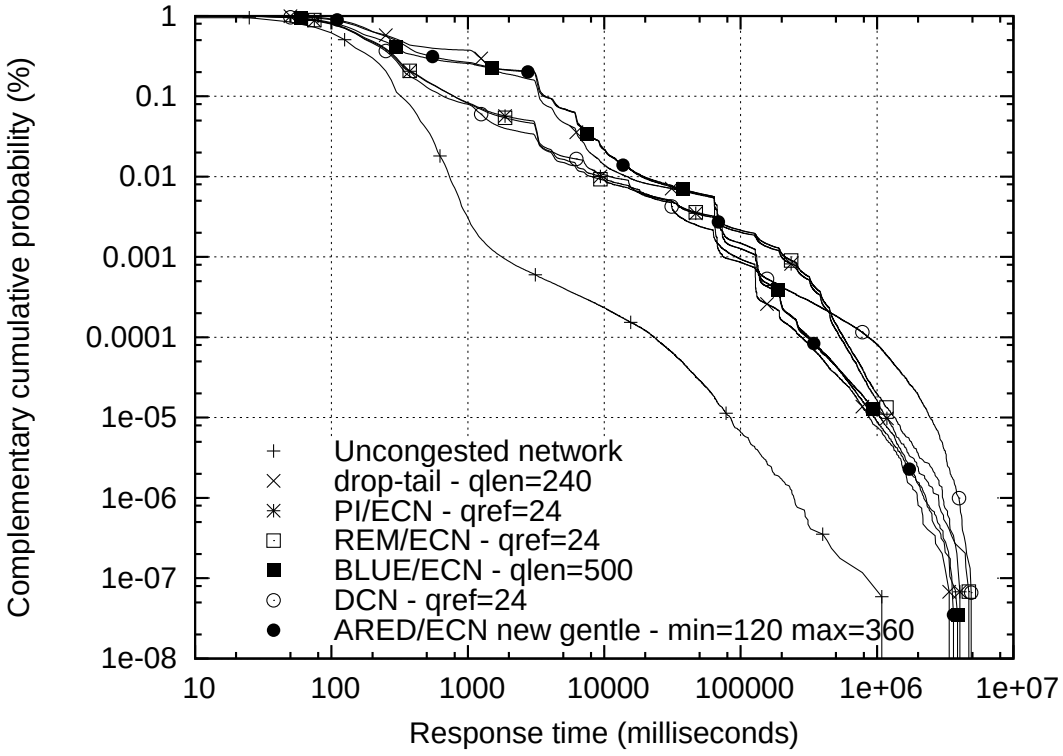

Figure 4.117: Comparison of all AQM algorithms at 105% load (CCDF)

4.9 Comparison of All Results

Figure 4.110, 4.111, 4.112, and 4.113 show a comparison of drop-tail, PI, REM, ARED, BLUE, and DCN at 80%, 90%, 98%, and 105% loads. Experimental results shown in these figures were obtained with the best parameter settings for each of the algorithms. PI, REM, and ARED were chosen to represent AQM algorithms that operate around a target queue threshold 4.2. The BLUE algorithm was chosen to represent AQM algorithms that avoid queue overflows by adjusting the probability for marking or dropping arriving packets (BLUE and AVQ in section 4.3. The DCN algorithm represented AQM algorithms that use differential treatment of flows 4.8.

At 80% offered load, all AQM algorithms delivered good performance that was comparable with the performance of drop-tail and approximated the performance of the uncongested network.

At 90% load, all AQM algorithms continued to deliver good performance and outperformed drop-tail. This positive result demonstrated the benefit of AQM at high loads. Among the AQM algorithms, DCN obtained the best performance even without ECN and BLUE with ECN and a queue length of 500 packets gave the worst performance.

As the offered load increased to 98% and 105%, all AQM algorithms continued to outperform drop-tail. The DCN algorithm continue to give the best performance even without ECN and demonstrated the benefits of differential treatment of flows. The PI and REM algorithms closely approximated the performance of DCN but these algorithms needed the addition of the ECN signaling protocol. The BLUE and ARED algorithms obtained better performance than drop-tail but their performance improvement over drop-tail was not significant.

4.10 Summary

In this Chapter, experimental results with Web traffic and a uniform RTT distribution were presented for a number of AQM algorithms that have been proposed recently in research literature. When response times are used as the primary performance measure, the experimental results in this Chapter lead to the following conclusions.

- At offered loads of 80% or lower, drop-tail with a queue length of 240 packets obtained performance that was competitive to that of all AQM algorithms. Further, since drop-tail closely approximated the performance of the uncongested network at this load, there appears to be no need for AQM at 80% offered load or lower.

- As the offered load increased to 90% or higher and when AQM algorithms were used with packet drops, ARED "byte mode", LQD and PI were the best performing algorithm among AQM algorithms that do not apply differential treatment of flows.

However, the performance improvement for PI and LQD could not offset the performance degradation that all non-differential AQM algorithms suffered at these high loads. LQD obtained slightly better performance than PI by balancing between queuing delay and loss rates.

- When AQM algorithms were used with ECN, they gained significant performance improvement and significantly outperformed drop-tail at 90% load or higher.
- The original ARED algorithm obtained very poor performance and consistently gave poorer performance than drop-tail at all loads. Further, the performance for the original ARED algorithm was not improved by the addition of the ECN signaling protocol.
- Two modification of the ARED algorithms, ARED "byte mode" and ARED/ECN "new gentle", gave significant performance improvement over the original ARED algorithm.
- Differential treatment of flows could obtain performance improvement for a majority of flows significantly. For example, the DCN algorithm could approximate the performance of the uncongested network without ECN at very high loads. Further, even when DCN was operated with packet drops, it still outperformed other AQM algorithms that were used with ECN.

Table 4.1: Loss rate, completed requests, and link utilization

	Offered load	Loss rate (%)		Completed requests (millions)		Link throughput (Mbps)	
		No ECN	ECN	No ECN	ECN	No ECN	ECN
Uncongested 1 Gbps network (drop-tail)	80%	0.0		13.2		80.6	
	90%	0.0		15.0		91.3	
	98%	0.0		16.2		98.2	
	105%	0.0		17.3		105.9	
Drop-tail queue size = 24	80%	0.2		13.2		80.3	
	90%	2.7		14.4		88.4	
	98%	6.5		14.9		91.1	
	105%	9.1		15.0		91.8	
Drop-tail queue size = 240	80%	0.0		13.2		80.6	
	90%	1.8		14.6		89.9	
	98%	6.0		15.1		92.0	
	105%	8.8		15.0		92.4	
Drop-tail queue size = 2400	80%	0.0		13.1		80.4	
	90%	0.1		14.7		88.6	
	98%	3.6		15.1		91.3	
	105%	7.9		15.0		91.1	
PI $q_{ref} = 24$	80%	0.0	0.0	13.3	13.2	80.2	79.3
	90%	1.3	0.3	14.4	14.6	87.9	88.6
	98%	3.9	1.8	15.1	14.9	89.3	89.4
	105%	6.5	2.5	15.1	15.0	89.9	89.5
PI $q_{ref} = 240$	80%	0.0	0.0	13.1	13.1	80.1	80.1
	90%	0.1	0.1	14.7	14.7	87.2	88.2
	98%	3.7	1.7	14.9	15.1	90.0	89.6
	105%	6.9	2.3	15.0	15.2	90.5	90.8
REM $q_{ref} = 24$	80%	0.01	0.0	13.2	13.1	79.8	80.1
	90%	1.8	0.1	14.4	14.6	86.4	88.2
	98%	5.0	1.7	14.5	14.9	87.6	89.6
	105%	7.7	2.4	14.6	14.9	87.5	89.3

Continued on next page

	Offered load	Loss rate (%)		Completed requests (millions)		Link throughput (Mbps)	
		No ECN	ECN	No ECN	ECN	No ECN	ECN
REM	80%	0.0	0.0	13.2	13.2	79.3	80.3
$q_{ref} = 240$	90%	3.3	0.2	14.0	14.7	83.3	88.6
	98%	5.4	1.6	14.4	15.1	86.2	90.4
	105%	7.3	2.3	14.6	15.1	87.7	90.4
ARED	80%	0.02	0.03	13.0	12.9	79.4	78.8
$th_{min} = 12$	90%	1.5	1.3	13.8	13.8	85.5	85.5
$th_{max} = 36$	98%	4.1	4.1	14.0	13.9	87.4	88.0
	105%	5.1	5.1	14.1	14.1	87.3	87.7
ARED	80%	0.02	0.02	13.0	13.1	80.2	80.5
$th_{min} = 120$	90%	1.4	1.2	14.0	14.1	85.5	86.2
$th_{max} = 360$	98%	4.8	4.7	14.2	14.1	87.9	88.2
	105%	6.8	6.3	13.9	13.9	85.2	85.8
ARED	80%	0.0		13.2		79.2	
"byte mode"	90%	0.8		14.6		88.0	
$th_{max} = 12$	98%	3.6		14.8		89.4	
$th_{min} = 36$	105%	6.0		14.8		88.5	
ARED	80%	0.0		13.2		80.2	
"byte mode"	90%	0.9		14.6		87.6	
$th_{max} = 120$	98%	4.2		14.6		87.8	
$th_{min} = 360$	105%	6.7		14.7		87.9	
ARED/ECN	80%		0.0		13.2		79.5
"new gentle"	90%		1.0		14.2		87.5
$th_{max} = 12$	98%		2.4		14.6		88.1
$th_{min} = 36$	105%		3.5		14.5		88.5
ARED/ECN	80%		0.0		13.2		79.8
"new gentle"	90%		0.9		14.6		88.4
$th_{max} = 120$	98%		2.3		14.6		88.7
$th_{min} = 360$	105%		3.2		14.6		89.0
BLUE	80%	0.1	0.1	13.2	13.0	78.9	79.4
queue size =	90%	1.4	2.3	14.6	14.4	87.5	87.6
240	98%	5.9	7.6	14.9	14.5	90.8	88.0
	105%	8.5	10.4	15.0	14.1	91.4	86.9

Continued on next page

	Offered load	Loss rate (%)		Completed requests (millions)		Link throughput (Mbps)	
		No ECN	ECN	No ECN	ECN	No ECN	ECN
BLUE queue size = 500	80%	0.1	0.1	13.1	13.1	79.8	80.0
	90%	1.8	2.0	14.5	14.6	88.6	88.4
	98%	5.4	6.8	15.0	14.6	90.8	88.9
	105%	8.2	9.7	15.0	14.5	91.4	89.6
AVQ queue size = 240	80%	0.6	0.0	13.0	13.2	78.4	80.2
	90%	4.2	0.0	13.8	14.7	83.8	88.4
	98%	7.0	0.7	14.1	15.1	84.0	90.5
	105%	9.1	2.1	14.1	15.2	84.0	90.8
AVQ queue size = 500	80%	0.6	0.0	13.2	13.1	80.2	79.1
	90%	3.9	0.0	13.9	14.7	83.8	88.6
	98%	6.9	0.0	14.0	15.2	83.9	90.6
	105%	9.2	2.2	14.1	15.3	84.0	90.8
LQD q_{ref} = 24	80%	0.0	0.0	13.3	13.3	80.0	80.1
	90%	0.4	0.2	14.7	14.7	88.5	88.7
	98%	2.7	1.4	15.3	15.4	91.6	91.7
	105%	4.9	2.3	15.6	15.6	91.9	91.9
LQD q_{ref} = 240	80%	0.0	0.0	13.3	13.3	80.1	80.2
	90%	0.2	0.1	14.7	14.8	88.3	88.6
	98%	2.6	1.2	15.3	15.4	91.9	92.1
	105%	5.1	2.0	15.7	15.7	92.1	92.2
AFD q_{ref} = 24	80%	0.2	0.2	13.1	13.2	78.7	79.8
	90%	5.0	7.8	13.7	13.1	83.2	79.8
	98%	8.9	11.0	13.6	12.9	81.9	79.6
	105%	10.9	12.8	13.4	12.9	81.3	79.8
AFD q_{ref} = 240	80%	0.0	0.0	13.2	13.1	79.2	80.0
	90%	0.5	0.5	14.7	14.6	88.3	88.3
	98%	6.1	6.2	14.5	14.5	87.6	87.9
	105%	8.9	9.2	14.6	14.4	87.8	87.9
RIO-PS	80%	0.1	0.1	13.3	13.2	79.5	80.0
	90%	2.1	1.3	14.6	14.6	88.8	87.9
	98%	5.3	4.7	15.0	15.2	91.2	91.0
	105%	8.2	7.9	15.3	15.3	91.9	91.9

Continued on next page

	Offered load	Loss rate (%)		Completed requests (millions)		Link throughput (Mbps)	
		No ECN	ECN	No ECN	ECN	No ECN	ECN
DCN q_{ref} = 24	80%	0.0	0.0	13.1	13.2	79.3	78.5
	90%	1.0	0.0	14.5	14.6	86.1	86.9
	98%	2.3	1.4	14.9	15.1	88.6	89.2
	105%	3.0	2.2	15.3	15.5	90.5	90.8
DCN q_{ref} = 240	80%	0.0	0.0	13.2	13.3	80.8	79.6
	90%	0.6	0.4	14.6	14.7	88.4	88.1
	98%	2.6	2.3	15.1	15.2	89.5	89.5
	105%	3.3	3.1	15.5	15.5	91.3	91.4

Table 4.2: Percentiles of response times

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
Uncongested 1 Gbps network (drop-tail)	80%	0.137	0.237	0.312
	90%	0.137	0.237	0.312
	98%	0.137	0.237	0.312
	105%	0.137	0.237	0.312
Drop-tail queue size = 24	80%	0.137	0.237	0.362
	90%	0.162	0.312	1.187
	98%	0.237	1.162	3.112
	105%	0.287	1.387	3.337
Drop-tail queue size = 240	80%	0.137	0.237	0.362
	90%	0.187	0.312	1.037
	98%	0.262	1.137	2.862
	105%	0.312	1.362	3.337
Drop-tail queue size = 2400	80%	0.137	0.237	0.362
	90%	0.262	0.412	0.612
	98%	0.487	0.862	1.887
	105%	0.637	1.662	3.662
PI $q_{ref} = 24$	80%	0.137	0.237	0.362
	90%	0.162	0.287	0.587
	98%	0.212	0.637	1.662
	105%	0.237	1.262	3.162
PI/ECN $q_{ref} = 24$	80%	0.137	0.237	0.362
	90%	0.162	0.287	0.487
	98%	0.187	0.312	0.712
	105%	0.187	0.312	0.787
PI $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.187	0.312	0.462
	98%	0.262	0.562	1.737
	105%	0.337	1.362	3.212
PI/ECN $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.187	0.337	0.537
	98%	0.237	0.362	0.837
	105%	0.237	0.387	0.937

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
REM $q_{ref} = 24$	80%	0.137	0.237	0.362
	90%	0.162	0.312	1.137
	98%	0.212	1.162	2.437
	105%	0.237	1.287	3.337
REM/ECN $q_{ref} = 24$	80%	0.137	0.262	0.387
	90%	0.162	0.287	0.487
	98%	0.187	0.337	0.787
	105%	0.192	0.371	0.801
REM $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.237	0.512	1.612
	98%	0.262	1.237	2.737
	105%	0.287	1.337	3.362
REM/ECN $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.187	0.337	0.537
	98%	0.237	0.362	0.837
	105%	0.237	0.387	0.937
ARED $th_{min} = 12$ $th_{max} = 36$	80%	0.137	0.262	0.637
	90%	0.212	1.112	3.037
	98%	0.262	1.287	3.287
	105%	0.312	1.487	4.087
ARED/ECN $th_{min} = 12$ $th_{max} = 36$	80%	0.137	0.287	0.862
	90%	0.212	1.087	2.787
	98%	0.262	1.262	3.262
	105%	0.312	1.462	3.887
ARED $th_{min} = 120$ $th_{max} = 360$	80%	0.137	0.287	0.737
	90%	0.212	0.837	2.062
	98%	0.287	1.312	3.287
	105%	0.387	1.712	4.312
ARED/ECN $th_{min} = 120$ $th_{max} = 360$	80%	0.137	0.287	0.662
	90%	0.212	0.812	2.062
	98%	0.287	1.287	3.287
	105%	0.387	1.762	4.337

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
ARED	80%	0.137	0.245	0.355
"byte mode"	90%	0.155	0.285	0.565
$th_{min} = 12$	98%	0.205	0.635	2.005
$th_{max} = 36$	105%	0.245	1.265	3.315
ARED	80%	0.145	0.245	0.375
"byte mode"	90%	0.185	0.315	0.615
$th_{min} = 120$	98%	0.265	0.995	2.415
$th_{max} = 360$	105%	0.315	1.375	3.385
ARED/ECN	80%	0.137	0.255	0.395
"new gentle"	90%	0.165	0.305	0.935
$th_{min} = 12$	98%	0.215	0.735	3.245
$th_{max} = 36$	105%	0.245	1.255	3.535
ARED/ECN	80%	0.137	0.245	0.365
"new gentle"	90%	0.185	0.325	0.575
$th_{min} = 120$	98%	0.265	0.585	3.295
$th_{max} = 360$	105%	0.295	1.175	3.705
BLUE	80%	0.137	0.245	0.365
queue size =	90%	0.155	0.295	0.925
240	98%	0.235	1.185	3.065
	105%	0.305	1.385	3.315
BLUE	80%	0.137	0.245	0.365
queue size =	90%	0.165	0.305	1.145
500	98%	0.235	1.155	2.615
	105%	0.305	1.375	3.305
AVQ	80%	0.137	0.245	0.415
queue size =	90%	0.155	0.335	1.515
240	98%	0.185	1.145	2.995
	105%	0.205	1.255	3.615
AVQ	80%	0.137	0.255	0.415
queue size =	90%	0.155	0.305	1.445
500	98%	0.185	1.145	2.875
	105%	0.205	1.265	3.685

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
LQD $q_{ref} = 24$	80%	0.137	0.237	0.362
	90%	0.155	0.285	0.465
	98%	0.225	0.385	1.425
	105%	0.235	1.135	2.005
LQD/ECN $q_{ref} = 24$	80%	0.137	0.237	0.337
	90%	0.137	0.262	0.412
	98%	0.137	0.262	0.462
	105%	0.162	0.262	0.487
LQD $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.165	0.285	0.445
	98%	0.245	0.425	1.465
	105%	0.295	0.725	1.715
LQD/ECN $q_{ref} = 240$	80%	0.137	0.237	0.337
	90%	0.162	0.287	0.437
	98%	0.162	0.287	0.537
	105%	0.187	0.312	0.537
AFD $q_{ref} = 24$	80%	0.137	0.237	0.387
	90%	0.162	0.337	1.487
	98%	0.212	1.137	3.462
	105%	0.245	1.405	5.315
AFD/ECN $q_{ref} = 24$	80%	0.137	0.245	0.365
	90%	0.187	0.437	3.262
	98%	0.212	1.387	4.562
	105%	0.235	3.095	6.205
AFD $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.187	0.312	0.462
	98%	0.237	0.512	2.787
	105%	0.262	1.212	3.687
AFD/ECN $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.187	0.312	0.487
	98%	0.212	0.412	3.212
	105%	0.237	0.612	3.462

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
RIO-PS	80%	0.137	0.245	0.355
	90%	0.165	0.285	0.495
	98%	0.195	0.325	1.035
	105%	0.265	0.485	3.225
RIO-PS/ECN	80%	0.137	0.245	0.365
	90%	0.155	0.275	0.465
	98%	0.185	0.315	0.725
	105%	0.245	0.425	3.165
DCN $q_{ref} = 24$	80%	0.137	0.237	0.362
	90%	0.162	0.262	0.437
	98%	0.162	0.287	0.512
	105%	0.187	0.312	0.662
DCN/ECN $q_{ref} = 24$	80%	0.137	0.237	0.362
	90%	0.162	0.262	0.437
	98%	0.162	0.287	0.487
	105%	0.187	0.312	0.587
DCN $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.162	0.287	0.462
	98%	0.212	0.337	0.662
	105%	0.237	0.387	0.762
DCN/ECN $q_{ref} = 240$	80%	0.137	0.237	0.362
	90%	0.162	0.287	0.462
	98%	0.212	0.362	0.662
	105%	0.262	0.387	0.762

Chapter 5

Results with Web Traffic and General RTT Distributions

The results presented in Chapter 4 were very encouraging and demonstrated the potential benefits of AQM algorithms, especially when they were used in combination with ECN or when they applied appropriate differential treatment of flows. Nevertheless, these results were limited to only Web traffic and wide-area networks inside the continental U.S. where propagation delays between an arbitrary pair of end systems could be approximated by a uniform distribution. In this Chapter, the assumption about the uniform distribution of RTTs is relaxed and the sensitivity of results presented in Chapter 4 to distributions of round-trip times is investigated by performing experiments with Web traffic using a more general distribution of RTTs obtained from a measurement study [AKSJ03]. This general distribution of RTTs (shown in Figures 3.6 and 3.7) was used as an input by the *dummynet* software to model the minimum RTTs between the end systems in the laboratory network.

Using the general distribution of RTTs, experiments were performed with PI, REM, ARED, LQD, and DCN. These AQM algorithms were chosen to obtain a range or "envelope" of results that an AQM algorithm could possibly achieve. The DCN algorithm was chosen to represent differential AQM algorithms. LQD and PI were the best performing non-differential algorithms when AQM was used with packet drops. REM was chosen to demonstrate the potential benefits of ECN (as shown in Chapter 4, the performance of REM was improved significantly by the addition of ECN). The ARED algorithm was chosen because its original design gave the poorest performance among all AQM algorithms but the performance for ARED was significantly improved by two modification ARED "byte mode" and ARED "new gentle" with ECN. For comparison purposes, experimental results were also obtained with drop-tail and with the uncongested network.

Experimental results with the general RTT distribution were obtained aforementioned AQM algorithms at 90% and 98%. This was because AQM algorithms showed no benefits at 80% load or below (as demonstrated in Chapter 4) and the performance degradation for

most AQM algorithms at 105% offered load was so significant that there would be little incentive to operate a network at that load. With the new RTT distribution, the procedure for experiment calibration in section 3.3 was repeated and the numbers of emulated web users were adjusted to achieve the offered loads of 90% and 98%.

The rest of this Chapter is organized as follows. Section 5.1 show experimental results for drop-tail that are used as base results in comparison with results for AQM algorithms. Section 5.2 and 5.3 present experimental results for AQM algorithms when they were used with packet drops and ECN. Section 5.4 and 5.5 give a comparison of results for all AQM algorithms and a summary of the Chapter.

5.1 Results for Drop-Tail

Figures 5.1 and 5.2 show experimental results for drop-tail that were obtained with a queue length of 24 and 240 packets at 90% and 98% loads. These results are used as base results to compare with experimental results for AQM algorithms in subsequent sections.

At 90% load, drop-tail gave similar response times for approximately 60% of flows with queue lengths of 24 and 240 packets. The other 40% of flows experienced better response times with a queue length of 240 packets. Drop-tail experienced some performance degradation when compared to the performance of the uncongested network at 90% load. However, drop-tail still obtained relatively good performance with both queue lengths at this load.

As the offered load increased to 98%, drop-tail suffered noticeable performance degradation with both queue lengths of 24 and 240 packets. At this load, drop-tail again obtained similar performance with both queue lengths for 60% of flows. For the rest 40% of flows, drop-tail obtained better performance with a queue length of 240 packets.

5.2 Results for PI, REM, LQD, DCN, and ARED

Experimental results were obtained for PI, REM, LQD, and DCN with a queue reference of 24 and 240 packets at 90% and 98% loads. Since ARED does not have an explicit parameter to set a queue reference, experiments were performed for ARED "packet mode" and "byte mode" with (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) to achieve a target queue reference of 24 and 240 packets respectively. Experimental results of AQM algorithms were compared with the results of drop-tail and of the uncongested network to investigate the effects of AQM algorithms in an environment of diverse RTTs.

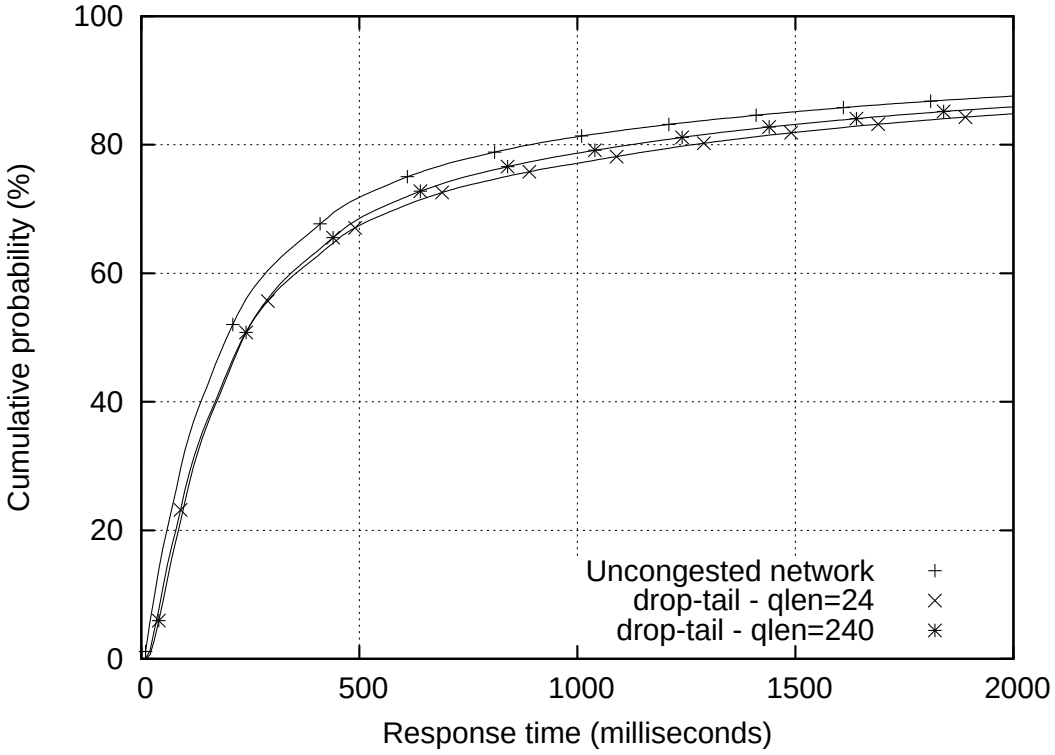

Figure 5.1: Drop-tail performance at 90% load

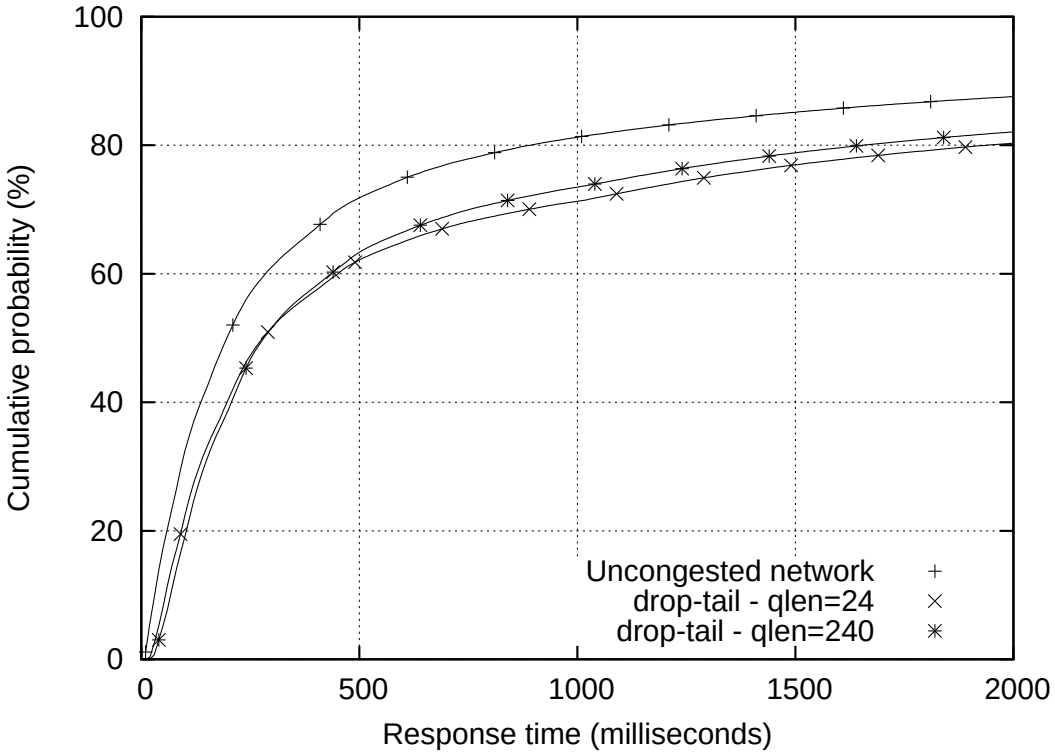

Figure 5.2: Drop-tail performance at 98% load

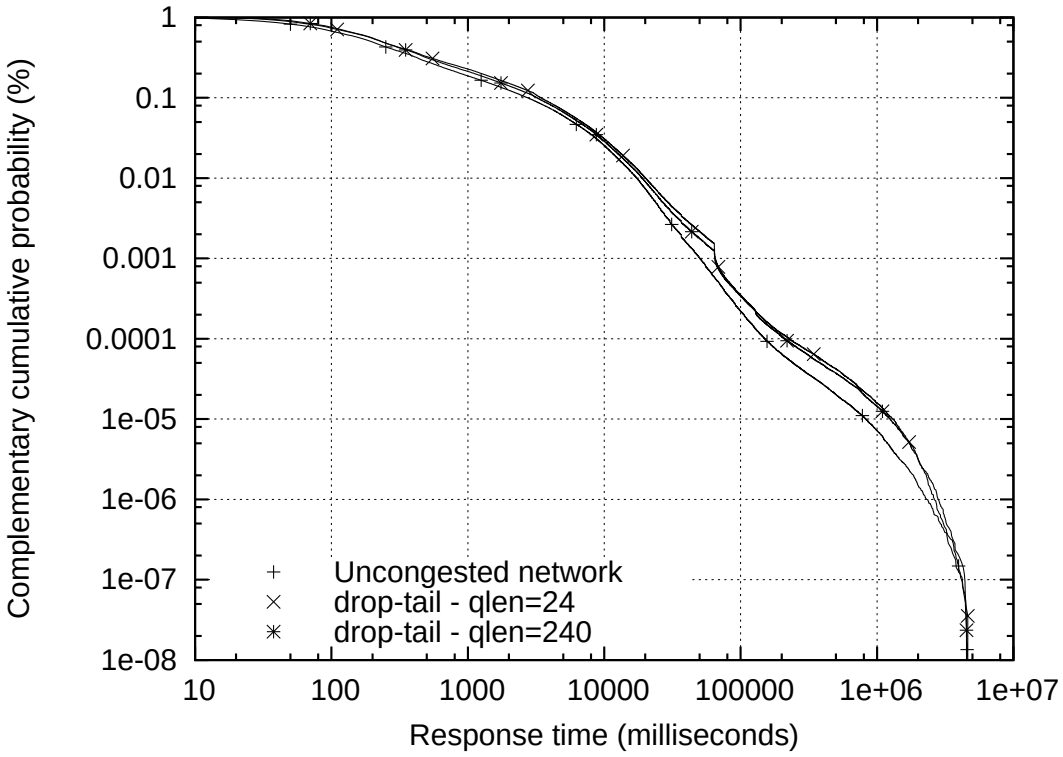

Figure 5.3: Drop-tail performance at 90% load (CCDF)

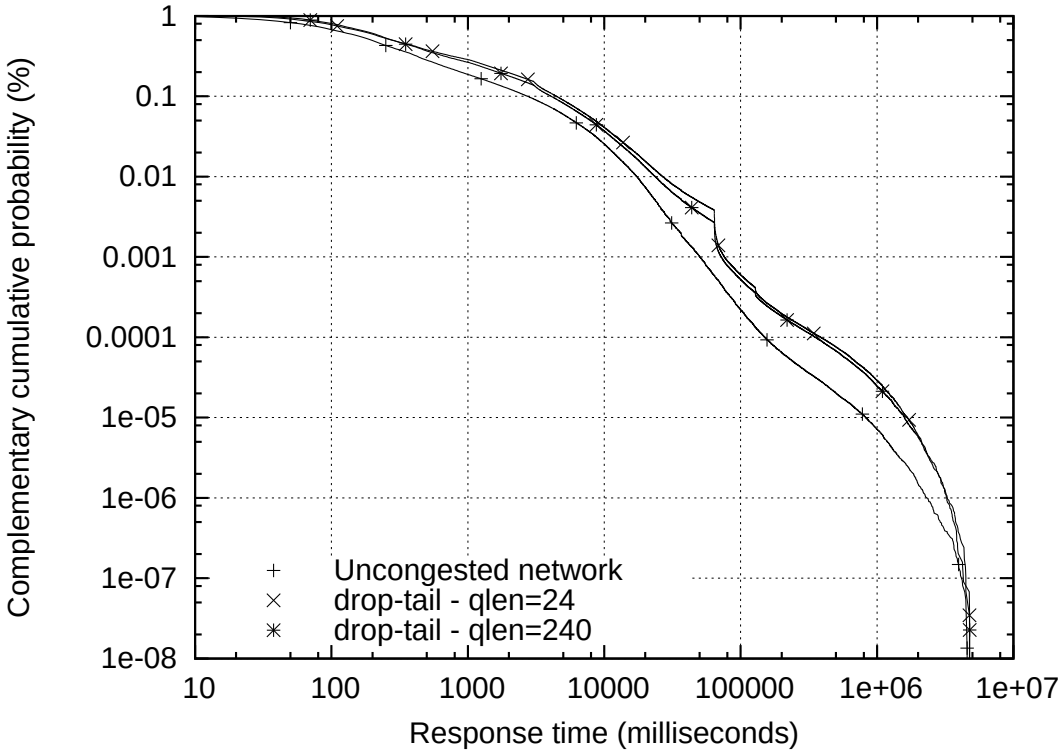

Figure 5.4: Drop-tail performance at 98% load (CCDF)

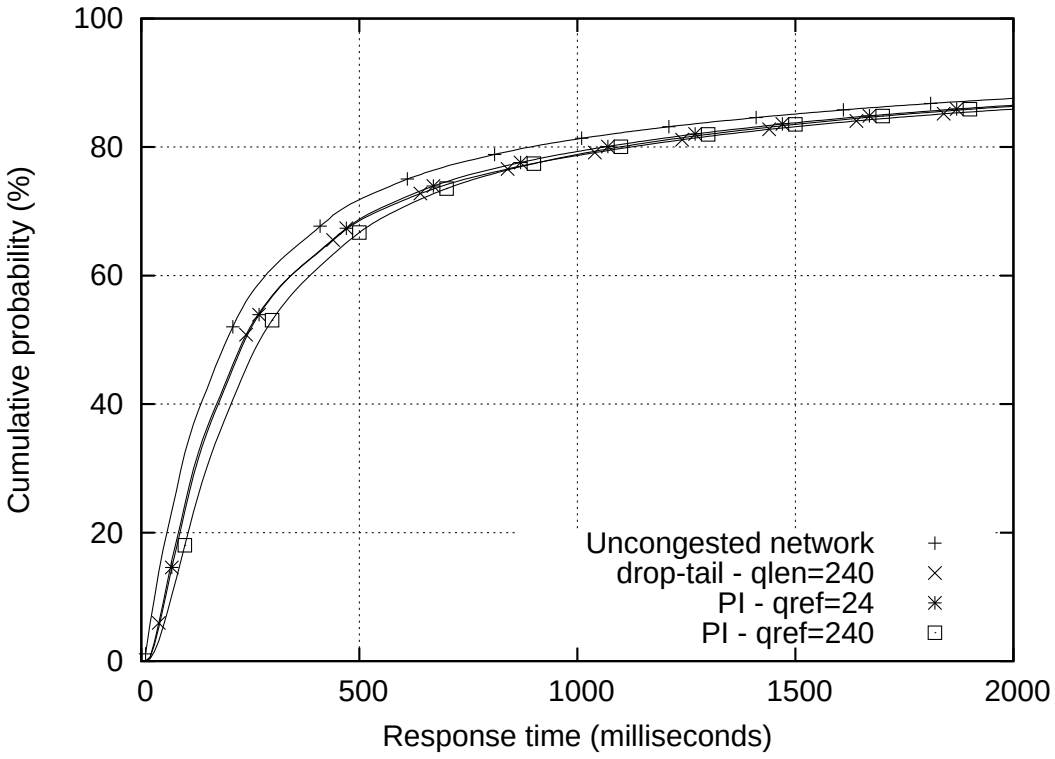

Figure 5.5: PI performance at 90% load

5.2.1 Results for PI

Figures 5.5 and 5.6 show experimental results for PI with a queue reference of 24 and 240 packets at 90% and 98% loads when PI operated with packet drops.

At 90% load, PI with a queue reference of 24 packets obtained similar performance as drop-tail. However, PI with a queue reference of 240 packets gave slightly worse performance than drop-tail for approximately 80% of flows and equal performance to drop-tail for the rest 20% of flows. PI with both queue references of 24 and 240 packets suffered a small performance degradation when compared to the performance of the uncongested network at this load.

At 98% load, the performance for PI with both queue references of 24 and 240 packets degraded considerably. PI with a queue reference of 24 packets gave similar performance as drop-tail. However, PI with a queue reference of 240 packets delivered worse performance than drop-tail for approximately 70% of flows. For the rest 30% of flows, PI with a queue reference of 240 packets obtained similar performance as drop-tail and PI with a queue reference of 24 packets.

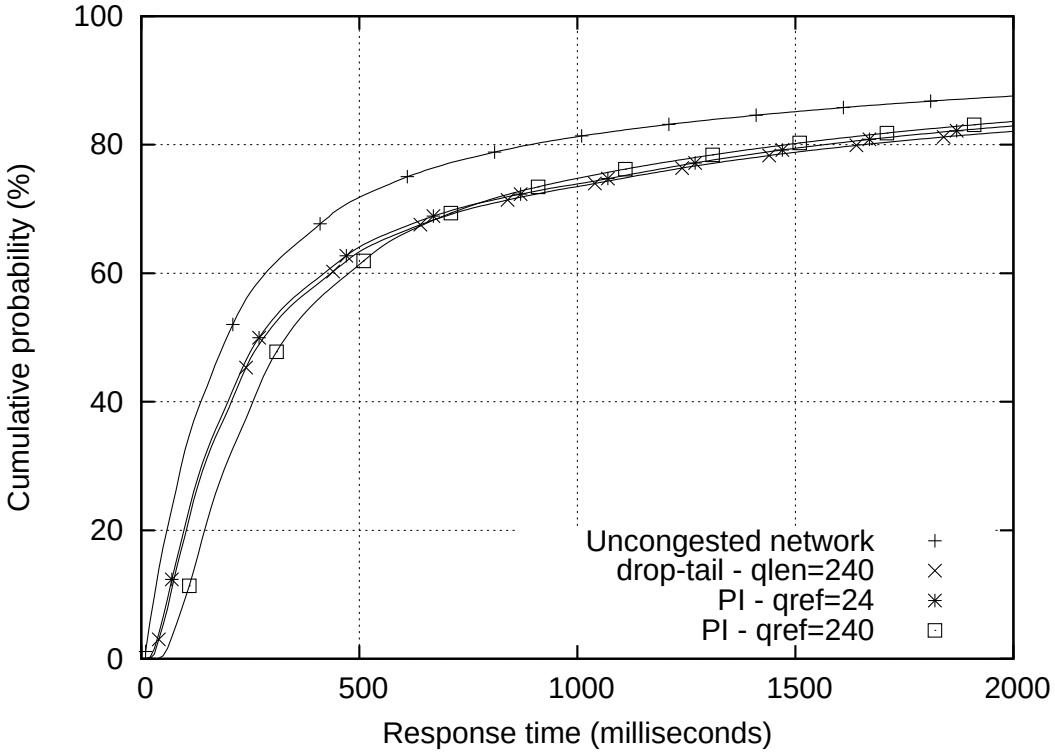

Figure 5.6: PI performance at 98% load

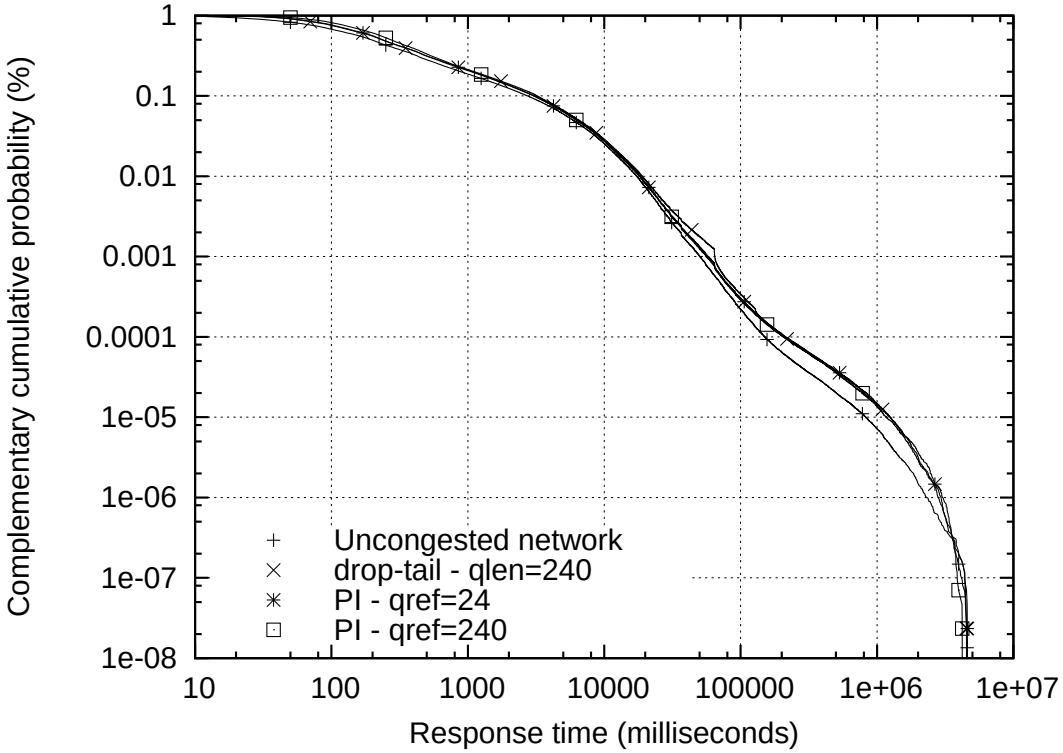

Figure 5.7: PI performance at 90% load (CCDF)

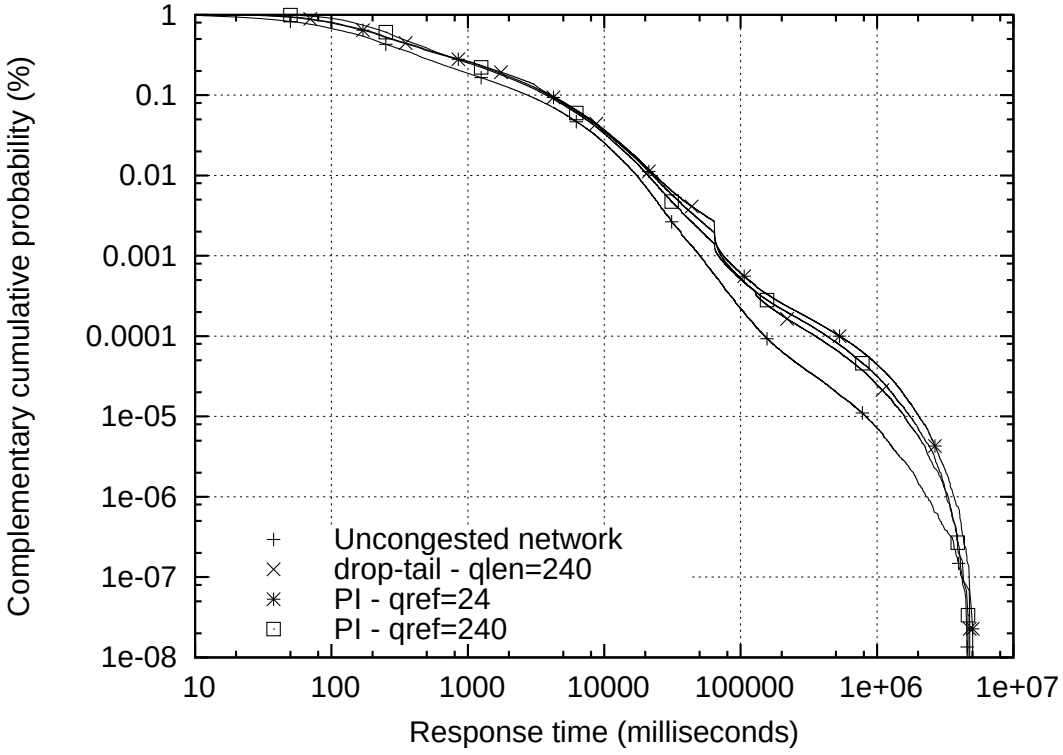

Figure 5.8: PI performance at 98% load (CCDF)

5.2.2 Results for REM

Figures 5.9 and 5.10 show experimental results for REM with a queue reference of 24 and 240 packets at 90% and 98% loads when REM was used with packet drops.

At 90% offered load, REM delivered the same performance with both queue reference of 24 and 240 packets. The performance for REM with both queue references at this load was similar to the performance of drop-tail and closely approximated the performance of the uncongested network.

As the offered load increased to 98% load, the performance for REM with both queue references decreased considerably. The performance for REM with both queue references was slightly worse than that of drop-tail at 98% offered load. Between the two queue references, REM obtained slightly better performance with a queue reference of 24 packets at this load.

5.2.3 Results for ARED

Figures 5.13, 5.14, 5.15, and 5.16 give experimental results for ARED "packet mode" and "byte mode" at 90% and 98% loads. These results were obtained by performing experiments for ARED "packet mode" and "byte mode" with the parameter settings ($th_{min} = 12$ packets,

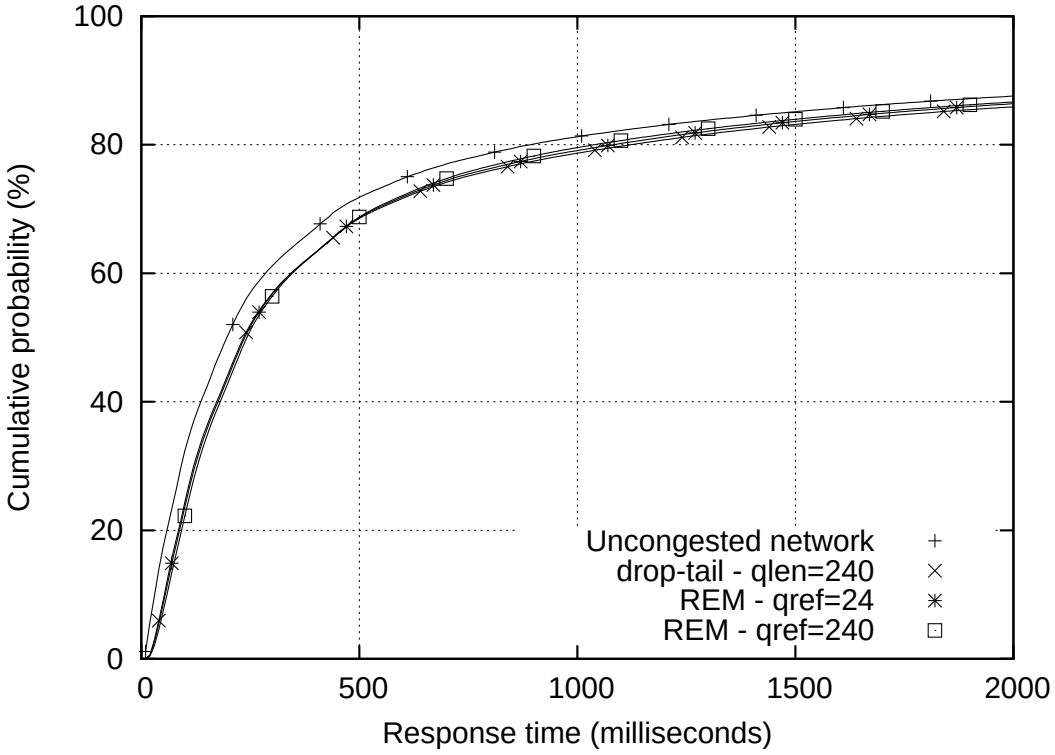

Figure 5.9: REM performance at 90% load

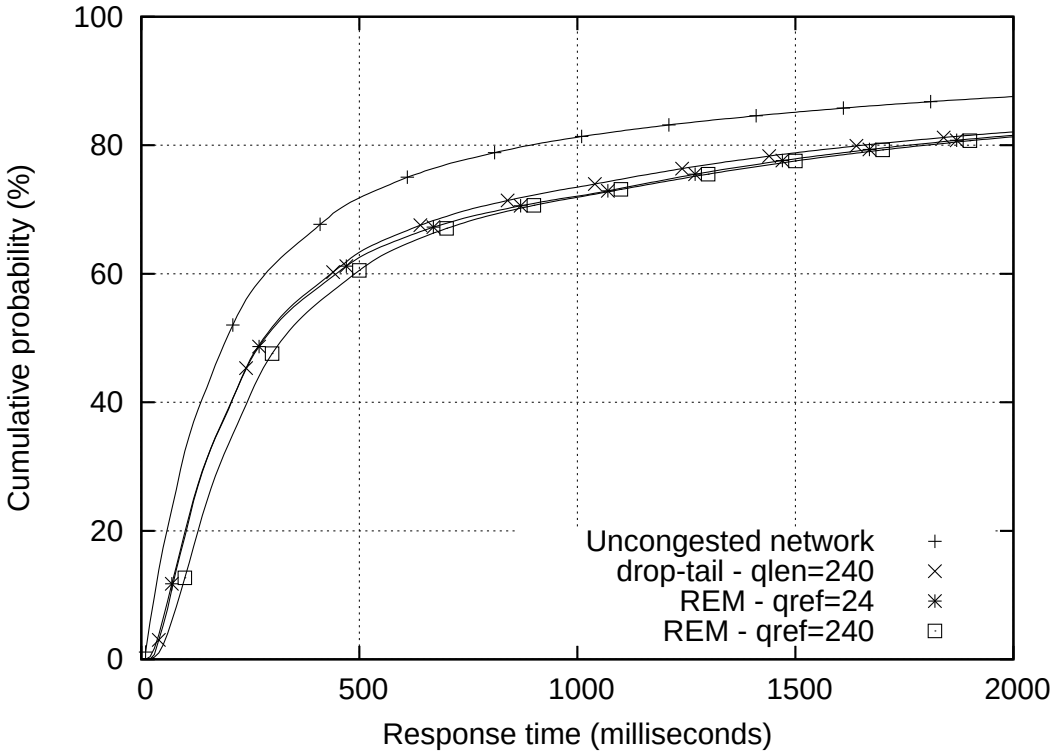

Figure 5.10: REM performance at 98% load

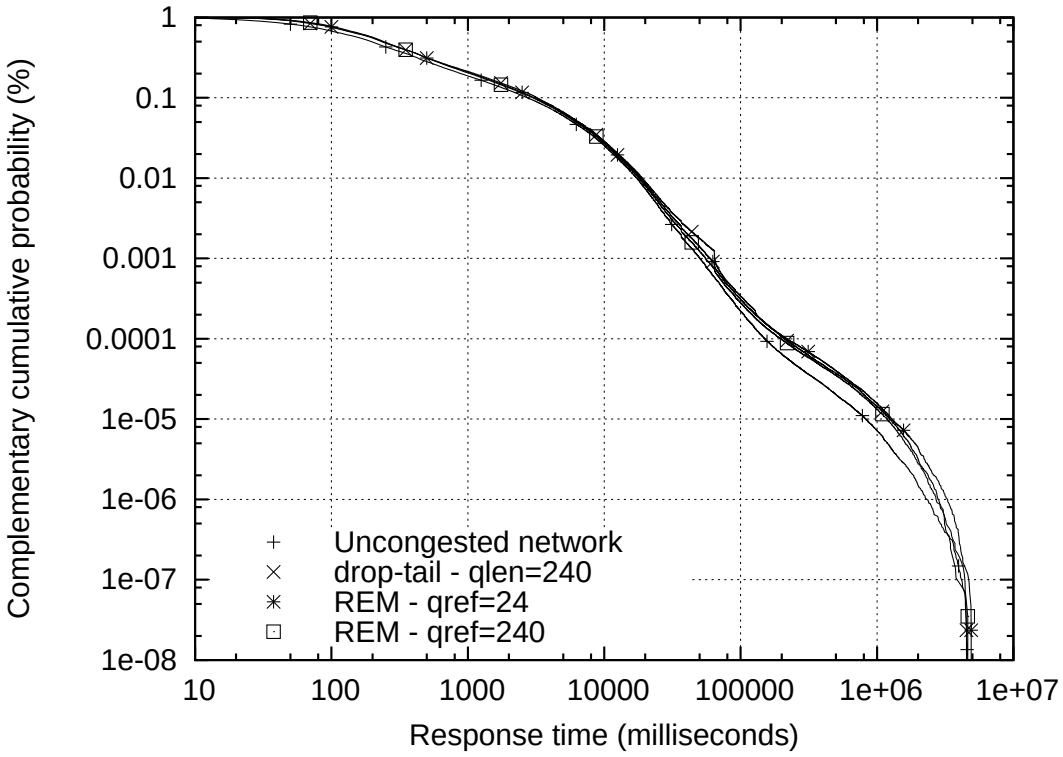

Figure 5.11: REM performance at 90% load (CCDF)

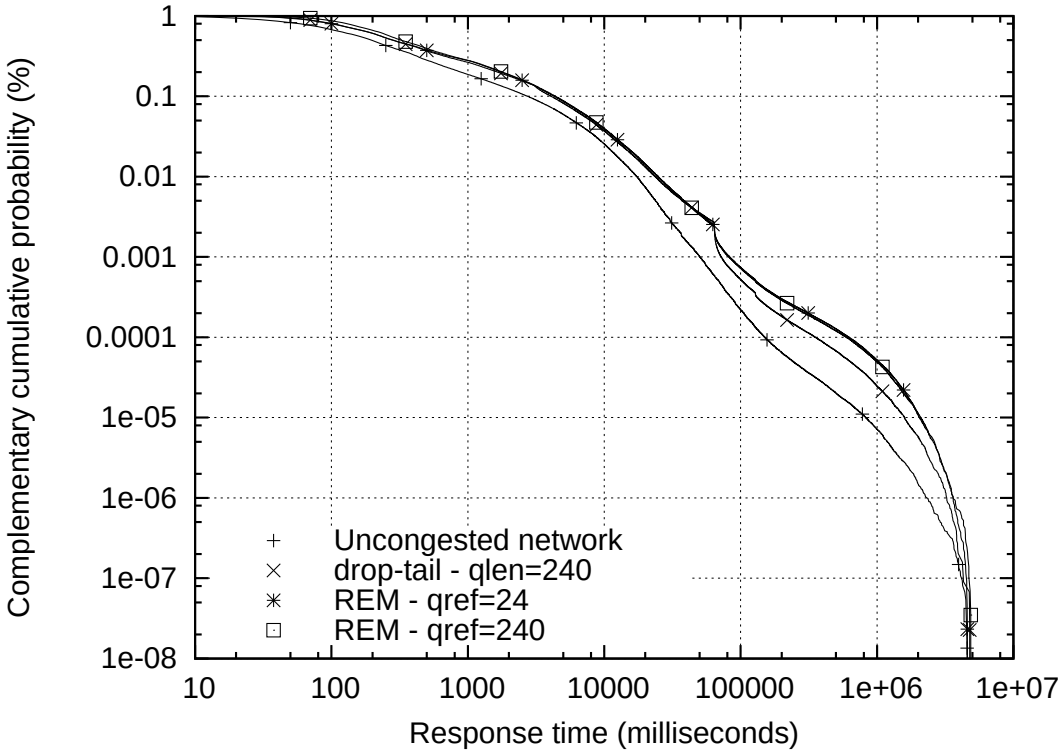

Figure 5.12: REM performance at 98% load (CCDF)

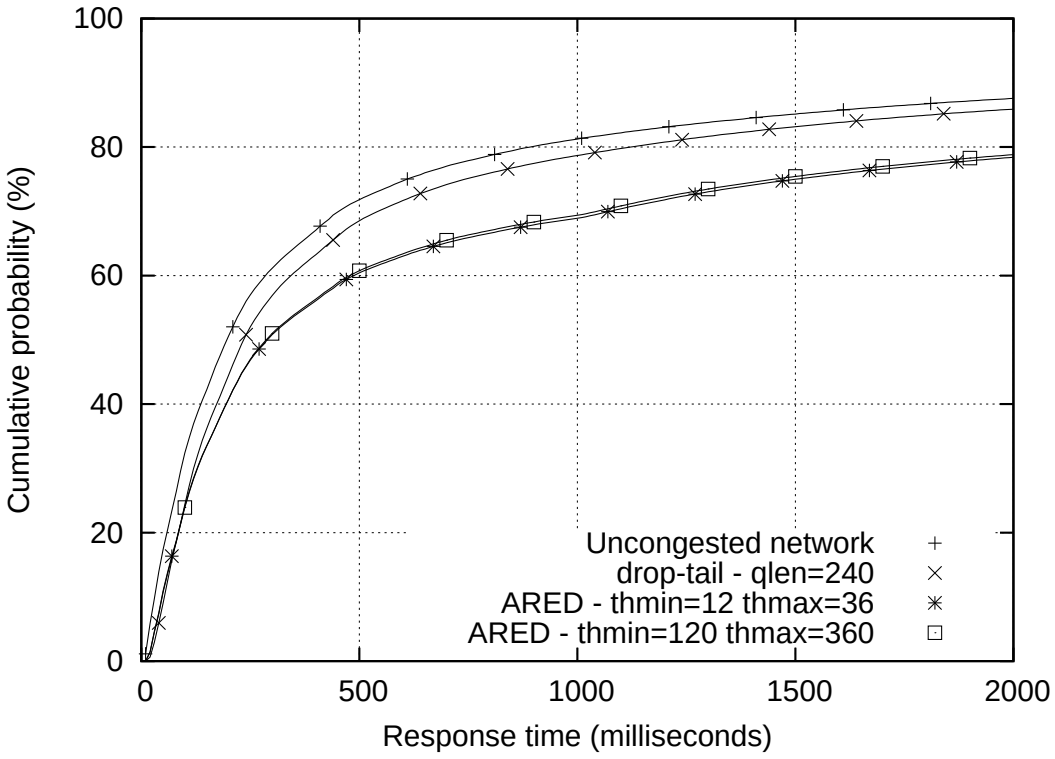

Figure 5.13: ARED packet mode performance at 90% load

th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) to achieve a target queue reference of 24 and 240 packets respectively.

At 90% offered load, ARED "packet mode", the original ARED algorithm, delivered equally poor performance with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). The performance for ARED "packet mode" was significantly lower than that of drop-tail at this load. However, ARED "byte mode" gave good performance with both parameter settings at this load. The performance for ARED "byte mode" was the same as the performance of drop-tail and came close to that of the uncongested network.

At 98% load, ARED "packet mode" suffered even more performance degradation and gave similarly poor performance with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). Although the performance for ARED "byte mode" also decreased considerably, it significantly outperformed ARED "packet mode" and delivered similar performance as drop-tail at this load. ARED "byte mode" obtained the same performance with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) at this load.

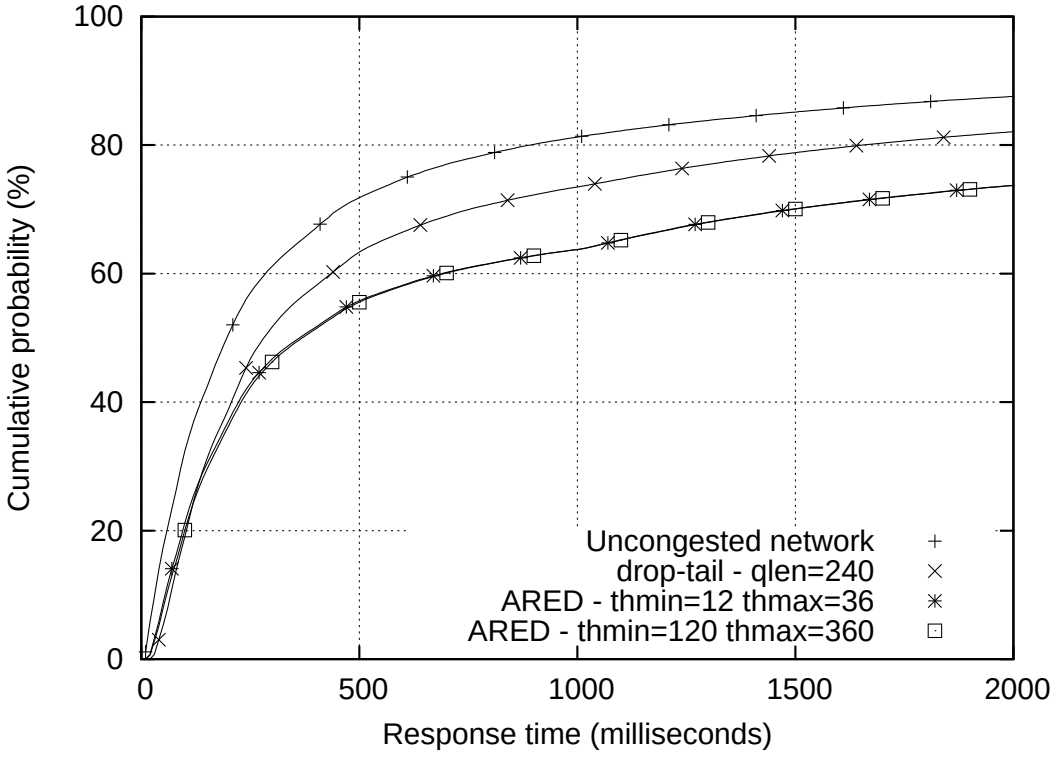

Figure 5.14: ARED packet mode performance at 98% load

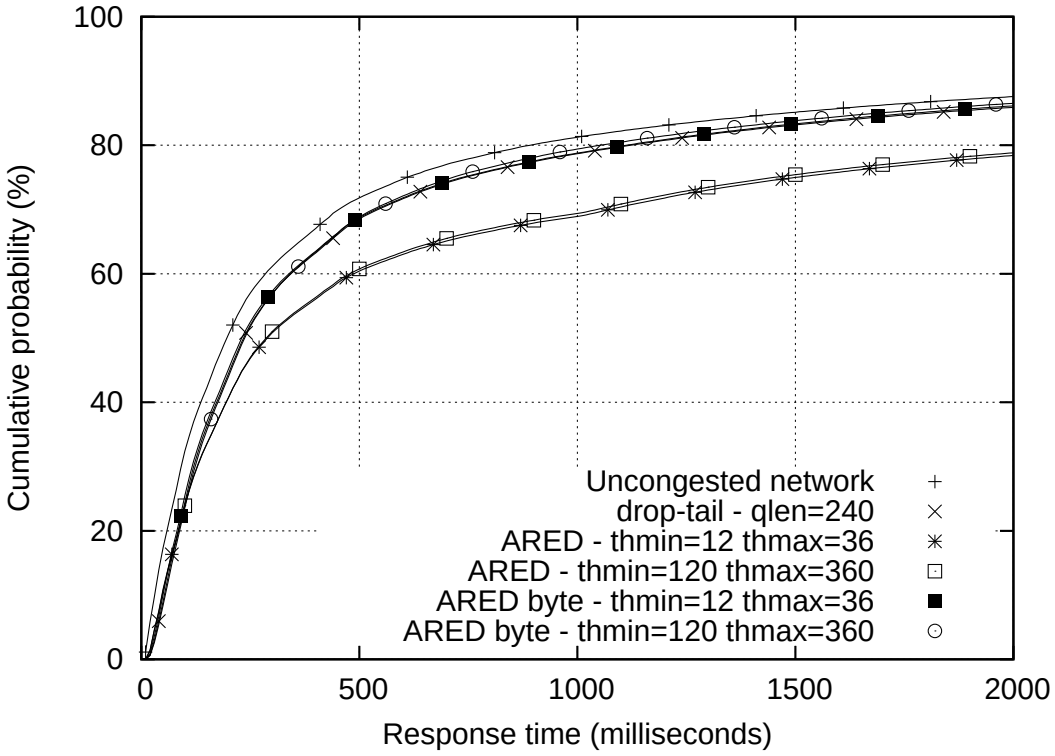

Figure 5.15: ARED byte mode performance at 90% load

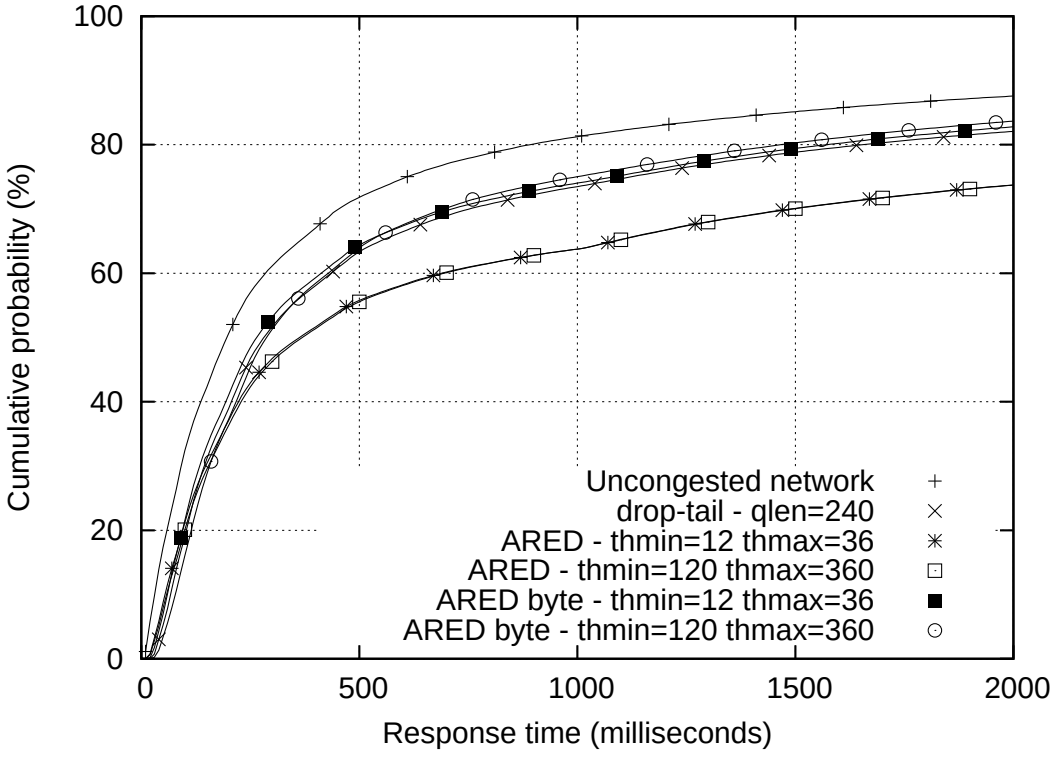

Figure 5.16: ARED byte mode performance at 98% load

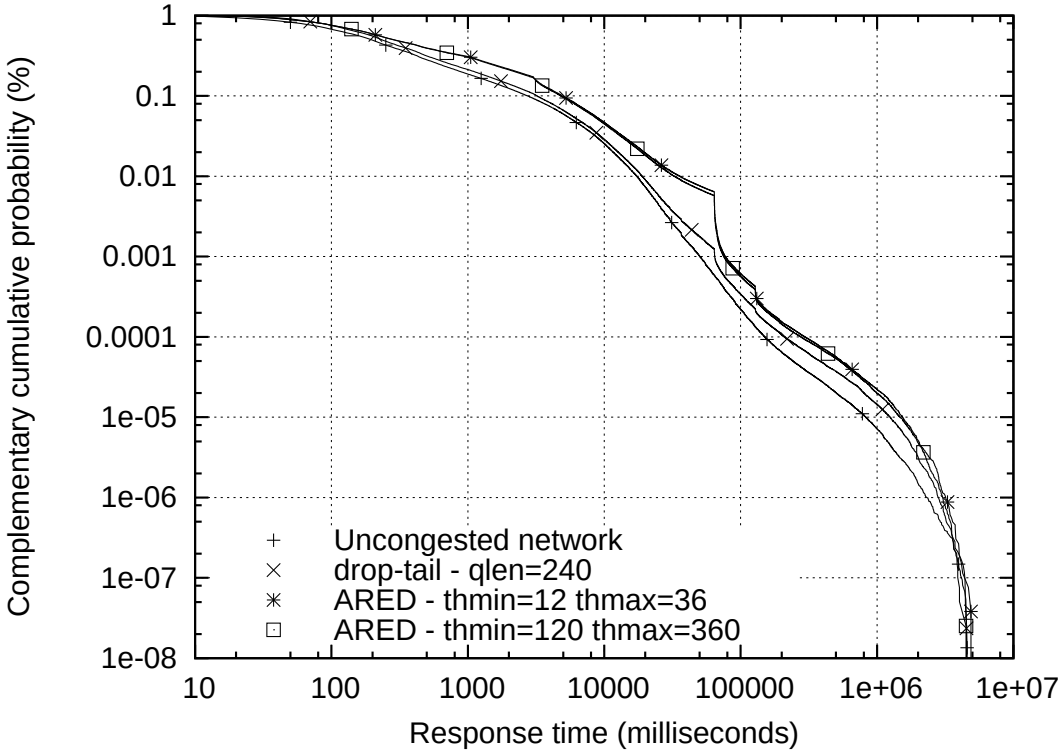

Figure 5.17: ARED packet mode performance at 90% load (CCDF)

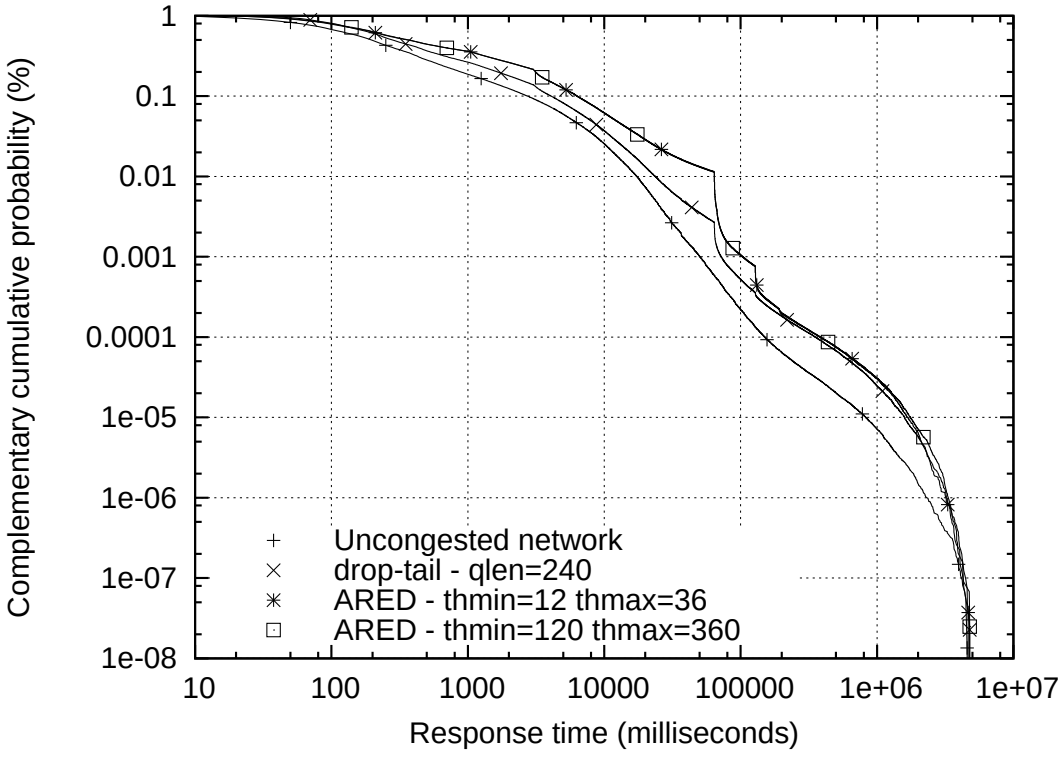

Figure 5.18: ARED packet mode performance at 98% load (CCDF)

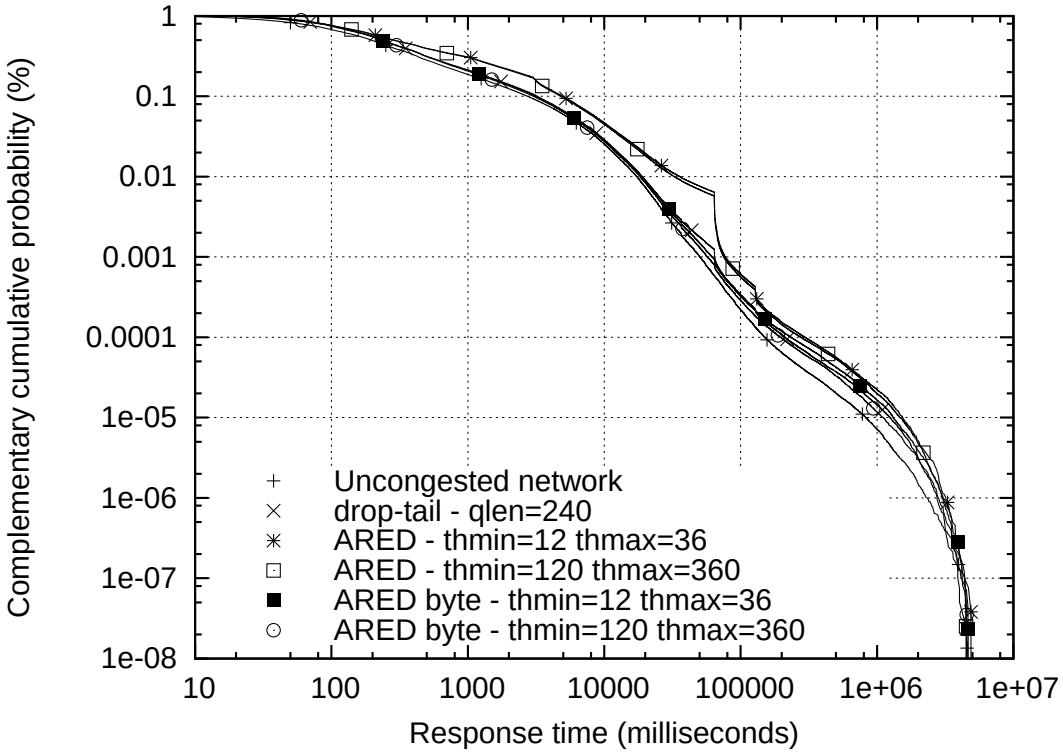

Figure 5.19: ARED byte mode performance at 90% load (CCDF)

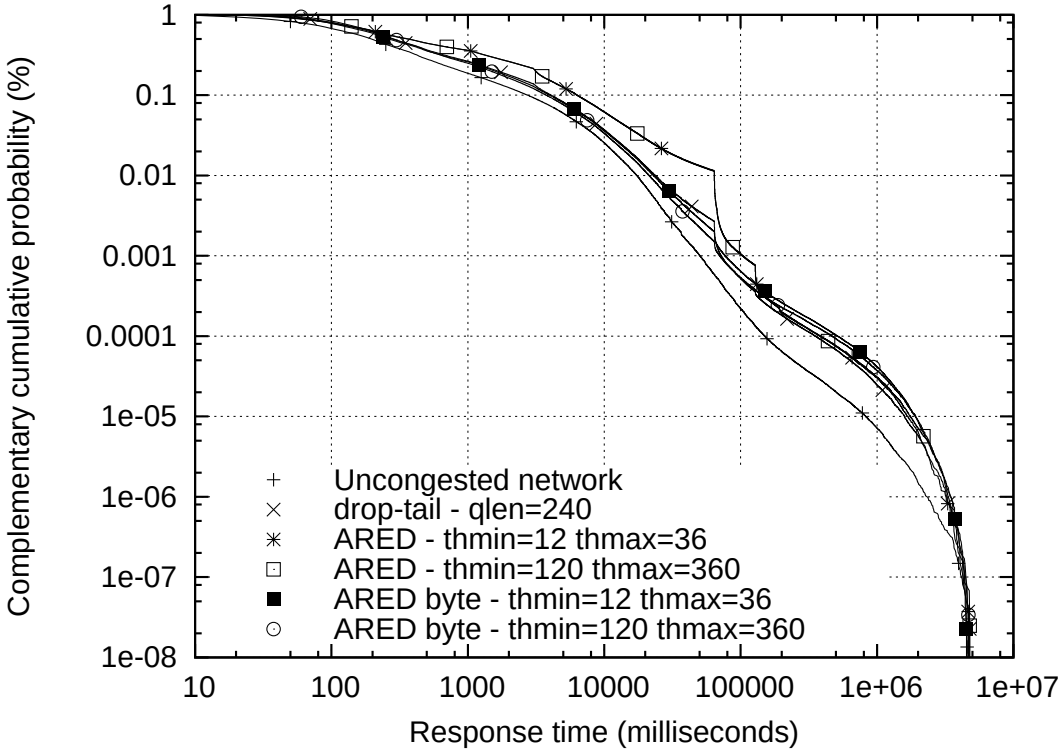

Figure 5.20: ARED byte mode performance at 98% load (CCDF)

5.2.4 Results for LQD

Figures 5.21 and 5.22 show experimental results for LQD at 90% and 98% loads when LQD operated with packet drops. These results were obtained with a queue reference of 24 and 240 packets at these loads.

At 90% load, LQD obtained good performance with both queue references of 24 and 240 packets. The performance of LQD was identical to the performance of drop-tail at this load and closely approximated that of the uncongested network.

As the offered load increased to 98%, the performance for LQD decreased considerably. LQD achieved similar performance with both queue references of 24 and 240 packets and gave about the same performance as drop-tail as this load.

5.2.5 Results for DCN

Figures 5.25 and 5.26 depict experimental results for DCN with a queue reference of 24 and 240 packets at 90% and 98% offered loads when DCN was used with packet drops.

At 90% offered load, DCN obtained similar performance with both queue references of 24 and 240 packets. The performance for DCN was about the same as that of drop-tail and came close to the performance of the uncongested network at this load.

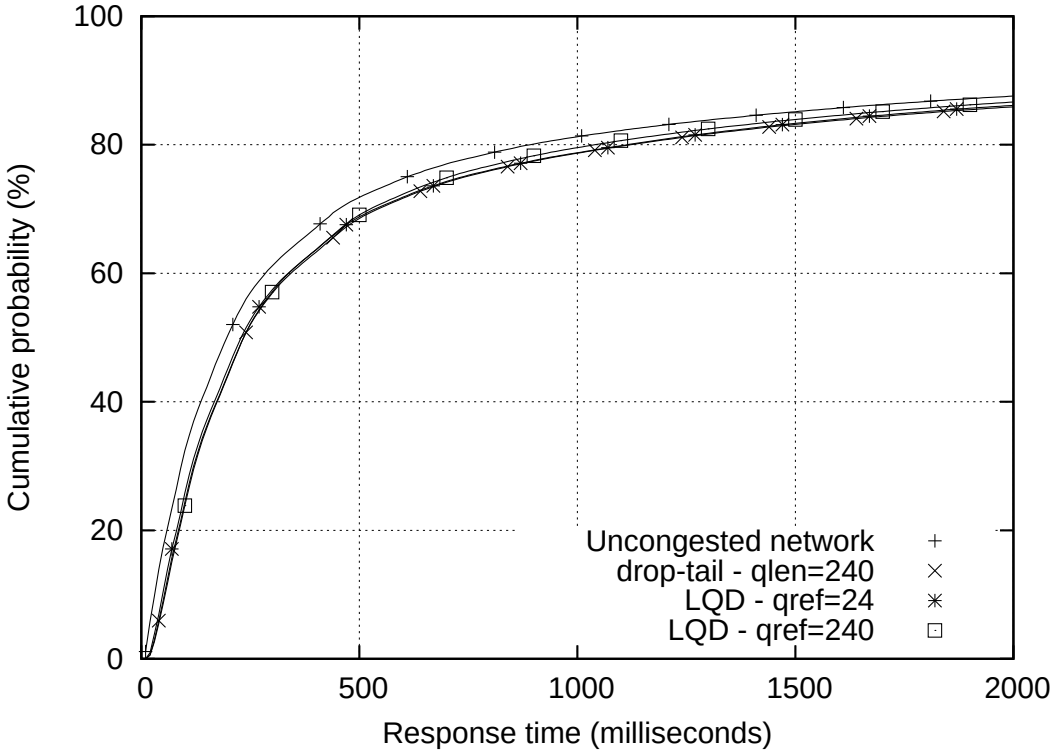

Figure 5.21: LQD performance at 90% load

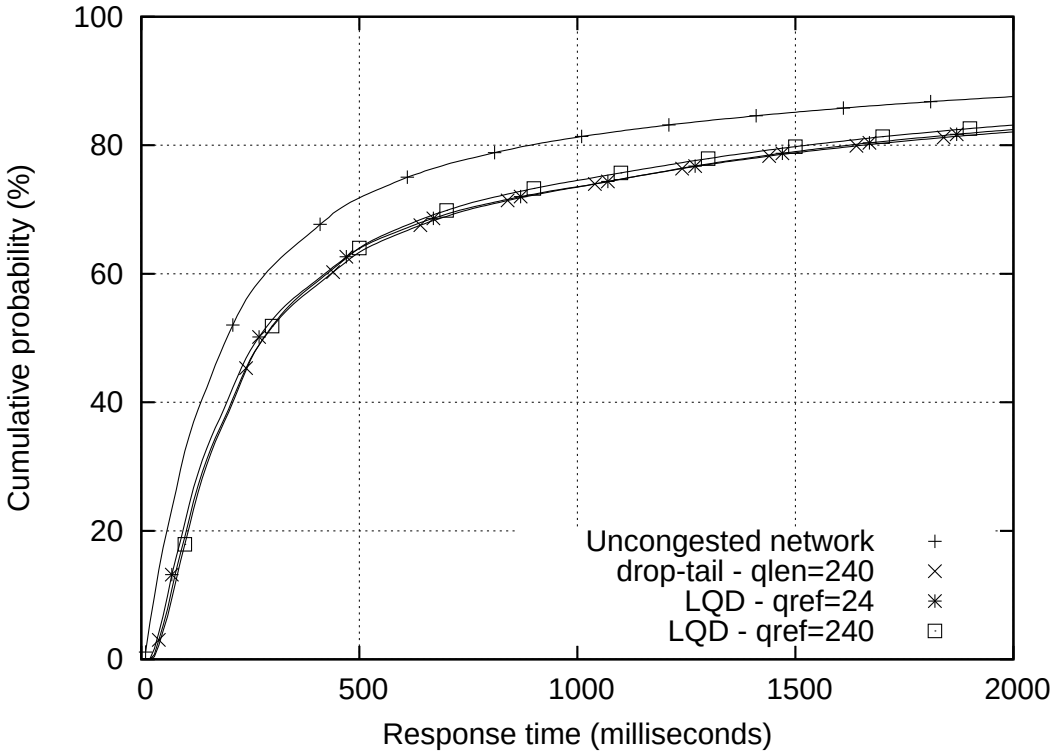

Figure 5.22: LQD performance at 98% load

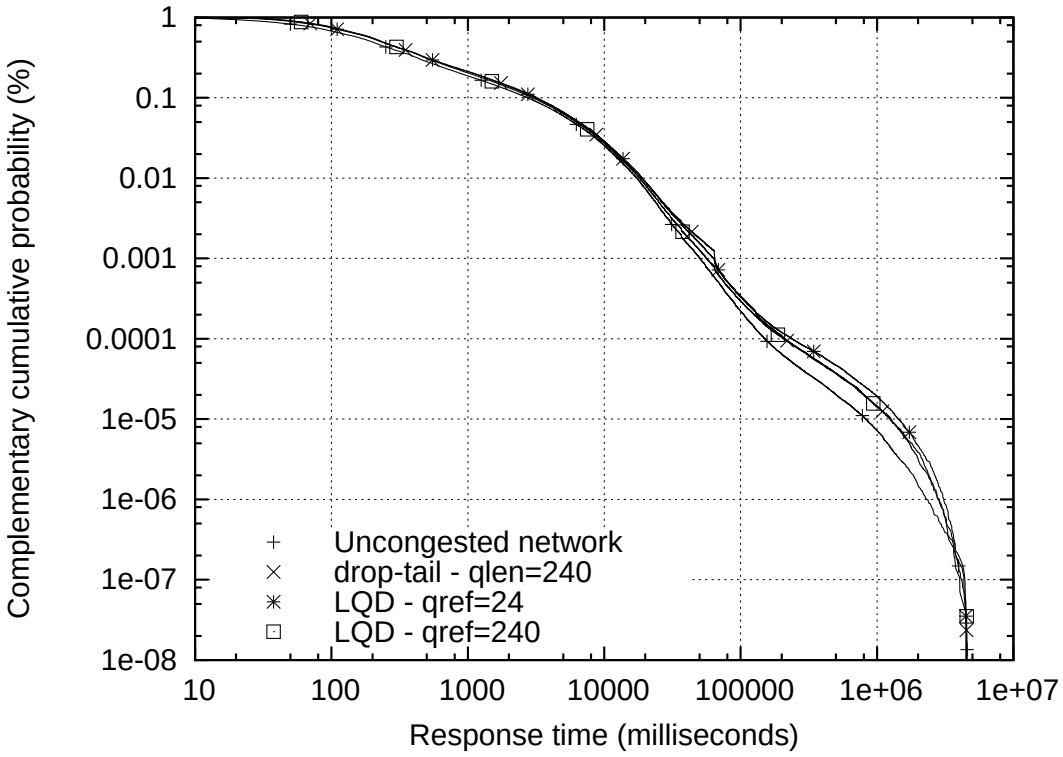

Figure 5.23: LQD performance at 90% load (CCDF)

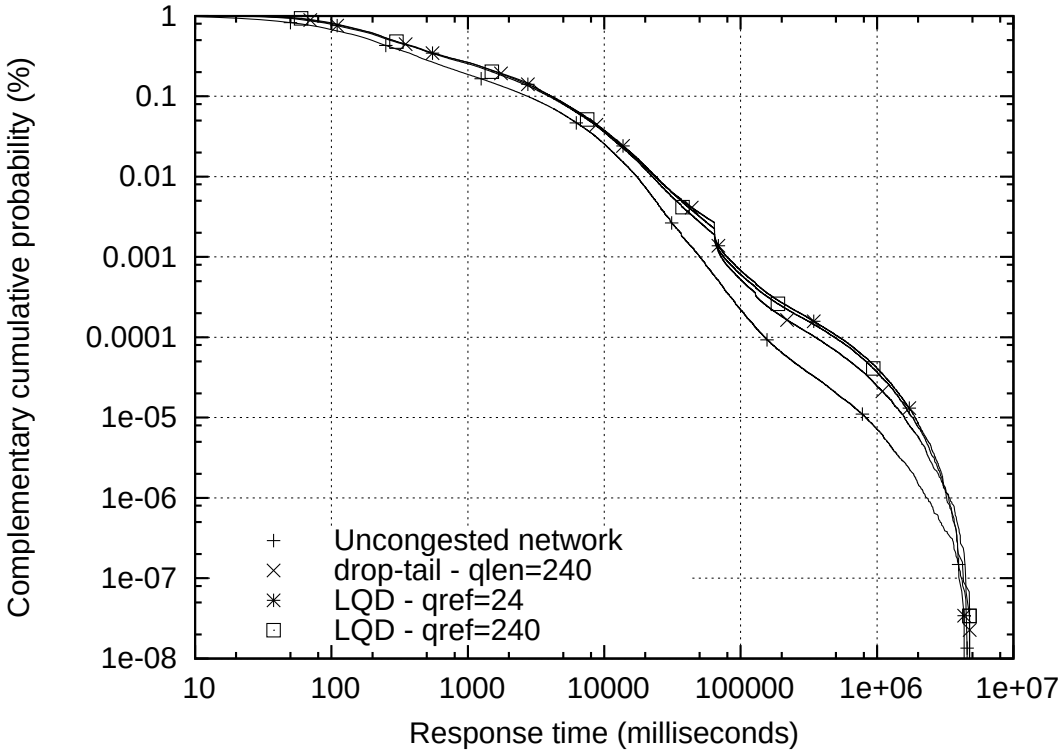

Figure 5.24: LQD performance at 98% load (CCDF)

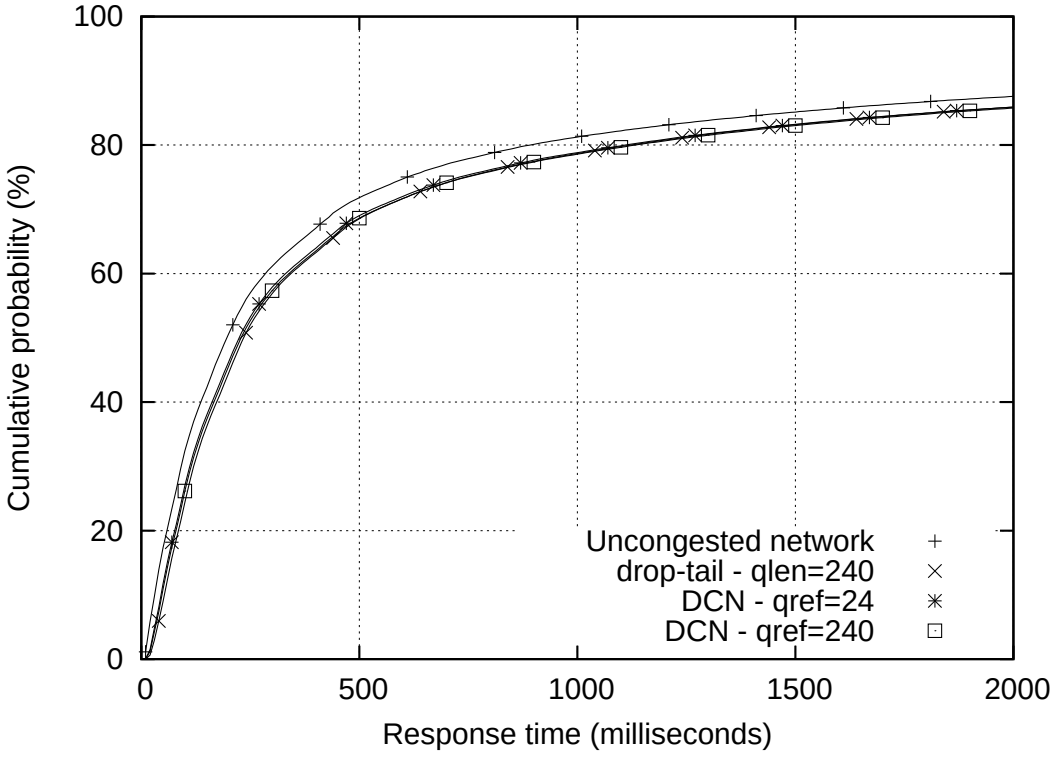

Figure 5.25: DCN performance at 90% load

As the offered load increased to 98%, DCN still delivered good performance with both queue references and outperformed drop-tail. Further, DCN closely approximated the performance of the uncongested network at this high load. This positive result demonstrated the benefits of differential treatment of flows in achieving good application and network performance.

5.3 Results for PI, REM, LQD, DCN, and ARED with ECN

Experimental results for AQM algorithms presented in section 5.2 were obtained when AQM algorithms operated with packet drops. Experiments for AQM algorithms were also performed when ECN was turned on at both routers and end systems to quantify the effects of the ECN signaling protocol.

5.3.1 Results for PI/ECN

Figures 5.29 and 4.23 show experimental results for PI with ECN and without ECN at 90% and 98% loads. These results were obtained when PI operated with a queue reference of 24 and 240 packets.

At 90% offered load, the addition of ECN did not improve the performance of PI since PI

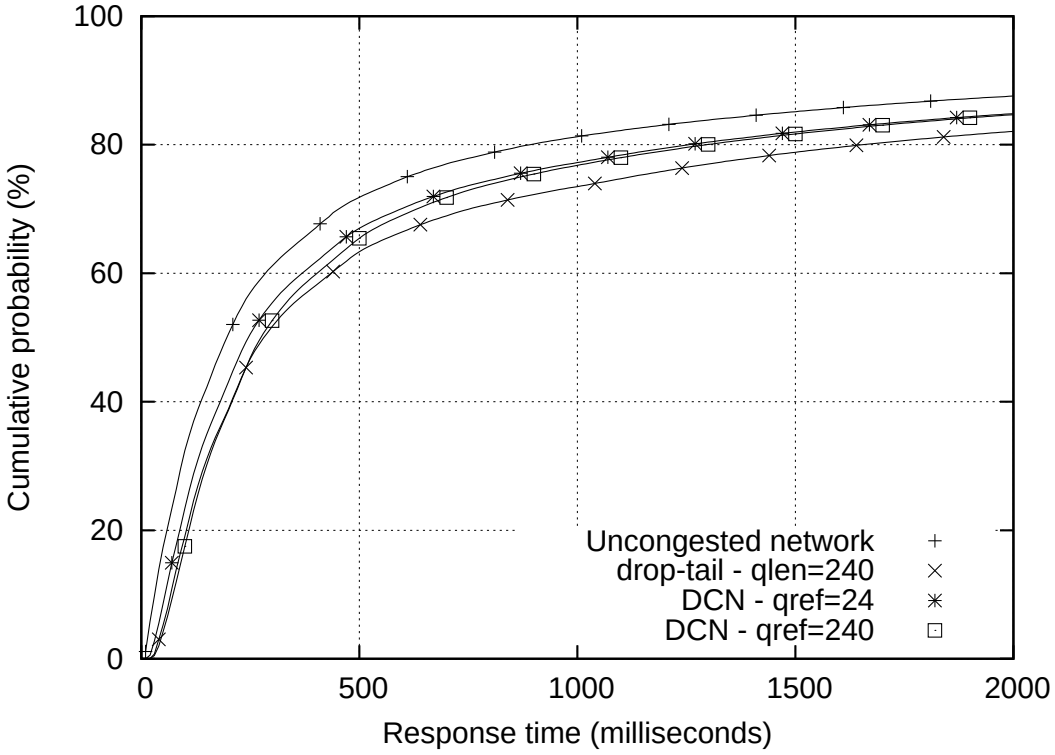

Figure 5.26: DCN performance at 98% load

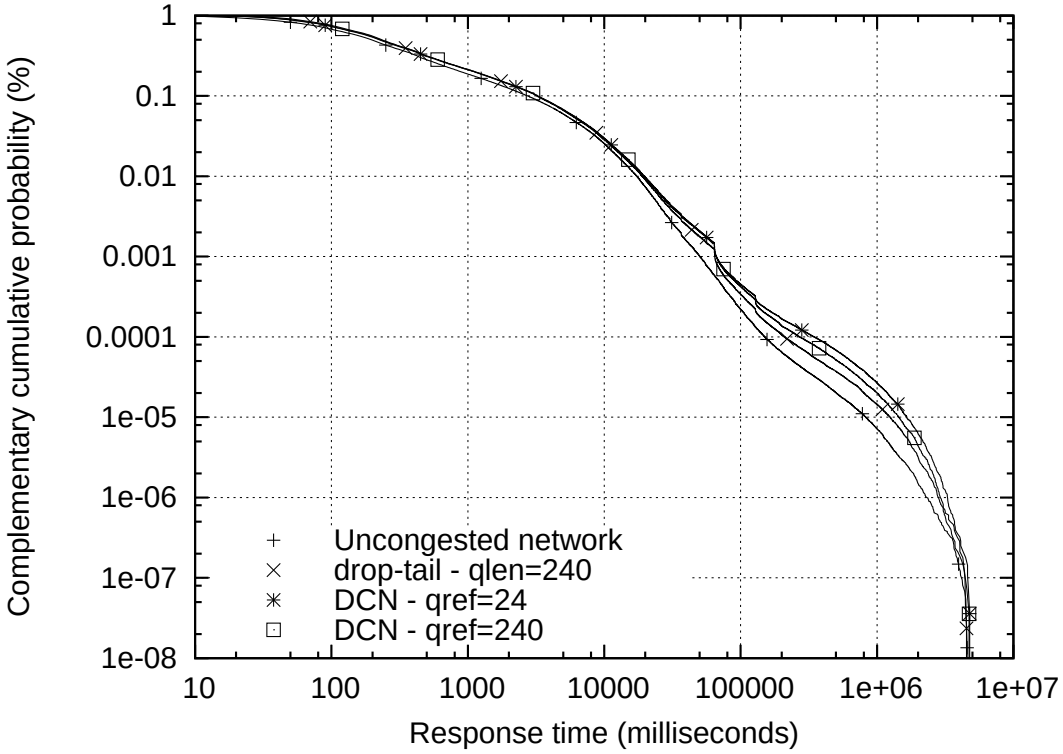

Figure 5.27: DCN performance at 90% load (CCDF)

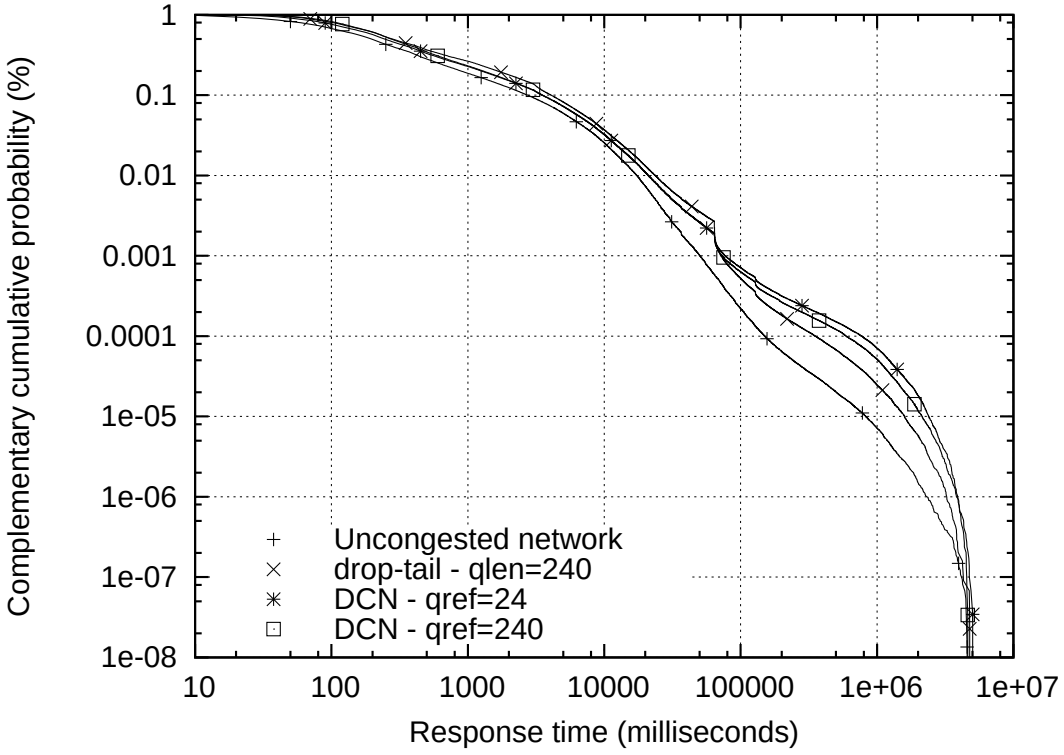

Figure 5.28: DCN performance at 98% load (CCDF)

already obtained very good performance without ECN at this load. With or without ECN, PI with a queue reference of 24 packets gave about the same performance as drop-tail while PI with a queue reference of 240 packets gave slightly worse performance than drop-tail.

At 98% load, PI with a queue reference of 24 packets obtained better performance than with a queue reference of 240 packets when PI operated in ECN mode. Further, PI in ECN mode and with a queue reference of 24 packets outperformed PI with packet drops and drop-tail. The performance for PI in ECN mode and with a queue reference of 24 packets came reasonably close to that of the uncongested network. ECN also increased performance for PI slightly when PI was used with a queue reference of 240 packets.

5.3.2 Results for REM/ECN

Figures 5.33 and 5.34 show the experimental results for REM with and without ECN at 90% and 98% loads. These results were obtained with a queue reference of 24 and 240 packets for REM.

At 90% load, REM did not gain performance improvement when it was used with ECN because REM already obtained very good performance without ECN at this load. With or without ECN, REM obtained about the same performance with both queue references of 24 and 240 packets. The performance for REM was similar to that of drop-tail and came

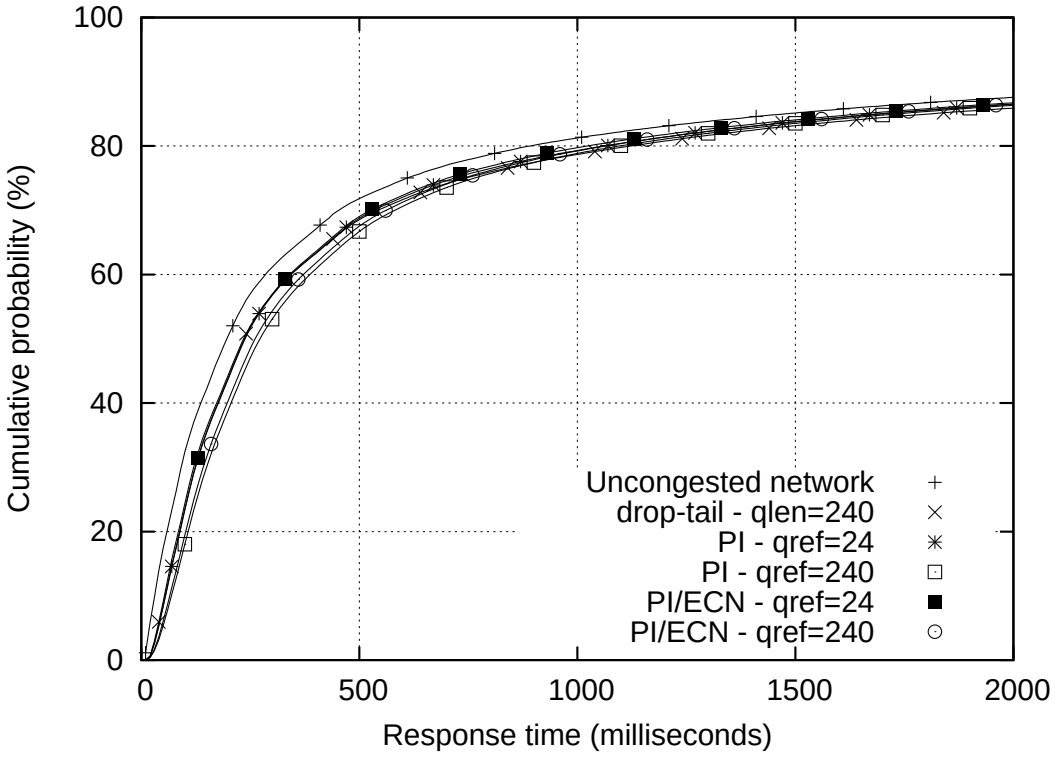

Figure 5.29: PI/ECN performance at 90% load

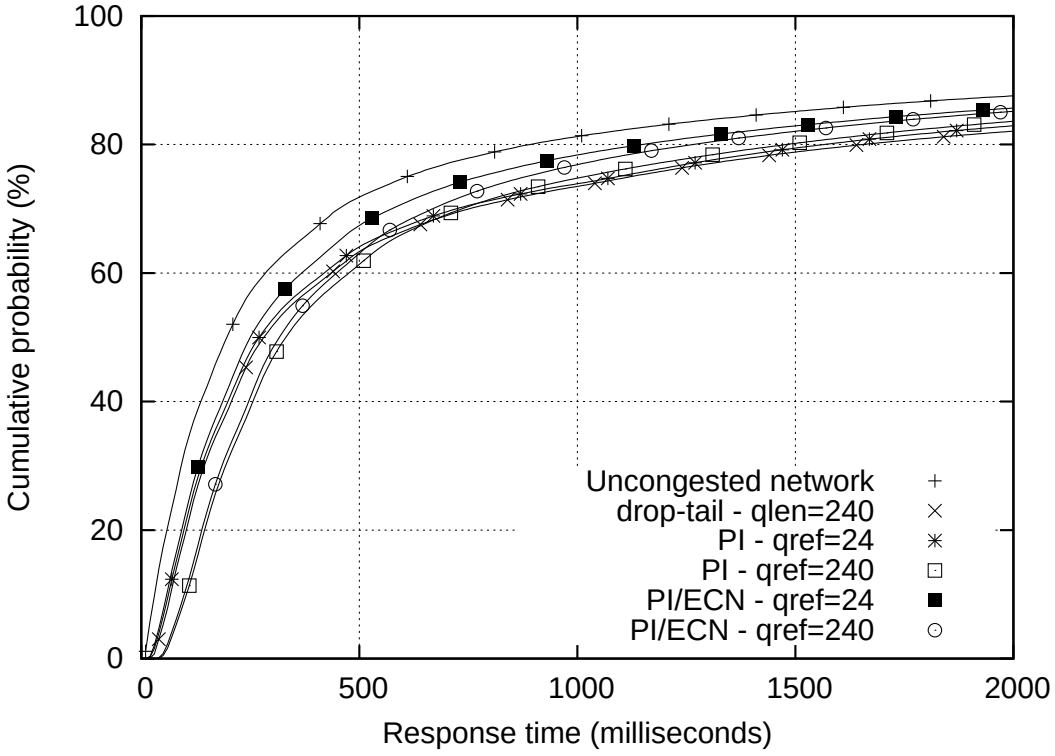

Figure 5.30: PI/ECN performance at 98% load

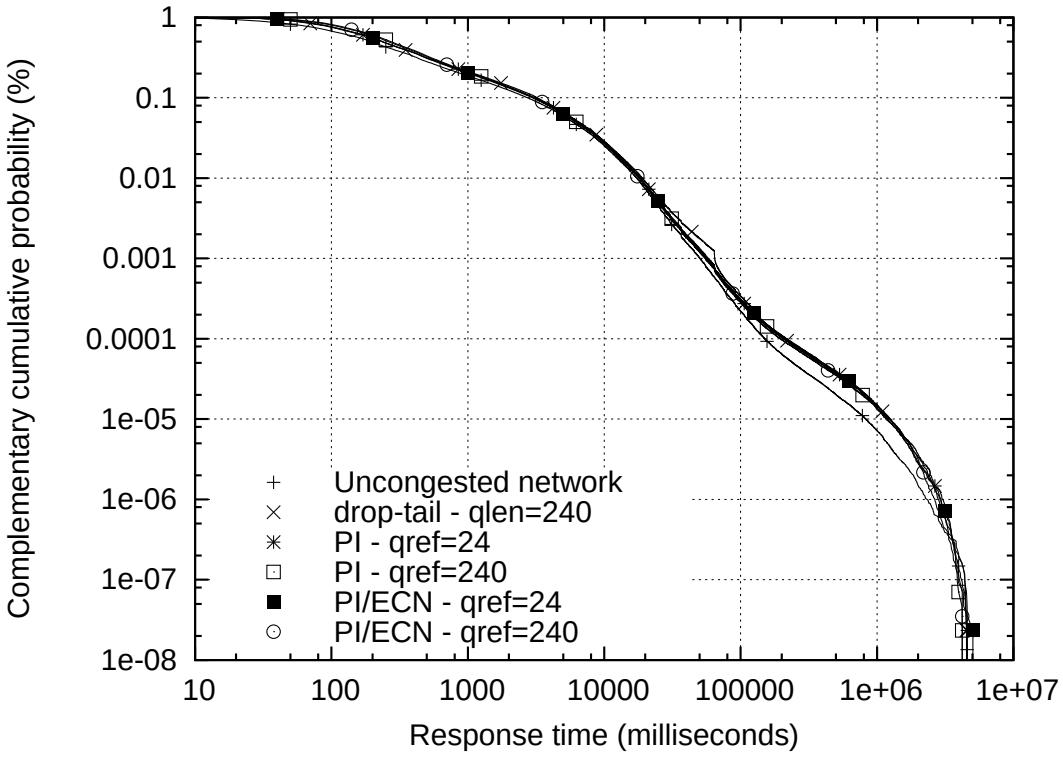

Figure 5.31: PI/ECN performance at 90% load (CCDF)

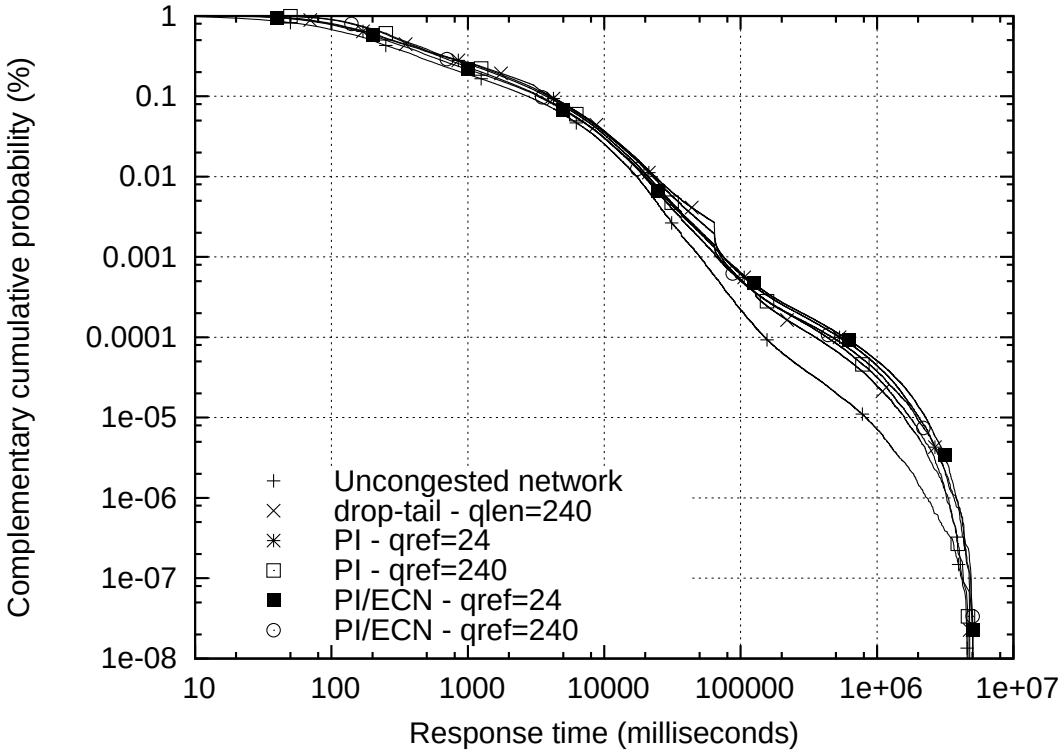

Figure 5.32: PI/ECN performance at 98% load (CCDF)

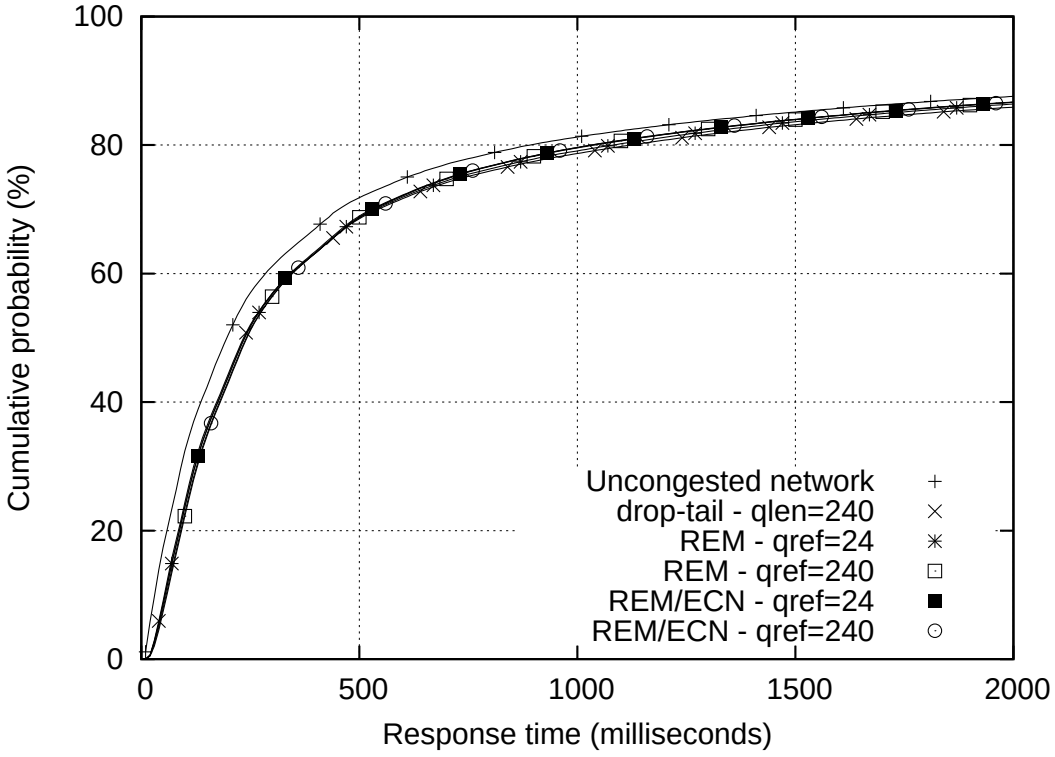

Figure 5.33: REM/ECN performance at 90% load

close to the performance of the uncongested network at this load.

At 98% offered load, the performance for REM with both queue references was increased considerably when REM was used with ECN. When operated in ECN mode, REM with both queue references outperformed drop-tail and closely approximated the performance of the uncongested network. With ECN, REM gave slightly better performance with a queue reference of 24 packets than with a queue reference of 240 packets.

5.3.3 Results for ARED/ECN

Figures 5.37, 5.38, 5.39, and 5.40 show experimental results for ARED with and without ECN at 80%, 90%, 98%, and 105% loads. Experiments were performed for ARED with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) to achieve a target queue reference of 24 and 240 packets. Further, results for ARED with ECN were obtained with the original ARED algorithm and the modified algorithm ARED "new gentle".

Figures 5.37 and 5.38 show that when the original ARED algorithm was used, the ECN signaling protocol did not improve the poor performance for ARED with both parameter settings at all. Even with ECN, the original ARED algorithm with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets)

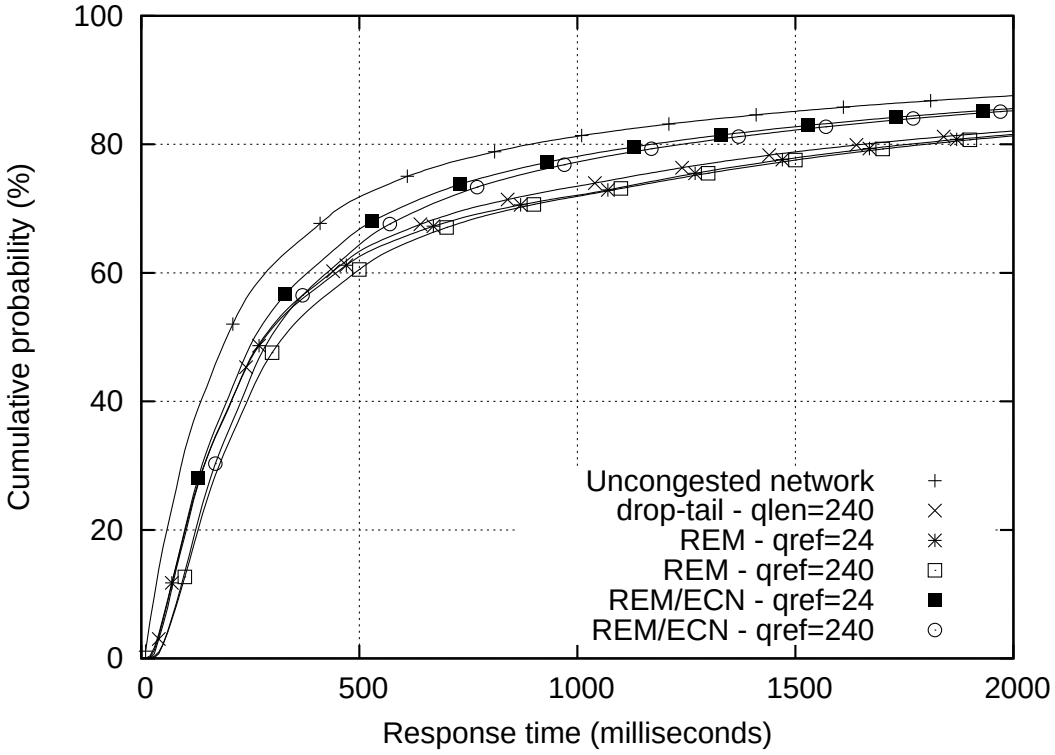

Figure 5.34: REM/ECN performance at 98% load

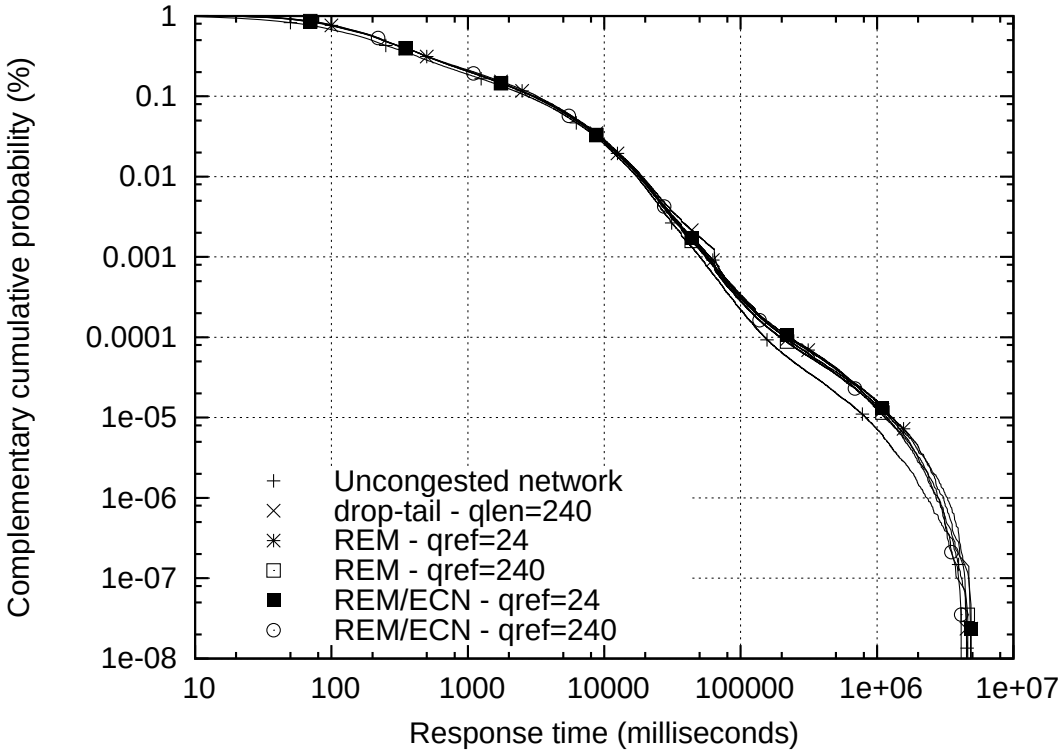

Figure 5.35: REM/ECN performance at 90% load (CCDF)

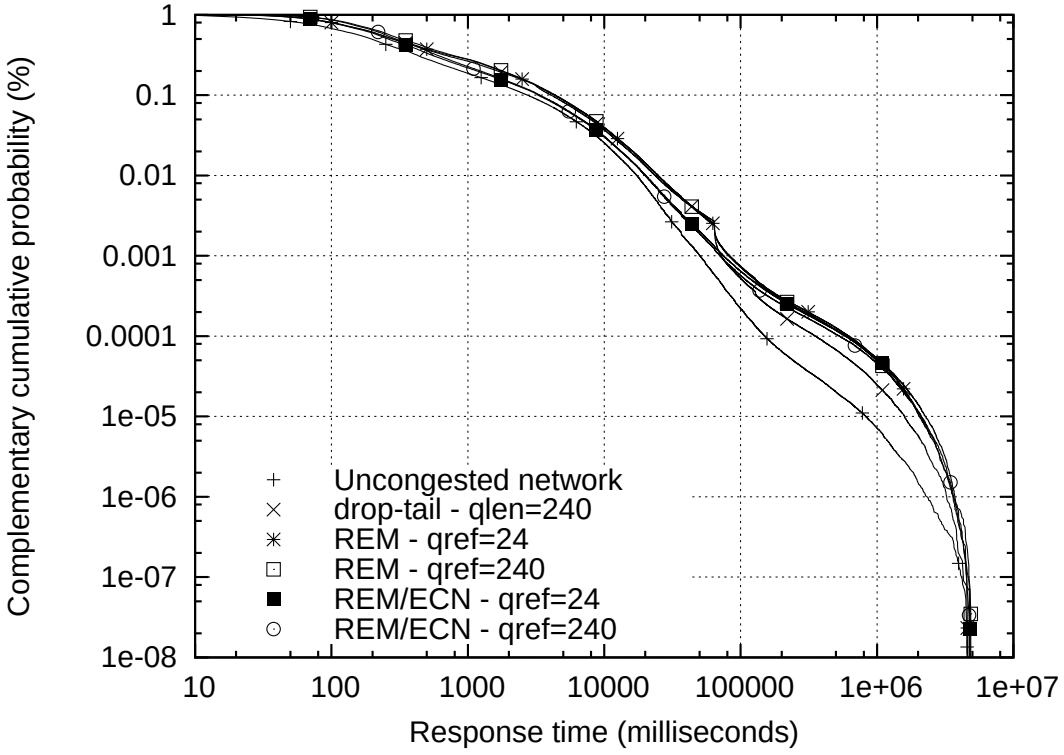

Figure 5.36: REM/ECN performance at 98% load (CCDF)

underperformed drop-tail considerably at 90% and 98% loads.

Figures 5.39and 5.40 show that when ECN was used with the ARED "new gentle" algorithm, the performance for ARED with both parameter settings was improved considerably at 90% and 98% loads. The ARED "new gentle" algorithm obtained similar performance with both parameter settings and delivered about the same performance as drop-tail at 90% and 98% loads. Further, ARED "new gentle" significantly outperformed the original ARED algorithm with ECN at these loads. The performance for ARED "new gentle" also came relatively close to that of the uncongested network, especially at 90% offered load.

5.3.4 Results for LQD/ECN

Figures 5.45 and 5.46 show experimental results for LQD with ECN and without at 90% and 98% loads. These results were obtained LQD with a queue reference of 24 and 240 packets.

At 90% offered load, LQD delivered identical performance for both queue references of 24 and 240 packets when it was operated in ECN mode. Figure 5.45 shows that the performance for LQD with and without ECN is indistinguishable from that of drop-tail and came close to the performance of the uncongested network.

As the offered load increased to 98%, LQD obtained equally good performance with

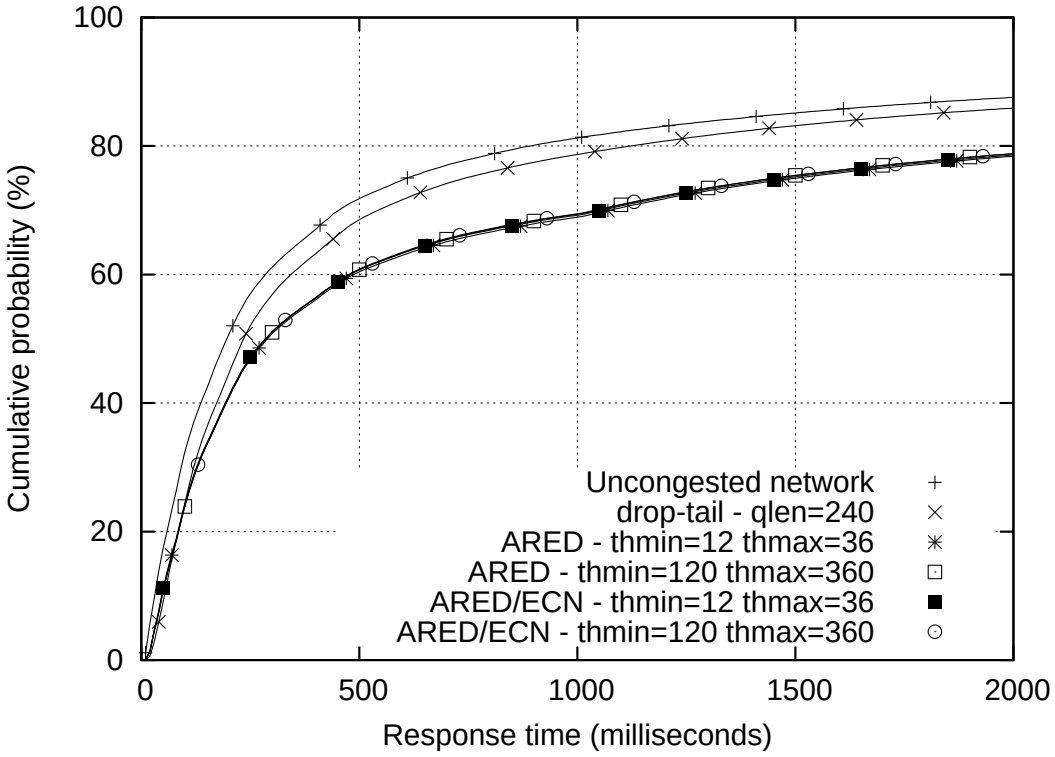

Figure 5.37: ARED/ECN performance at 90% load

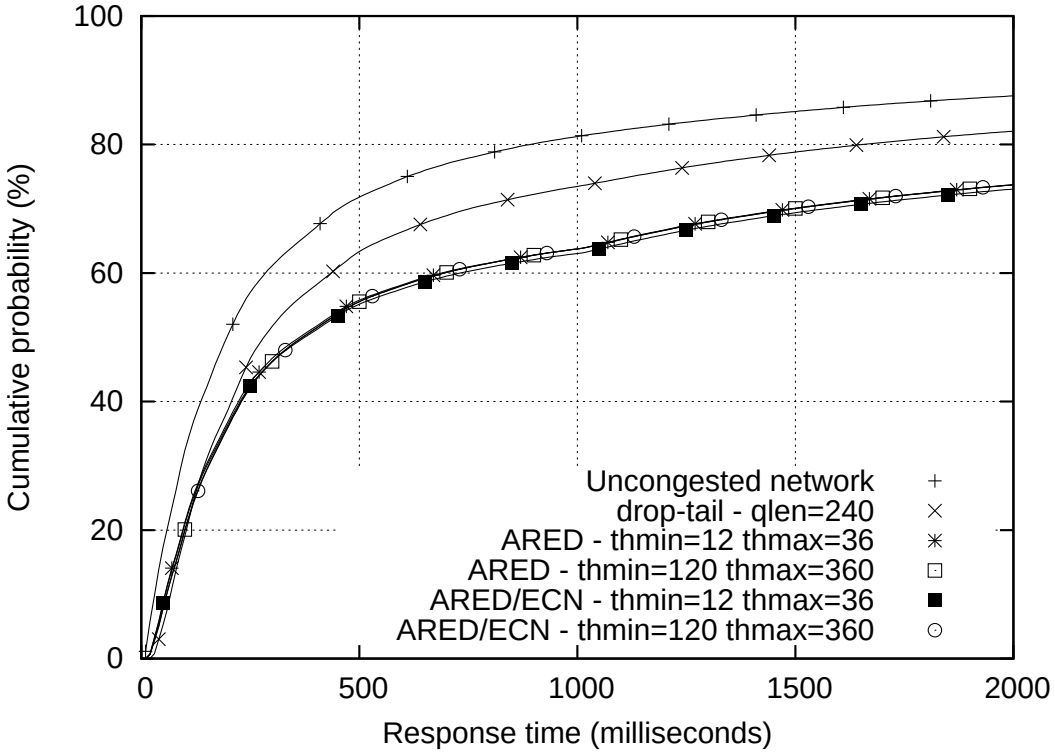

Figure 5.38: ARED/ECN performance at 98% load

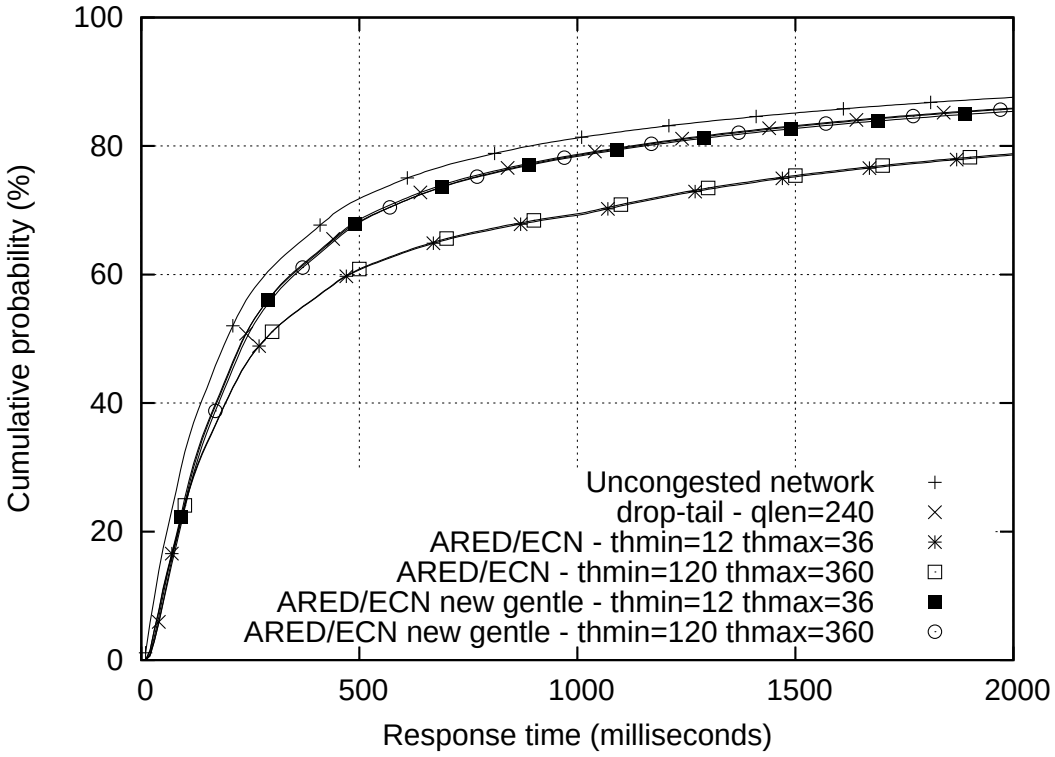

Figure 5.39: ARED/ECN new gentle performance at 90% load

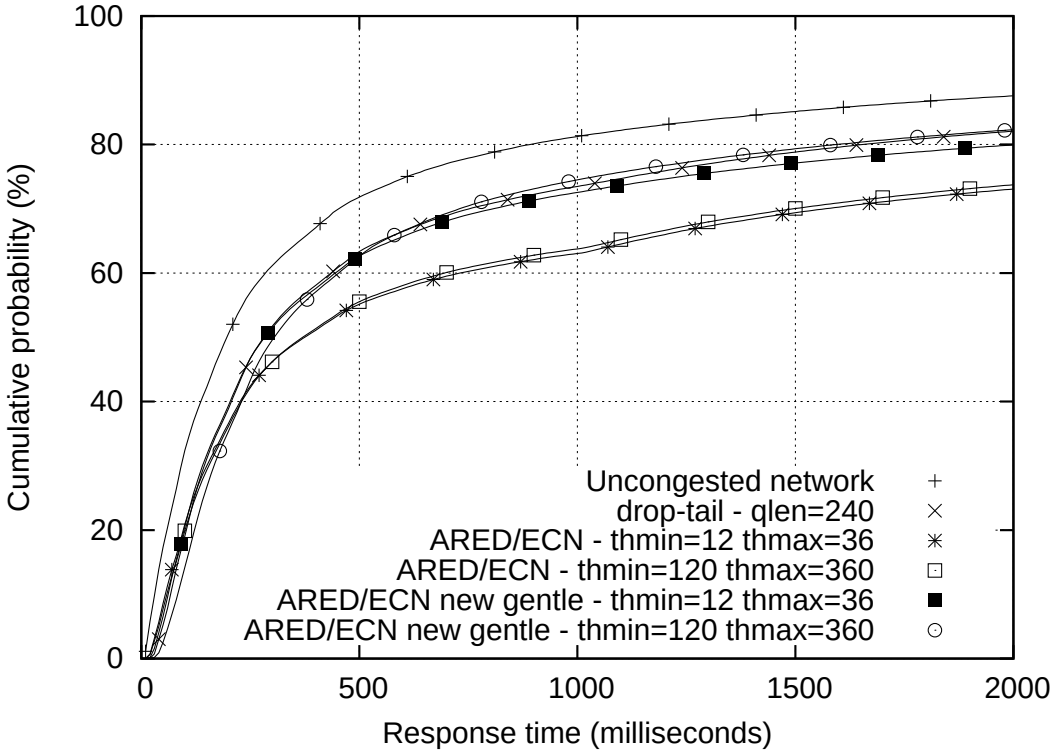

Figure 5.40: ARED/ECN new gentle performance at 98% load

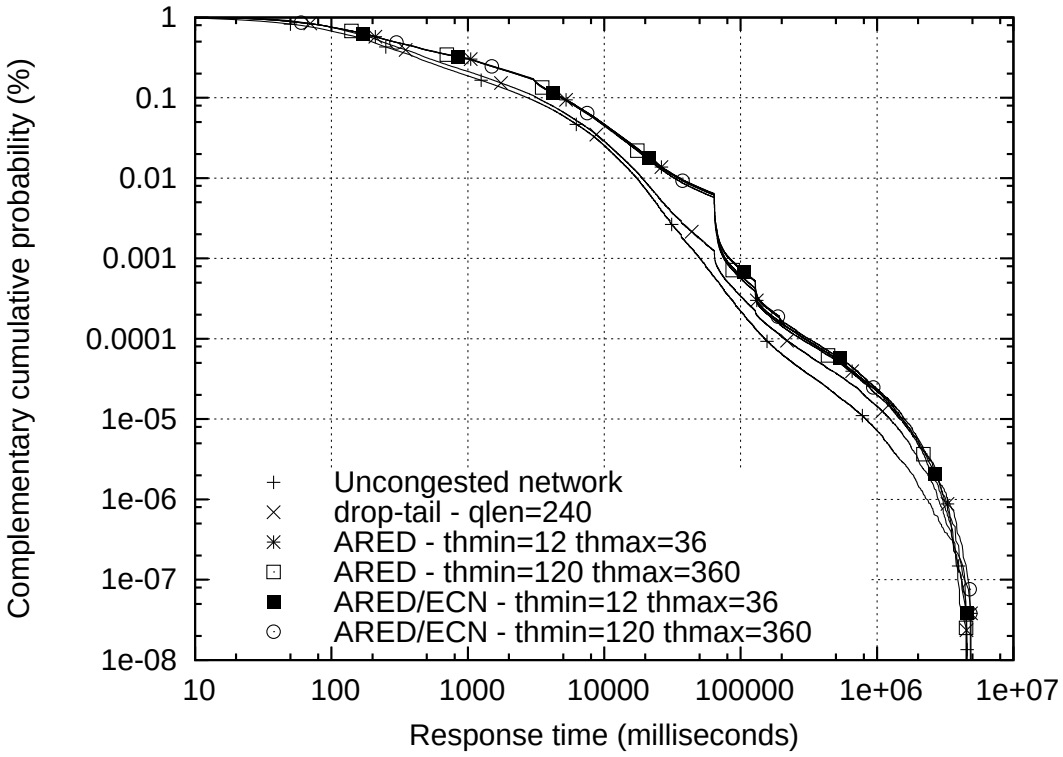

Figure 5.41: ARED/ECN performance at 90% load (CCDF)

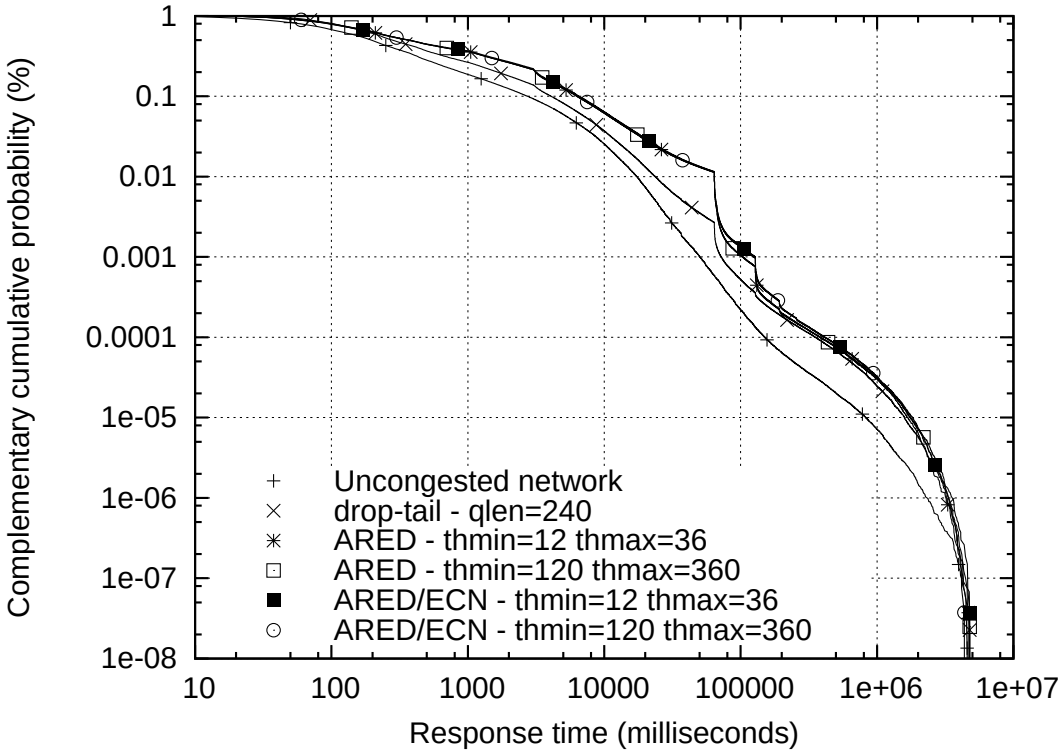

Figure 5.42: ARED/ECN performance at 98% load (CCDF)

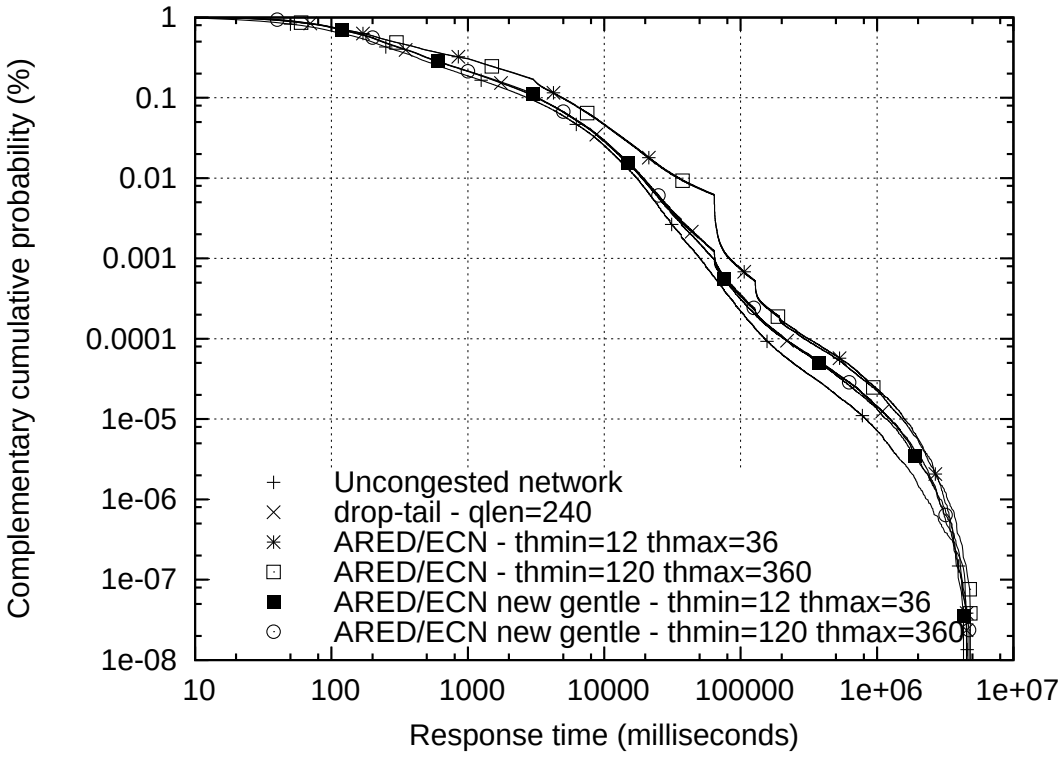

Figure 5.43: ARED/ECN new gentle performance at 90% load (CCDF)

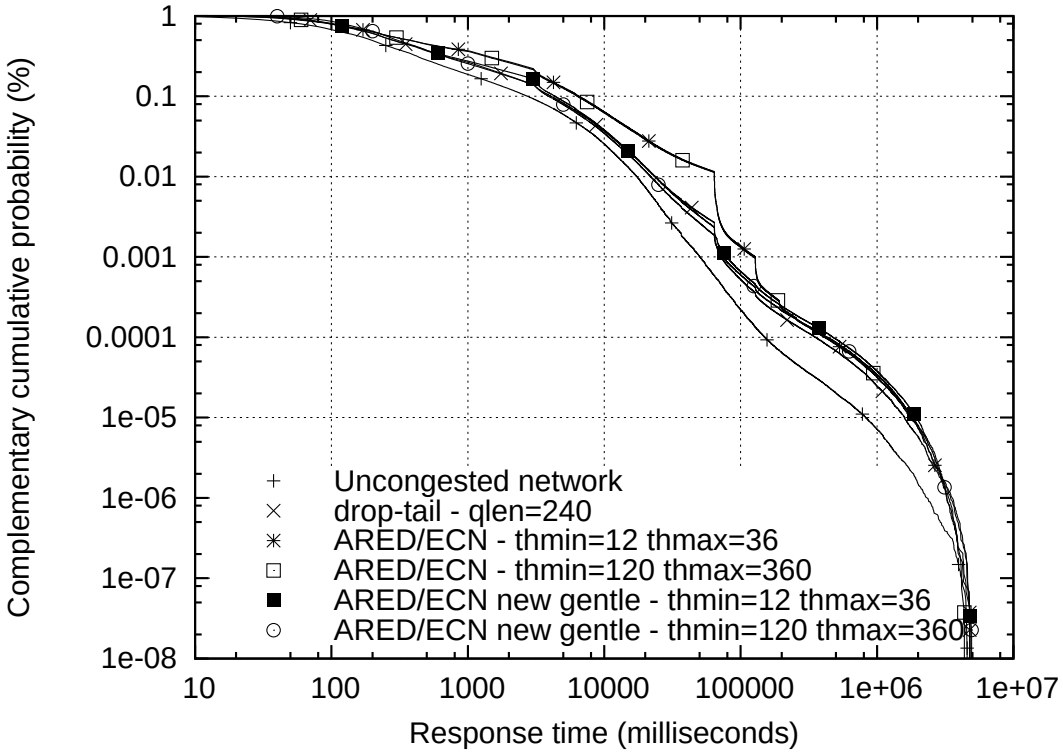

Figure 5.44: ARED/ECN new gentle performance at 98% load (CCDF)

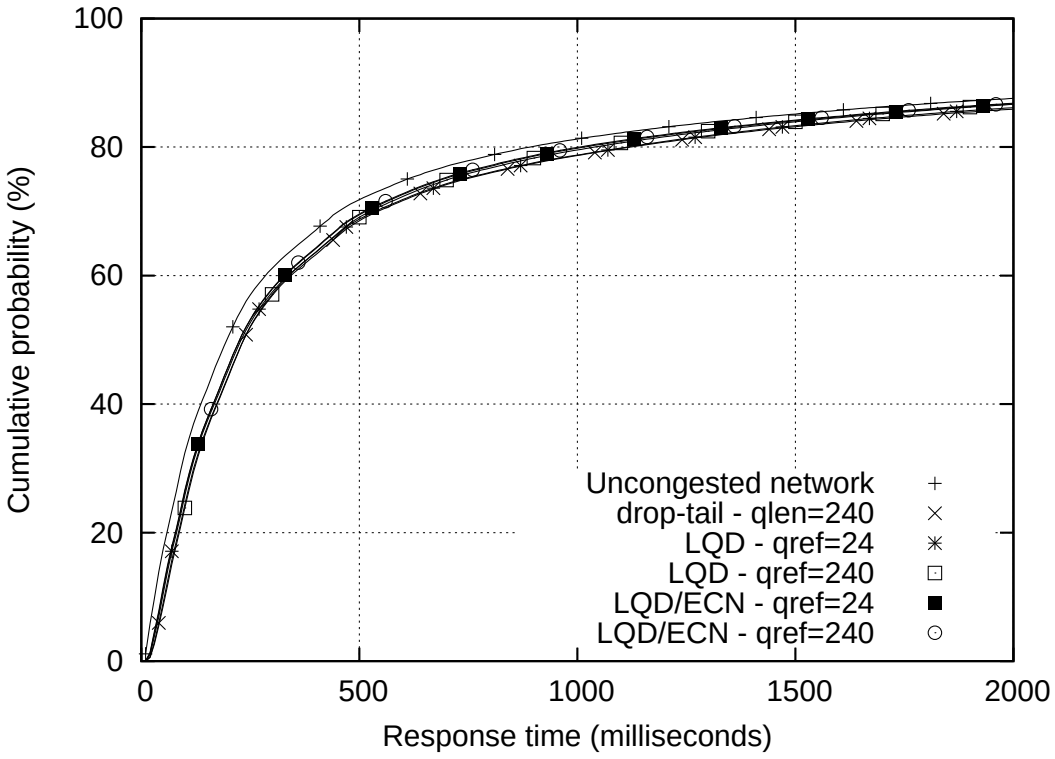

Figure 5.45: LQD/ECN performance at 90% load

both queue references when it was used with ECN. LQD obtained significant performance improvement with ECN and outperformed drop-tail at this load. Further, LQD with ECN closely approximated the performance of the uncongested network at this high load.

5.3.5 Results for DCN/ECN

Figures 5.49 and 5.50 show experimental results for DCN with and without ECN at 90% and 98% offered loads. The results were obtained for DCN with a queue reference of 24 and 240 packets.

At 90% load, DCN did not benefit from the addition of ECN. This was because DCN already provided very good performance with both queue references when it was used without ECN and there was virtually no room for improvement at this load. With and without ECN, DCN delivered performance that was almost comparable with the performance of the uncongested network. Of note is the fact that drop-tail also gave very good performance that was indistinguishable from the performance of DCN at this load.

As the offered load increased to 98%, DCN outperformed drop-tail and still gave identical performance when it was used with and without ECN. This result demonstrated the benefits of differential treatment of flows and provided a proof of concept that AQM algorithms do not need ECN to deliver good performance.

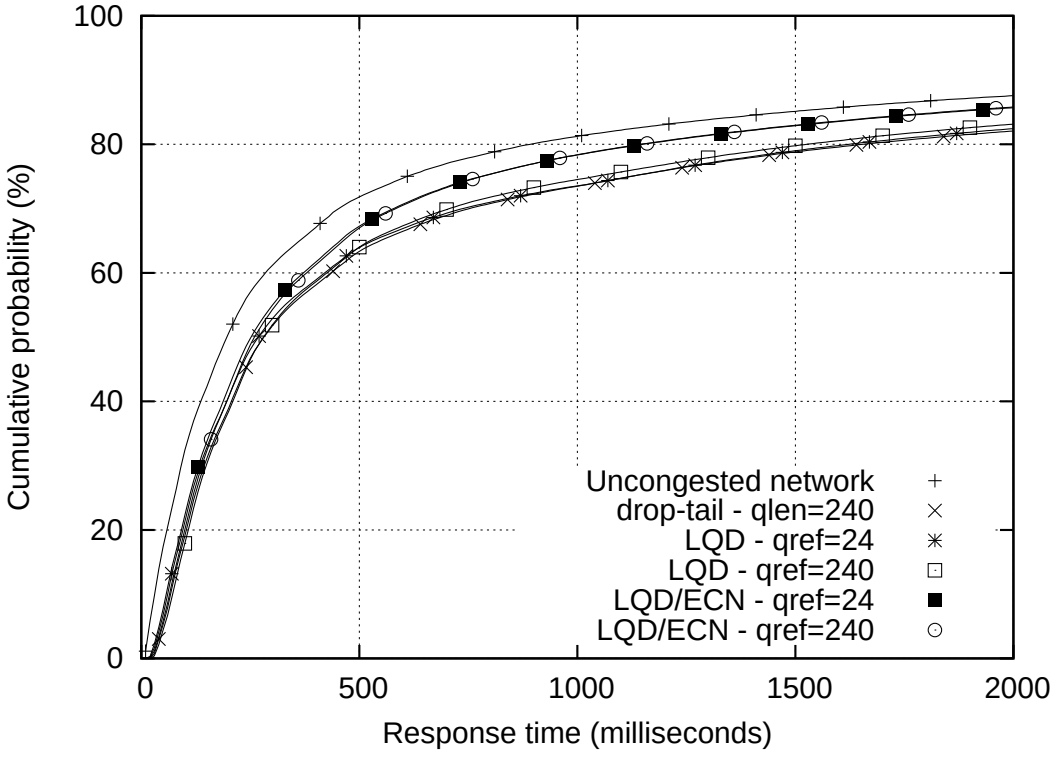

Figure 5.46: LQD/ECN performance at 98% load

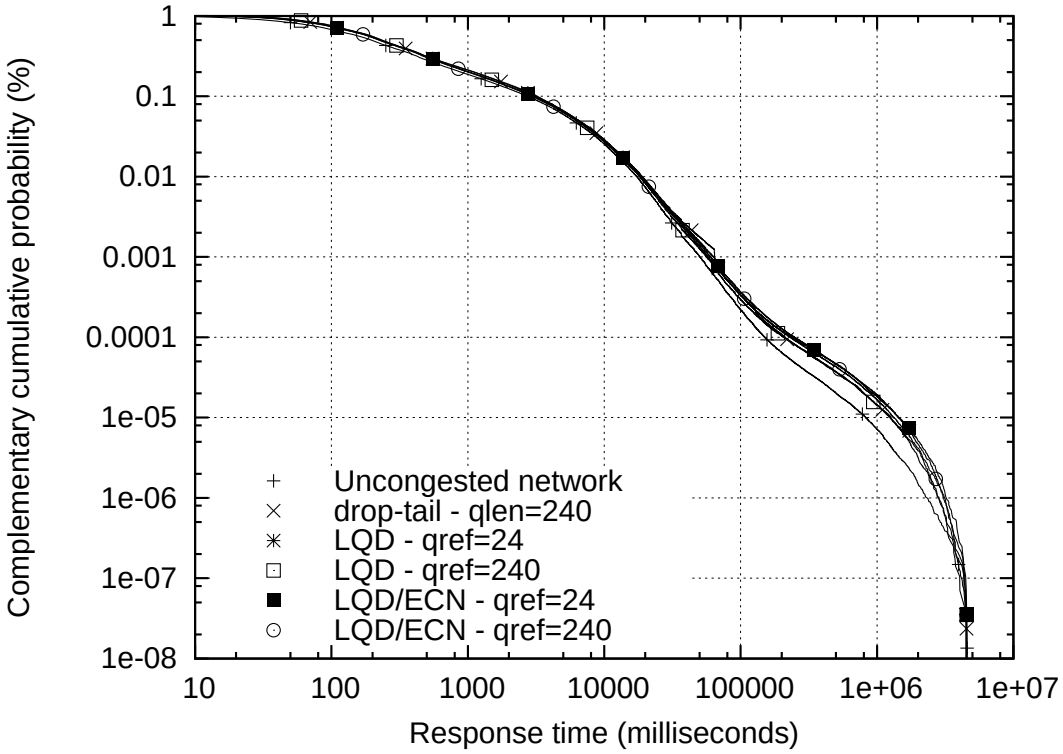

Figure 5.47: LQD/ECN performance at 90% load (CCDF)

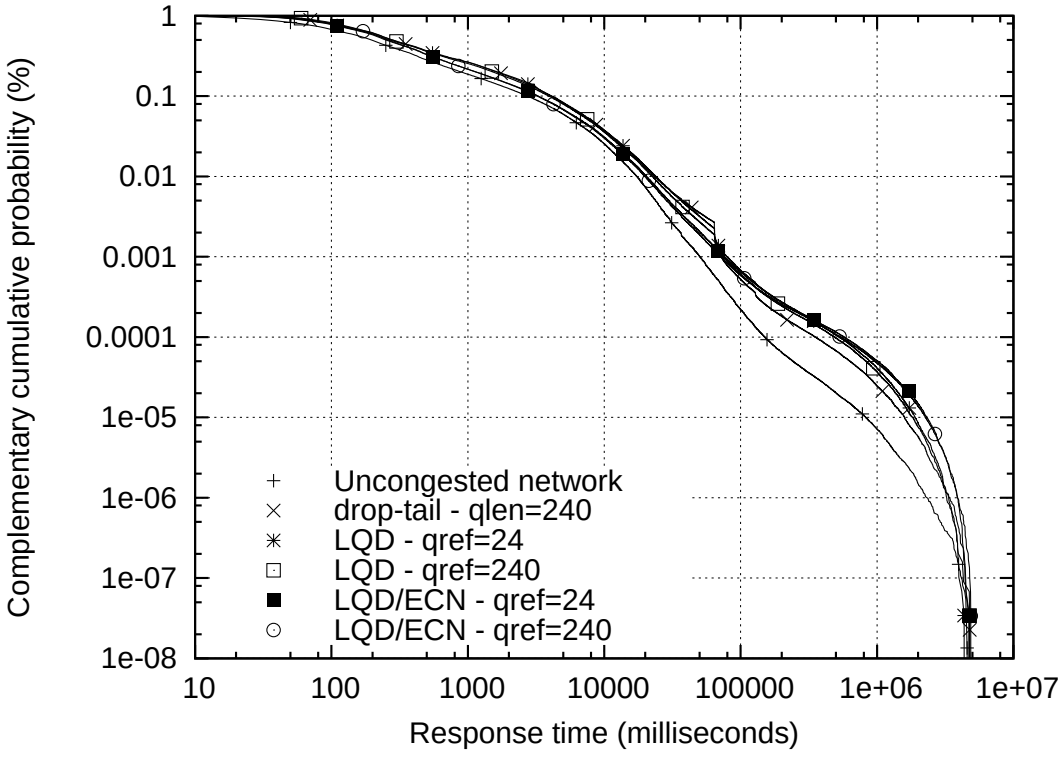

Figure 5.48: LQD/ECN performance at 98% load (CCDF)

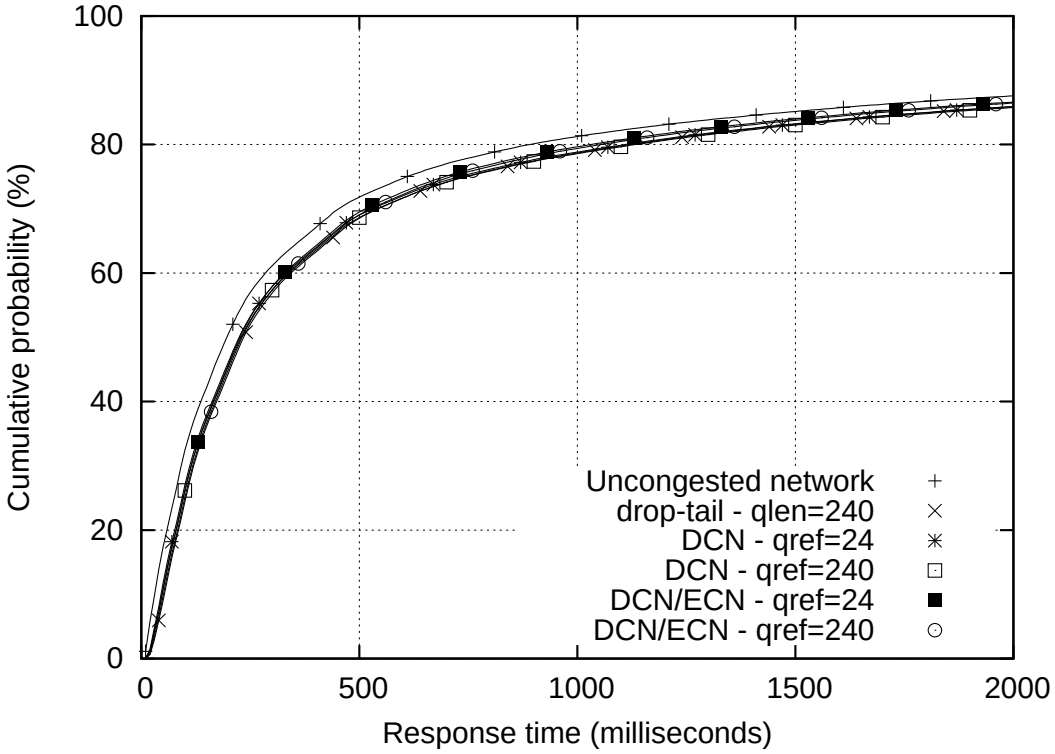

Figure 5.49: DCN/ECN performance at 90% load

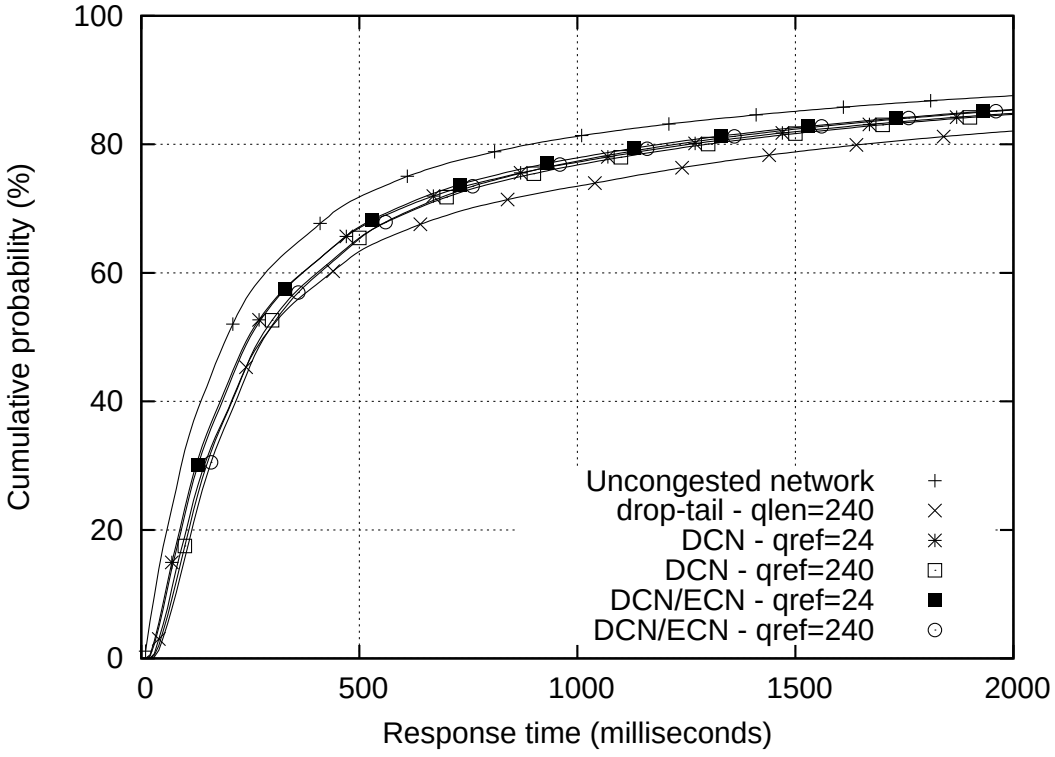

Figure 5.50: DCN/ECN performance at 98% load

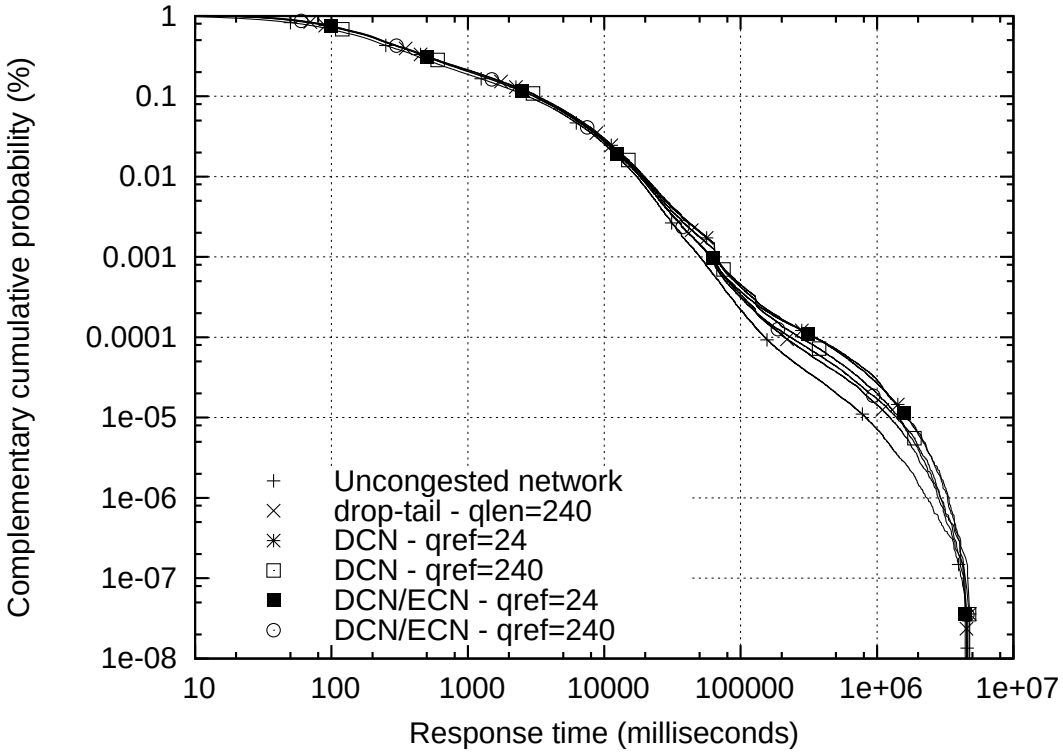

Figure 5.51: DCN/ECN performance at 90% load (CCDF)

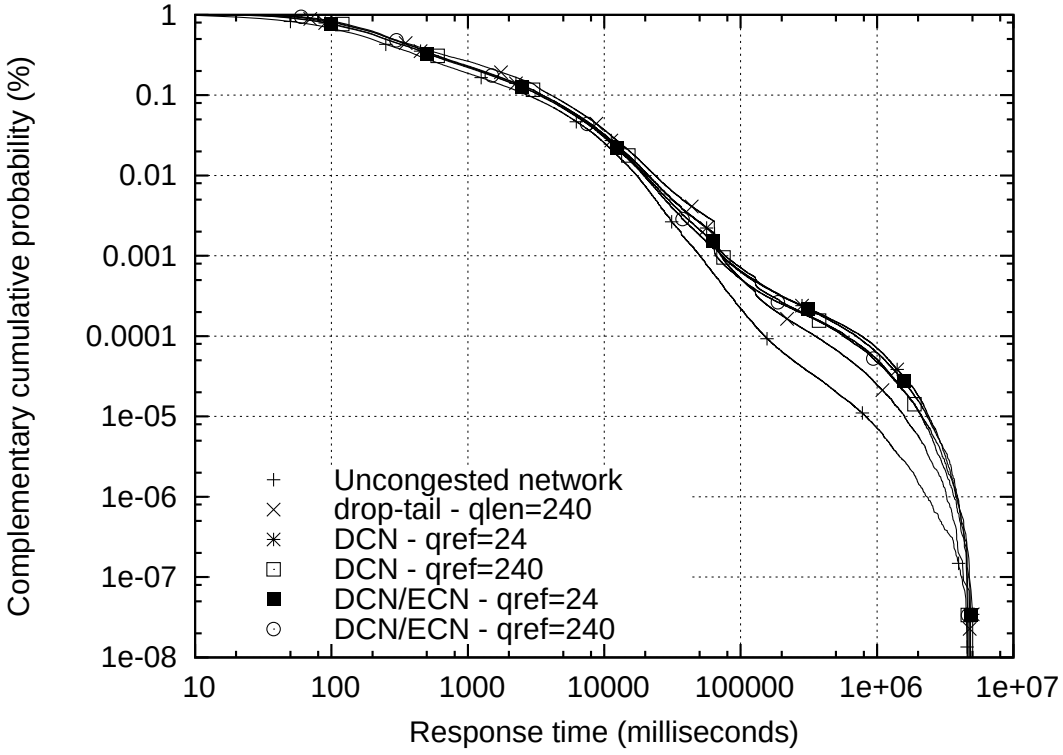

Figure 5.52: DCN/ECN performance at 98% load (CCDF)

5.4 Comparison of All Results

Figures 5.53 and 5.54 provide a comparison of PI, REM, ARED, and DCN at 90% and 98% loads. Experimental results for these AQM algorithms were obtained with the best parameter settings for each of them. Results for DCN was shown when DCN was not used with ECN to demonstrate the potential benefits of differential treatment of flows. For comparison purpose, Figures 5.53 and 5.54 also show the performance of drop-tail and of the uncongested network.

At 90% load, drop-tail and all AQM algorithms gave very similar performance that was almost competitive to the performance of the uncongested network. Thus, it appears that for network environments with a general RTT distribution as shown in Figures 3.6 and 3.7, Internet Service Providers could operate their network at an offered load as high as 90% without risking noticeable performance degradation and customer's dissatisfaction.

At 98% offered load, drop-tail and all AQM algorithms suffered considerable performance degradation but PI/ECN, REM/ECN, and DCN still delivered good performance. Further, they outperformed drop-tail at this load. DCN again demonstrated the benefits of differential treatment of flows since it gave good performance at this high load without relying on the ECN signaling protocol. The ARED "new gentle" algorithm in ECN mode

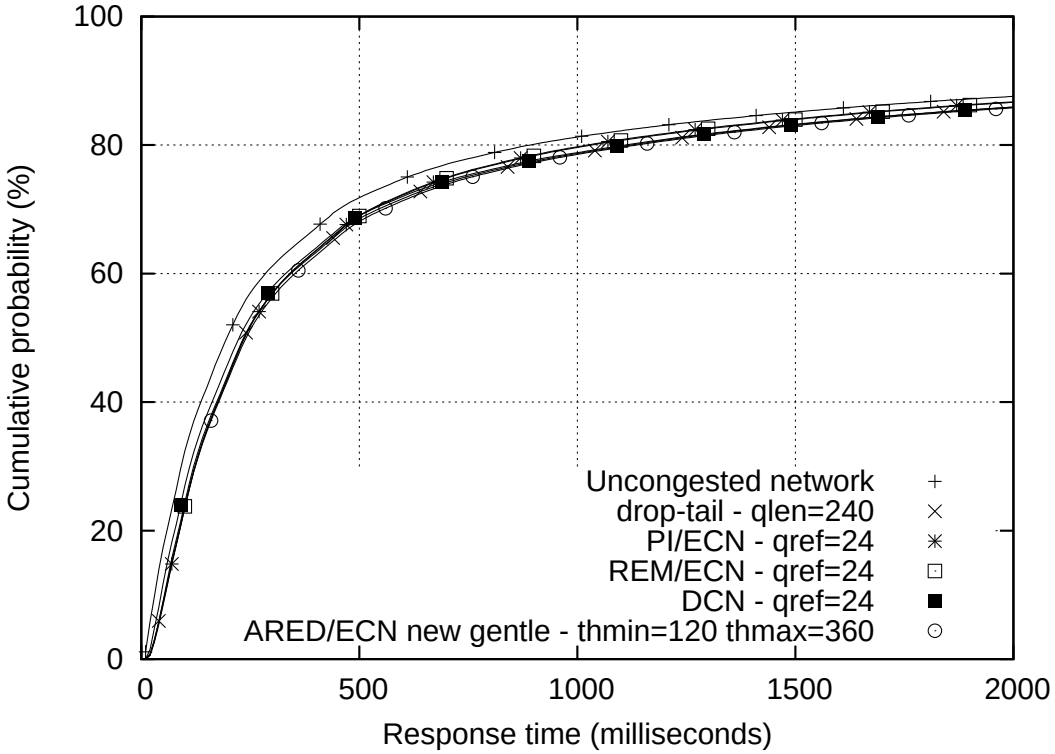

Figure 5.53: Comparison of all AQM algorithms at 90% load

gave comparable performance as drop-tail but underperformed other AQM algorithms.

5.5 Summary

This Chapter presents experimental results for a number of AQM algorithms that were obtained with Web traffic and a general distribution of RTTs. The results presented in this Chapter lead to the following conclusions. The conclusions were drawn under the assumption that response times are the primary performance measure, and that loss rates and link utilization, although important, are secondary.

- Overall, experimental results for Web applications with general RTT distribution showed that flows took longer to complete than in the case of uniform RTT distribution. This is because the minimum amount of delays that emulated propagation delays were larger in the case of general RTT distribution. However, the performance degradation for drop-tail and AQM algorithms with general RTT distribution at high loads was less significant than with uniform RTT distribution in Chapter 4.

- At 90% load or lower, drop-tail with a queue length of 240 packets obtained performance that was competitive to the performance of all AQM algorithms. This result

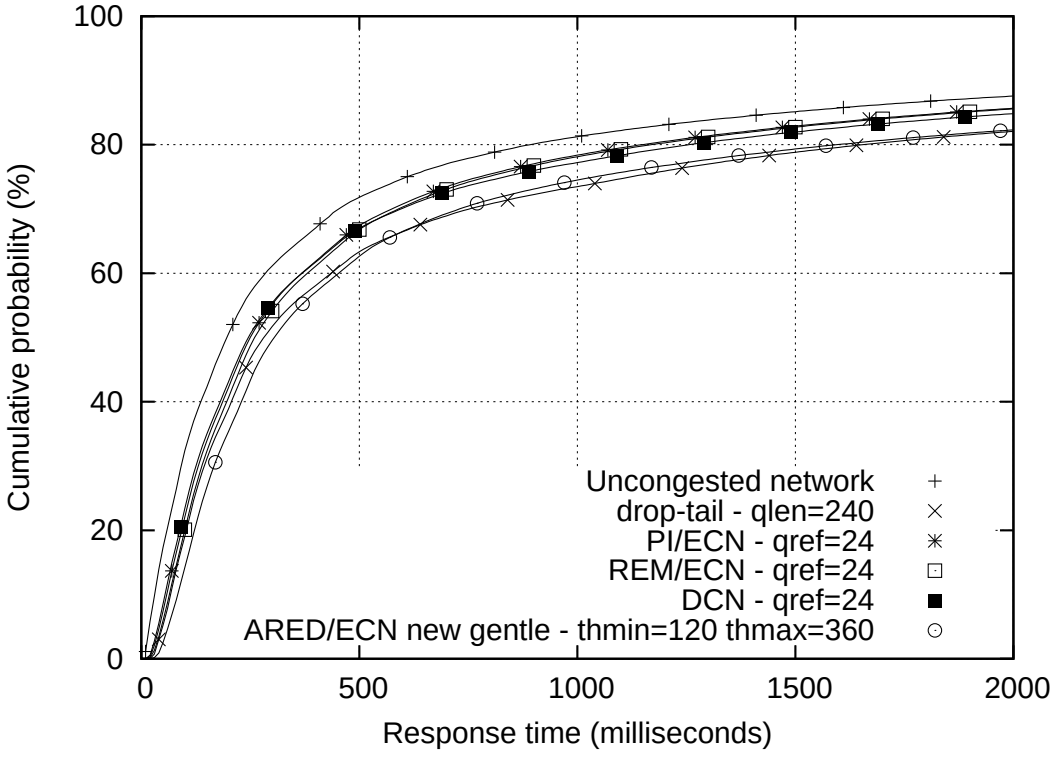

Figure 5.54: Comparison of all AQM algorithms at 98% load

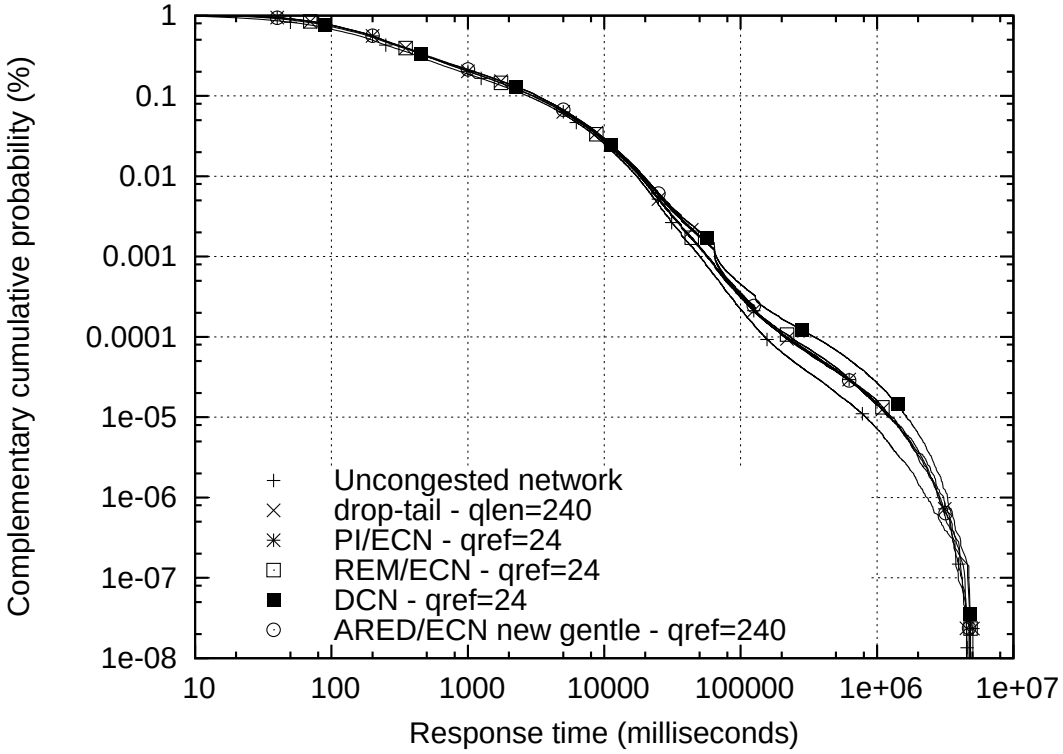

Figure 5.55: Comparison of all AQM algorithms at 90% load (CCDF)

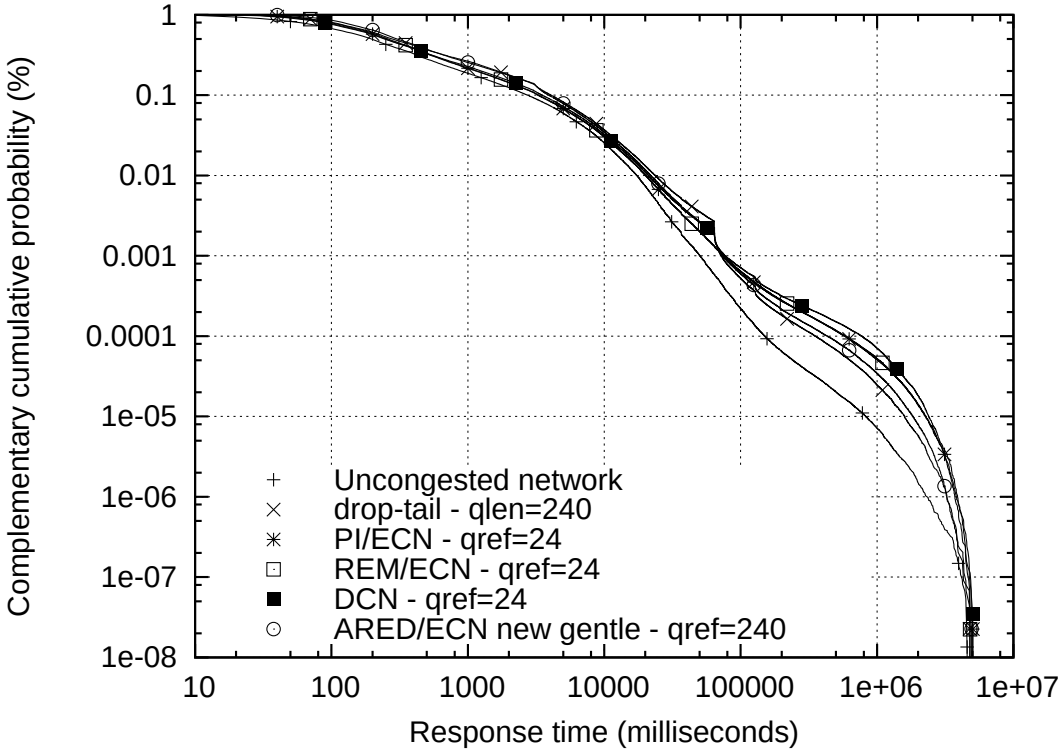

Figure 5.56: Comparison of all AQM algorithms at 98% load (CCDF)

was unchanged by the addition of the ECN protocol. Thus, AQM algorithms appear to have no advantage over drop-tail at 90% load or below.

- Since the performance degradation for drop-tail at 90% load was small, it appears that Internet Service Providers could operate their links at this high load without risking customer's dissatisfaction.

- Even at 98% load, AQM algorithms gave no performance improvement over drop-tail. However, when AQM algorithms were used with ECN, they obtained considerable performance improvement and outperformed drop-tail.

- The DCN algorithm again demonstrated the potential benefits of differential treatment of flows. In contrast to non-differential AQM algorithms PI and REM, DCN gave good performance at 98% load without requiring the ECN signaling protocol.

- The original ARED algorithm continued to give poor performance and underperformed drop-tail. This result did not change even when ECN was used. The two modifications for ARED proposed in Chapter 4 obtained considerable performance improvement over the original ARED algorithm.

Table 5.1: Loss rate, completed requests, and link utilization

	Offered load	Loss rate (%)		Completed requests (millions)		Link throughput (Mbps)	
		No ECN	ECN	No ECN	ECN	No ECN	ECN
Uncongested	90%	0.0		14.7		89.7	
1 Gbps network	98%	0.0		16.0		97.8	
Drop-tail	90%	0.8		14.4		86.2	
queue size = 24	98%	2.2		14.8		88.5	
Drop-tail	90%	0.3		14.4		86.9	
queue size = 240	98%	1.5		15.0		89.9	
PI	90%	0.1	0.0	14.5	14.6	87.3	87.4
$q_{ref} = 24$	98%	1.0	0.2	15.0	15.1	88.6	88.8
PI	90%	0.0	0.0	14.5	14.5	87.5	87.1
$q_{ref} = 240$	98%	0.5	0.1	15.2	15.2	90.5	90.4
REM	90%	0.2	0.0	14.5	14.6	87.1	87.4
$q_{ref} = 24$	98%	1.4	0.2	14.7	15.1	87.1	89.1
REM	90%	0.0	0.0	14.6	14.5	87.4	87.4
$q_{ref} = 240$	98%	1.2	0.1	14.6	15.2	86.7	89.7
ARED	90%	0.3	0.2	13.4	13.3	80.8	80.8
$th_{min} = 12$							
$th_{max} = 36$	98%	2.2	1.4	13.7	13.6	82.3	82.3
ARED	90%	0.2	0.1	13.5	14.4	84.4	86.5
$th_{min} = 120$							
$th_{max} = 360$	98%	2.1	1.3	13.7	15.0	85.9	89.7
ARED	90%	0.2		14.5		86.7	
"byte mode"							
$th_{min} = 12$	98%	1.0		14.9		89.0	
$th_{max} = 36$							
ARED	90%	0.1		14.5		87.4	
"byte mode"							
$th_{min} = 120$	98%	0.7		15.0		89.6	
$th_{max} = 360$							

Continued on next page

	Offered load	Loss rate (%)		Completed requests (millions)		Link throughput (Mbps)	
		No ECN	ECN	No ECN	ECN	No ECN	ECN
ARED/ECN "new gentle" $th_{min} = 12$ $th_{max} = 36$	90%		0.2		14.4		86.6
	98%		1.1		14.9		88.5
ARED/ECN "new gentle" $th_{min} = 120$ $th_{max} = 360$	90%		0.1		14.4		86.5
	98%		1.0		15.0		89.7
LQD $q_{ref} = 24$	90%	0.2	0.0	14.5	14.5	87.2	88.2
	98%	1.0	0.1	15.0	15.1	89.5	89.8
LQD $q_{ref} = 240$	90%	0.1	0.0	14.6	14.6	87.7	88.5
	98%	1.0	0.1	15.0	15.1	90.2	90.8
DCN $q_{ref} = 24$	80%	0.2	0.1	14.6	14.6	87.5	88.4
	90%	0.7	0.2	15.0	15.1	89.3	90.6
DCN $q_{ref} = 240$	80%	0.1	0.1	14.6	14.5	87.8	88.5
	90%	0.5	0.2	15.0	15.2	89.5	90.7

Table 5.2: Percentiles of response times

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
Uncongested	90%	0.195	0.605	2.745
1 Gbps network	98%	0.195	0.605	2.745
Drop-tail	90%	0.235	0.835	3.435
queue size = 24	98%	0.275	1.295	4.465
Drop-tail	90%	0.235	0.745	3.225
queue size = 240	98%	0.285	1.125	4.085
PI	90%	0.235	0.715	3.045
$q_{ref} = 24$	98%	0.275	1.095	3.955
PI/ECN	90%	0.235	0.705	2.995
$q_{ref} = 24$	98%	0.245	0.775	3.275
PI	90%	0.275	0.765	3.085
$q_{ref} = 240$	98%	0.335	1.015	3.755
PI/ECN	90%	0.265	0.735	3.045
$q_{ref} = 240$	98%	0.315	0.885	3.375
REM	90%	0.235	0.725	3.075
$q_{ref} = 24$	98%	0.285	1.235	4.325
REM/ECN	90%	0.235	0.705	3.025
$q_{ref} = 24$	98%	0.265	0.795	3.295
REM	90%	0.245	0.715	3.015
$q_{ref} = 240$	98%	0.325	1.255	4.395
REM/ECN	90%	0.245	0.705	3.005
$q_{ref} = 240$	98%	0.295	0.855	3.375
ARED	90%	0.285	1.505	4.945
$th_{min} = 12$				
$th_{max} = 36$	98%	0.365	2.215	6.325
ARED/ECN	90%	0.285	1.475	4.955
$th_{min} = 12$				
$th_{max} = 36$	98%	0.375	2.335	6.545

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
ARED $th_{min} = 120$ $th_{max} = 360$	90%	0.285	1.455	4.815
	98%	0.365	2.225	6.335
ARED/ECN $th_{min} = 120$ $th_{max} = 360$	90%	0.285	1.455	4.885
	98%	0.375	2.215	6.375
ARED "byte mode" $th_{min} = 12$ $th_{max} = 36$	90%	0.235	0.735	3.145
	98%	0.265	1.085	4.005
ARED "byte mode" $th_{min} = 120$ $th_{max} = 360$	90%	0.235	0.715	3.045
	98%	0.285	0.995	3.755
ARED/ECN "new gentle" $th_{min} = 12$ $th_{max} = 36$	90%	0.235	0.765	3.295
	98%	0.285	1.235	4.355
ARED/ECN "new gentle" $th_{min} = 120$ $th_{max} = 360$	90%	0.245	0.755	3.245
	98%	0.305	1.035	3.925
LQD $q_{ref} = 24$	90%	0.225	0.735	3.145
	98%	0.265	1.125	4.135
LQD/ECN $q_{ref} = 24$	90%	0.225	0.695	3.005
	98%	0.255	0.775	3.275
LQD $q_{ref} = 240$	90%	0.235	0.705	3.015
	98%	0.285	1.045	3.935
LQD/ECN $q_{ref} = 240$	90%	0.225	0.685	2.965
	98%	0.255	0.785	3.245
DCN $q_{ref} = 24$	90%	0.225	0.725	3.225
	98%	0.245	0.835	3.505

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
DCN/ECN	90%	0.225	0.695	3.045
$q_{ref} = 24$	98%	0.245	0.795	3.345
DCN	90%	0.225	0.745	3.255
$q_{ref} = 240$	98%	0.275	0.875	3.535
DCN/ECN	90%	0.235	0.715	3.075
$q_{ref} = 240$	98%	0.285	0.845	3.345

Chapter 6

Results with General TCP Traffic

Experimental results presented in Chapter 5 relaxed the assumption about the approximated uniform distribution of RTTs within the U.S. continental that was previously made for results in Chapter 4. Nevertheless, results presented in both Chapters 4 and 5 were limited to only Web traffic. In order to draw more general conclusions about the effects of AQM algorithms on network and application performance, experiments were performed with synthetic traffic that was derived from the full mix of TCP connections captured on Internet links.

A 2-hour packet trace taken on an Abilene (Internet 2) link between Cleveland and Indianapolis was used to drive experiments with general TCP traffic. The data to drive these experiments was acquired from the trace repository of the National Laboratory for Applied Network Research (NLANR) [NLA05]. The distributions of RTTs and the sizes of application data units (ADUs) derived from the packet trace are depicted in Figures 6.1 and 6.2. The packet trace was filtered for all TCP connections including HTTP, FTP, SMTP, NNTP, and peer-to-peer file-sharing traffic. The synthetic TCP traffic mix used to obtained experimental results in this Chapter represents the characteristics of existing Internet backbone traffic as seen by routers in real network and provides the most realistic method for evaluating AQM algorithms in a laboratory network. The application used to generate synthetic TCP traffic from raw packet traces is called *tmix* and was described in Chapter 3.

Nominal offered loads on the link between the routers in the laboratory network was obtained by a process of sub-sampling or superimposing connections from the original packet trace. The sub-sampling or superimposing process decreases or increases the nominal load from the original packet trace while preserving the mix and statistical characteristics of the TCP connections [CJS04].

The aforementioned scaling process was used to achieve offered loads of 80, 90, and 95 Mbps on an uncongested 1-Gbps network. These loads were termed 80%, 90%, and 95% because their corresponding throughputs represented 80%, 90%, and 95% utilization

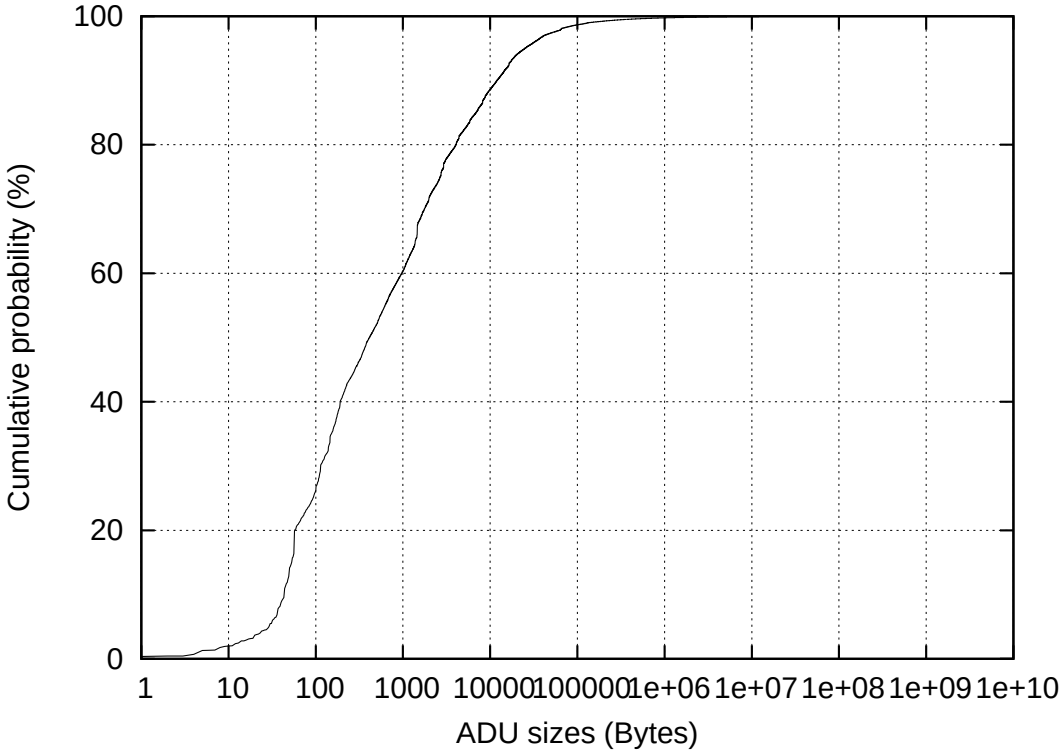

Figure 6.1: Distribution of ADU sizes of general TCP traffic

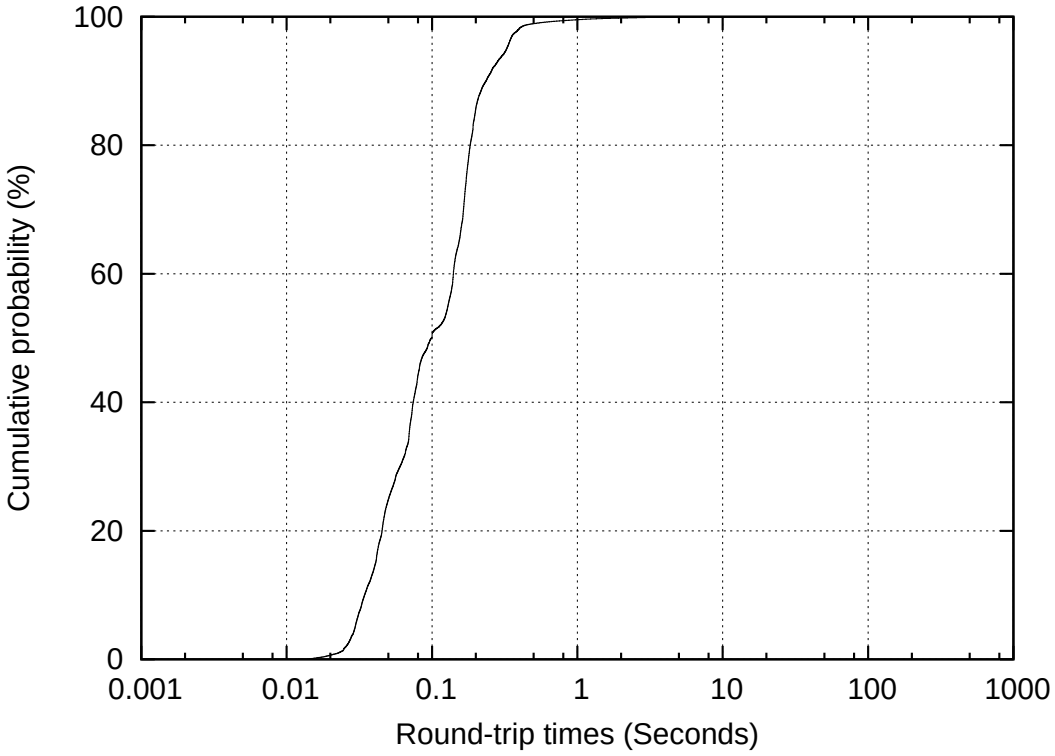

Figure 6.2: Distribution of RTT distribution of general TCP traffic

of a congested 100-Mbps link. Although these were the throughputs that could be possibly achieved on a 100-Mbps link, as the offered load approaches the saturation of a 100-Mbps link, the actual link utilization will be less than the intended offered load. This is because as the network becomes congested, TCP dynamics will regulate the transmission rates of the end systems. An interesting aspect of using Abilene traffic for experiments is that Abilene traffic is asymmetrical between forward and reverse paths. For example, the offered loads of 95% derived from the Abilene trace were approximately 95.73 Mbps (forward path) and 90.70 Mbps (reverse path).

The offered loads for general TCP traffic were slightly lower than those for Web traffic used in Chapter 4 and 5 because synthetic TCP traffic is generated in a different way than Web traffic. For Web traffic, a nominal offered load is obtained by using a fixed number of browsing users. In turn, each browsing user has a small number of parallel connections that can be used to fetch multiple web objects simultaneously. As the offered load approaches the saturation of the link between the two routers, connections need longer to complete. However, the number of active connections during an experiment stayed at a relatively constant level for Web traffic since a simulated user only starts a new connection when it completes one of its current connections.

In contrast to Web traffic, the start times for TCP connections in an experiment with general TCP traffic were scheduled ahead of time, i.e., before the experiment started. Since connections need longer to complete at a high load and more connections arrive as the experiment progresses, the number of active connections increase with time in the case of general TCP traffic. If the nominal offered load is too high, the number of active connections could exhaust resources of an end system (*mbufs*, physical memory, or sockets) and scheduled connections would fail to start. For this reason, the maximum offered load chosen for general TCP traffic (95%) is lower than that of Web traffic (105%). In particular, offered loads for general TCP traffic cannot be greater than 100%.

In Chapter 4 and 5, response times for exchanges of requests and responses were used as the primary yardstick in performance evaluation for AQM algorithms. However, since response times are rather specific for Web application, a new performance measure is needed for evaluation of AQM algorithms with general TCP applications.

The key indicator of performance used in reporting experimental results for general TCP applications is the end-to-end connection durations. This performance measure is defined as the elapsed time necessary to establish a connection between two end systems and perform a sequence of exchanges of application data units via that connection. Connection durations can be viewed as a generalization of response times that were used as the primary performance metric in Chapter 4 and 5. For Web applications, connection durations can be interpreted as the latency necessary to establish a persistent connection from a client to a web server and fetch multiple objects from the server. The end-to-end connection

durations are reported as plots of the cumulative distributions of times up to 5 seconds. Other performance metrics such as link utilization and loss rates are also reported.

As in Chapter 5, experiments were performed for PI, REM, ARED, LQD, and DCN to obtain a range or "envelope" of results that an AQM algorithm could possibly achieve. For comparison purposes, experimental results were also obtained for drop-tail and the uncongested network.

6.1 Results for Drop-Tail

Figures 6.3, 6.4, and 6.5 show experimental results for drop-tail with general TCP traffic at 80%, 90%, and 95% loads. These results were obtained for drop-tail with a queue length of 24 and 240 packets.

At 80% offered load, drop-tail with both queue lengths of 24 and 240 packets delivered indistinguishable performance from that of the uncongested network. It is interesting to note that only approximately 75% of connections complete within 5 seconds even on the uncongested network.

At 90% load, drop-tail suffered a small performance degradation with both queue lengths. At this load, both queue lengths for drop-tail obtained identical performance. Further, the performance degradation for drop-tail at this load is negligible.

As the offered load increased to 95%, the performance for drop-tail with both queue lengths degraded noticeably. However, drop-tail still gave reasonably good performance at this high load. Figure 6.5 shows a small trade-off between short and long queue lengths (24 packets vs. 240 packets) for drop-tail in obtaining good connection durations for short and long connections. Similar trade-offs in obtaining good response times for short and long connections were observed and discussed in sections 4.1 and 4.2.

6.2 Results for ARED, PI, LQD, and REM

Experiments were performed with general TCP traffic for PI, REM, LQD, and DCN with a queue reference of 24 and 240 packets at 80%, 90%, and 95% loads. Since ARED does not allow to set a queue reference explicitly, experiments were performed for ARED "packet mode" and "byte mode" with (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). These parameter settings yielded a target queue reference of 24 and 240 packets for ARED respectively. Experimental results of AQM algorithms were compared with the results of drop-tail and of the uncongested network to evaluate the effects of AQM algorithms on general TCP applications.

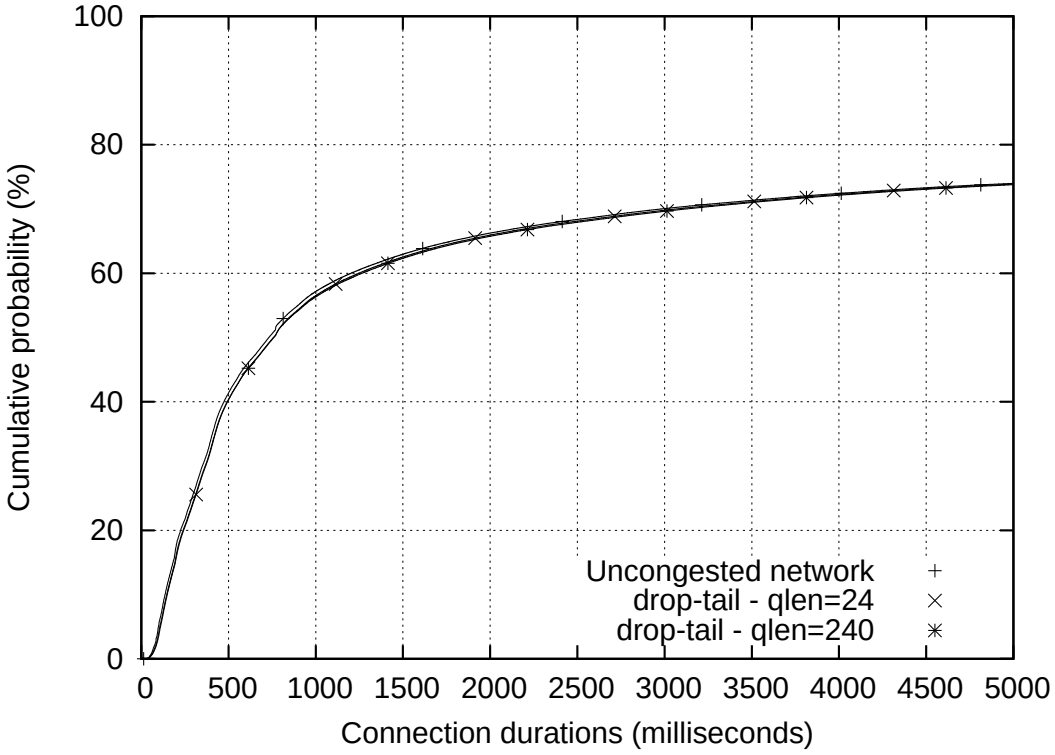

Figure 6.3: Drop-tail performance at 80% load

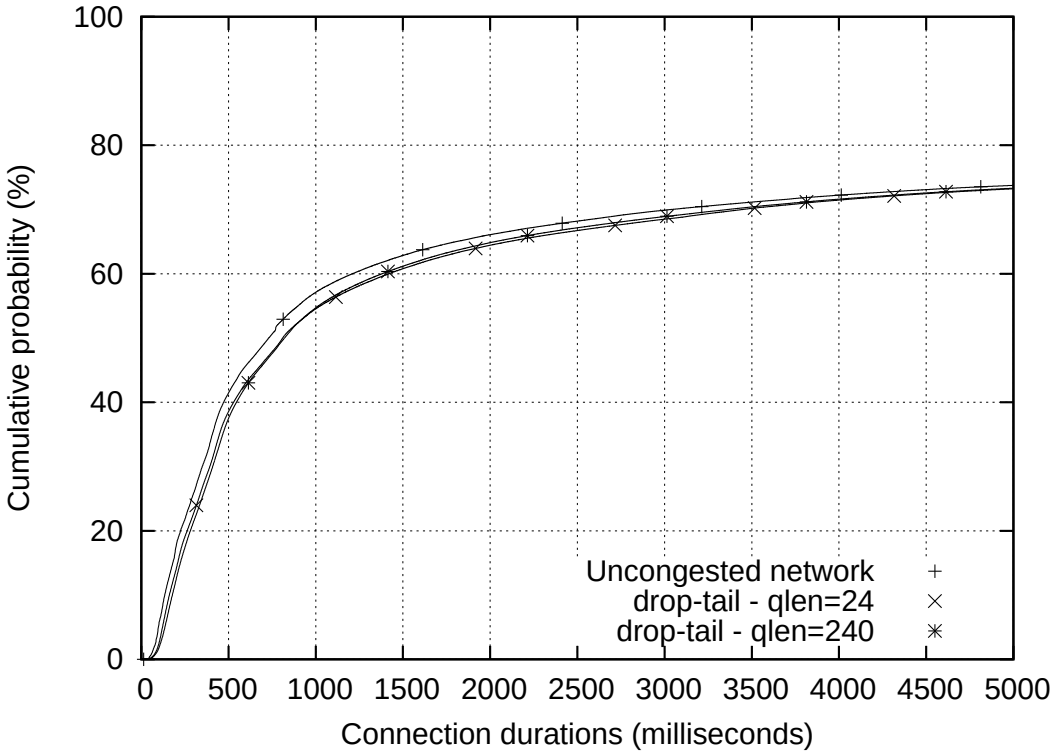

Figure 6.4: Drop-tail performance at 90% load

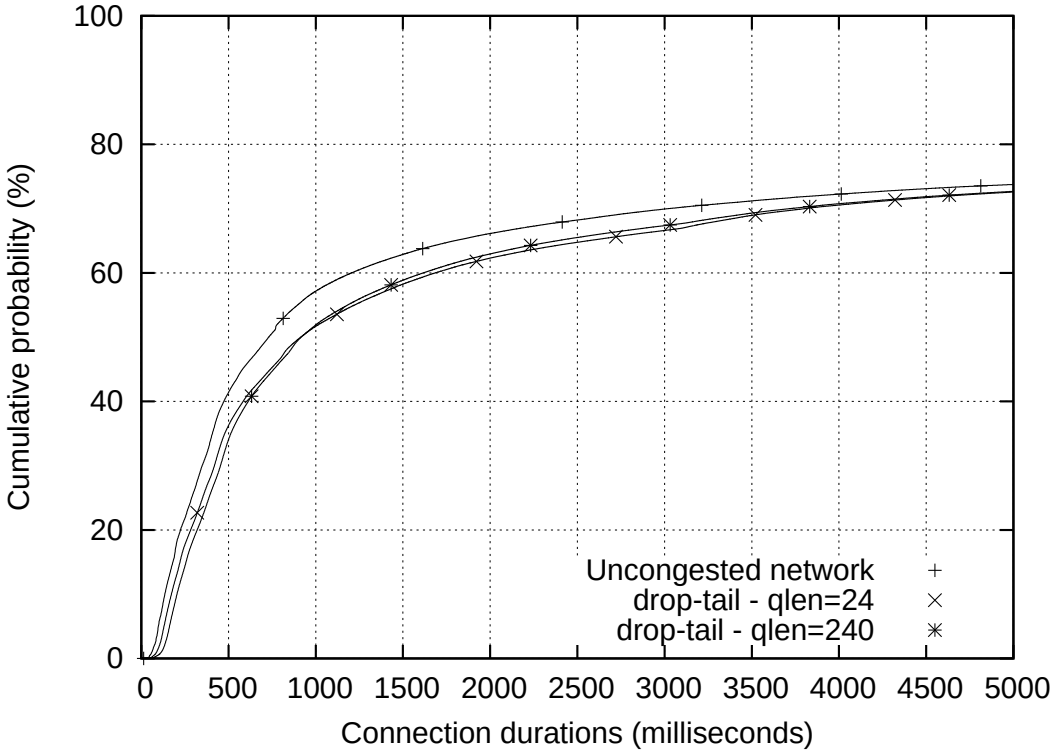

Figure 6.5: Drop-tail performance at 95% load

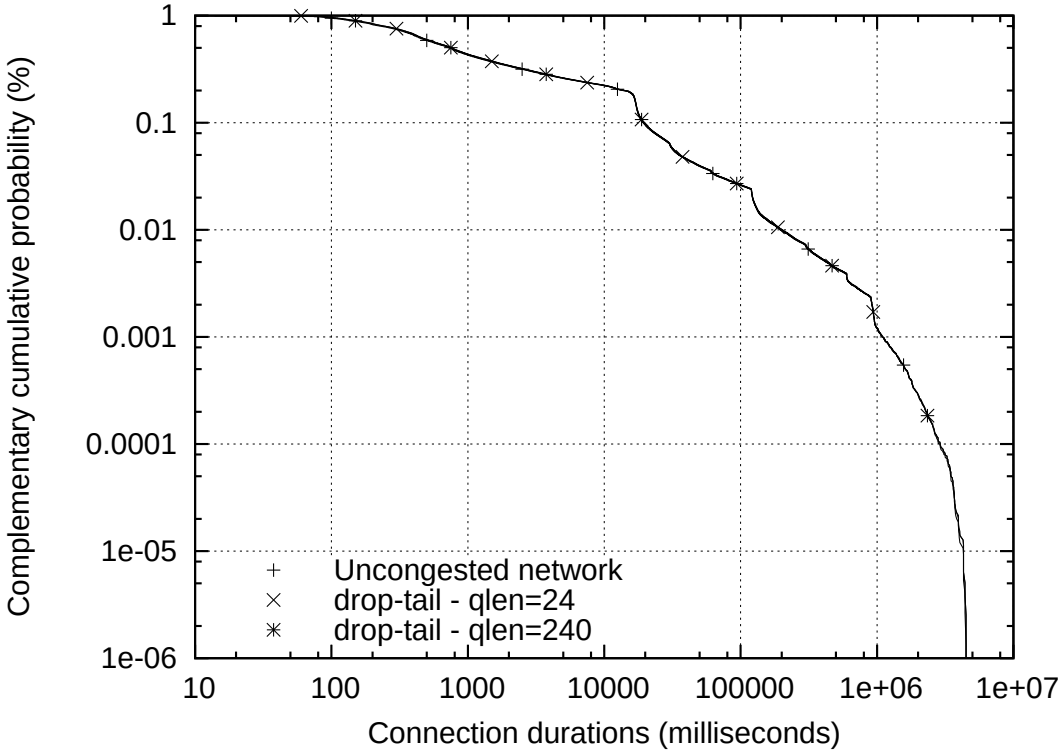

Figure 6.6: Drop-tail performance at 80% load (CCDF)

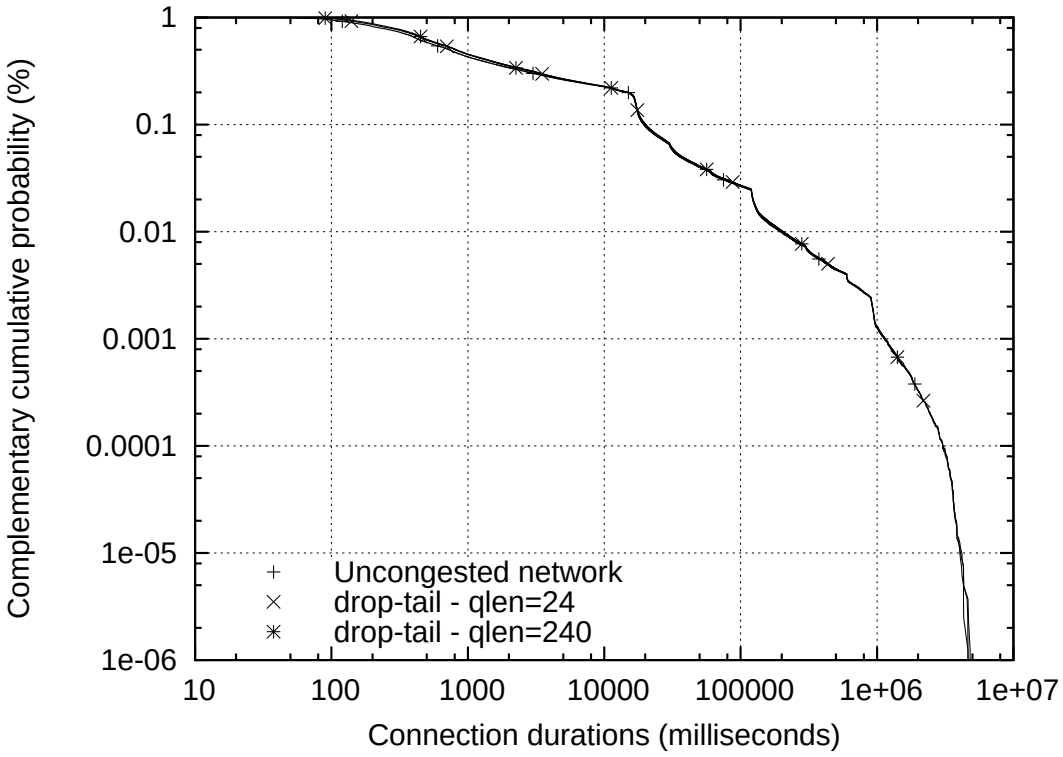

Figure 6.7: Drop-tail performance at 90% load (CCDF)

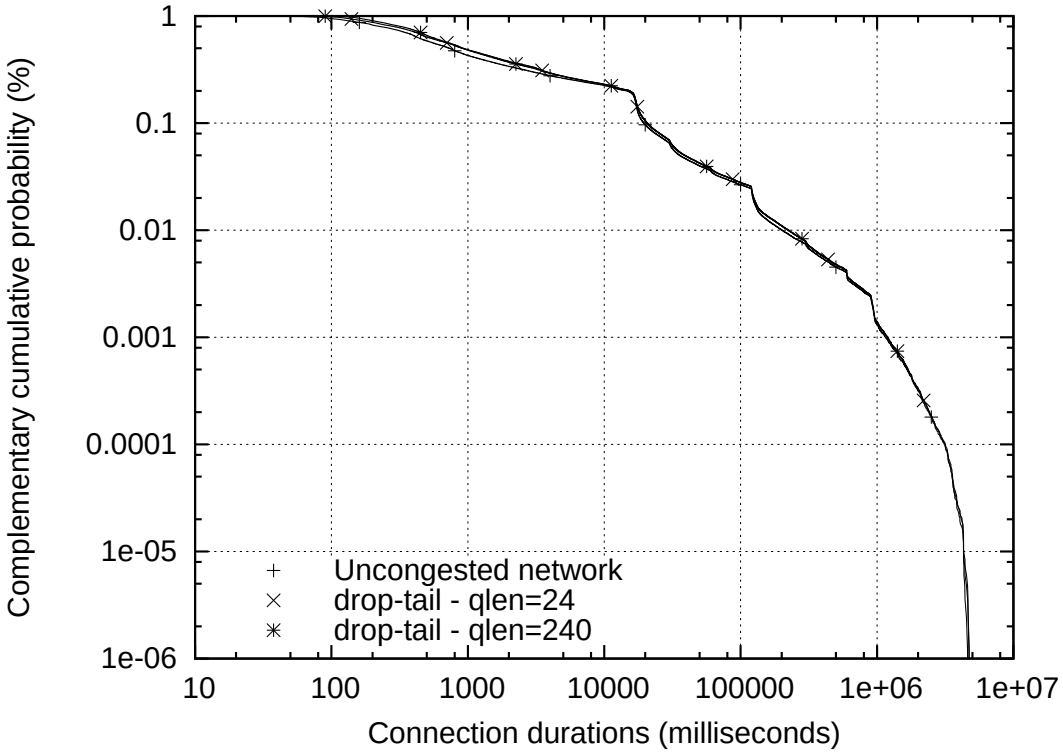

Figure 6.8: Drop-tail performance at 95% load (CCDF)

6.2.1 Results for PI

Figures 6.9, 6.10, and 6.11 show experimental results for PI with general TCP applications at 80%, 90%, and 95% loads. These results were obtained with a queue reference of 24 and 240 packets for PI.

At 80% load, PI gave equally good performance with both queue references of 24 and 240 packets. The performance of PI at this load was indistinguishable from that of drop-tail and of the uncongested network.

At 90% load, PI with a queue reference of 24 packets gave the same performance as drop-tail. However, PI with a queue reference of 240 packets delivered slightly worse performance than drop-tail for approximately 65% of flows. These were flows that completed within 2 seconds. The other 35% of flows, those that needed more than 2 seconds to finish, experienced the same performance with drop-tail and with both queue references for PI. Overall, drop-tail and PI with a queue reference of 24 packets closely approximated the performance of the uncongested network at this load.

As the offered load increased to 95%, the PI with both queue references experienced further performance degradation. However, PI with both queue references showed a small advantage over drop-tail at this load. PI with a queue references of 24 packets delivered slightly better performance than drop-tail for the shortest 70% of flows and identical performance as drop-tail for the rest 30% of flows. PI with a queue reference of 240 packets gave the same performance as drop-tail for 50% of flows that completed within 1 second and slightly better performance than drop-tail for the other 50% of flows. Between the two queue references of 24 and 240 packets, PI obtained better performance with a queue reference of 24 packets at this load.

6.2.2 Results for REM

Figures 6.15, 6.16, and 6.17 show experimental results for REM with general TCP applications at 80%, 90%, and 95% when REM was used with packet drops. These results were obtained for REM with a queue reference of 24 and 240 packets.

At 80% offered load, REM delivered similar performance for general TCP applications with both queue references of 24 and 240 packets. The performance for REM was identical to that of drop-tail and of the uncongested network at this load.

At 90% load, REM suffered a small performance degradation with both queue references. The performance for REM with both queue references at this load was equal to that of drop-tail and came close to the performance of the uncongested network.

At 95% offered load, REM obtained slightly better performance with a queue reference of 24 packets than with a queue reference of 240 packets. At this load, REM with a queue reference of 24 packets also gave better performance than drop-tail for approximately

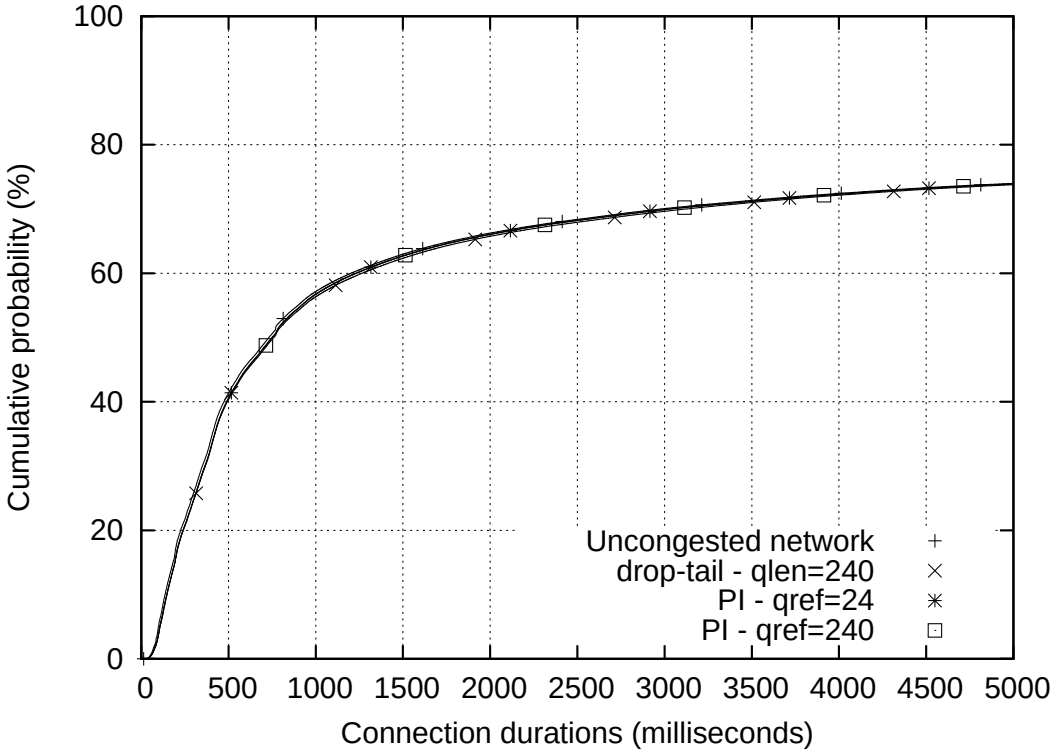

Figure 6.9: PI performance at 80% load

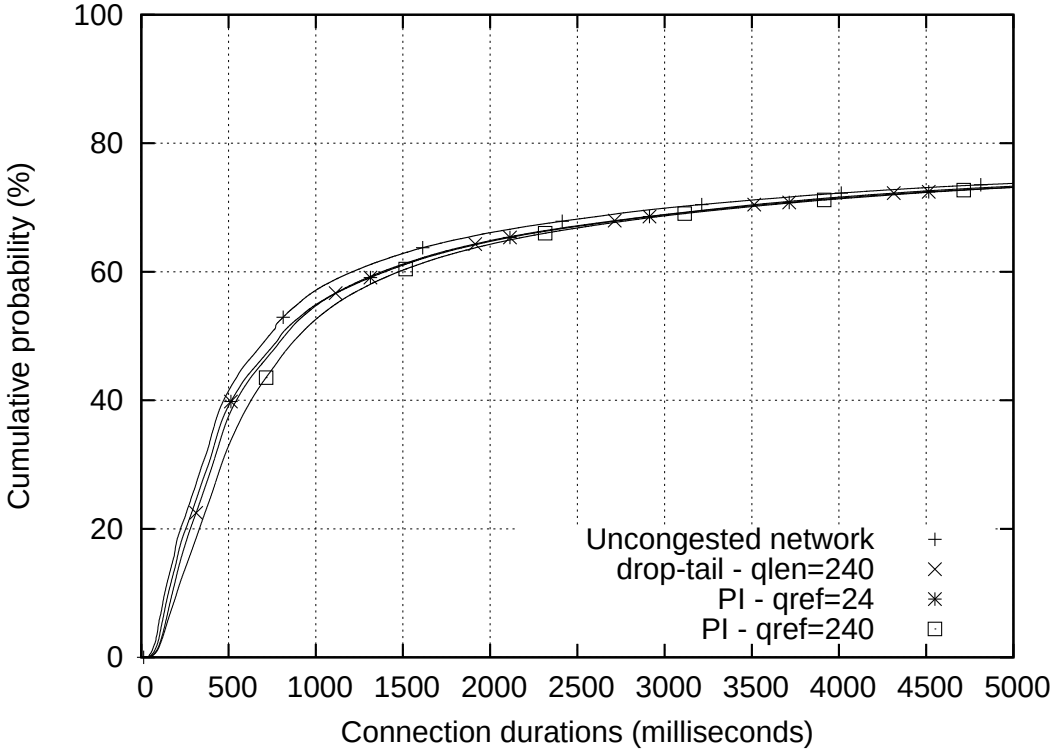

Figure 6.10: PI performance at 90% load

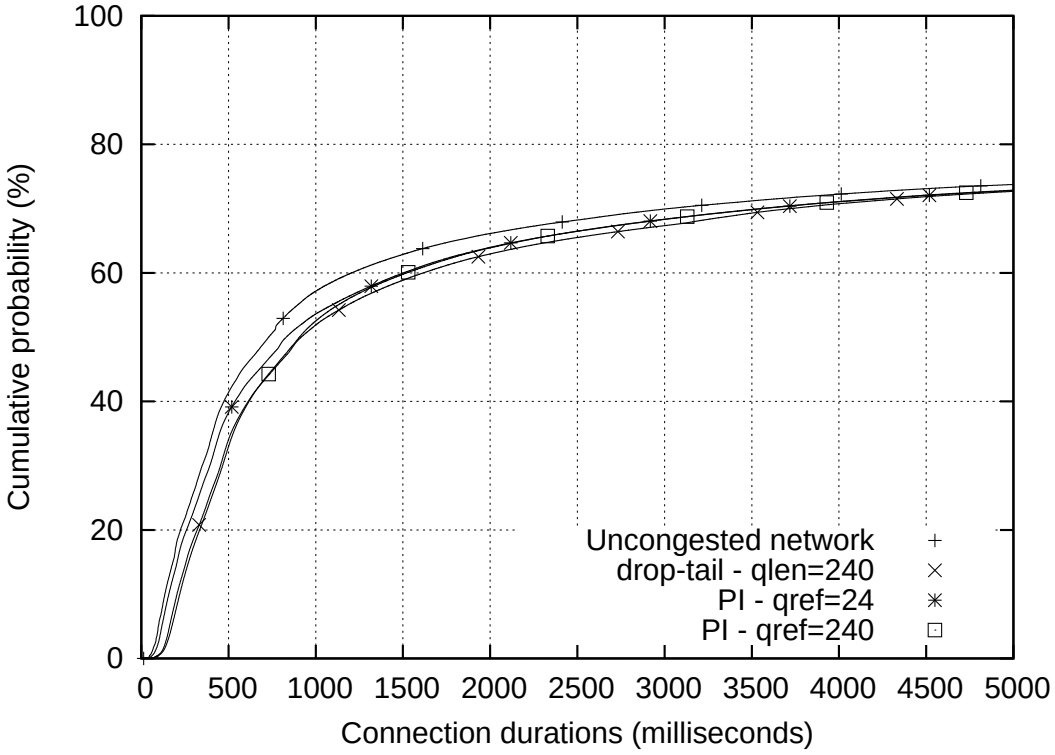

Figure 6.11: PI performance at 95% load

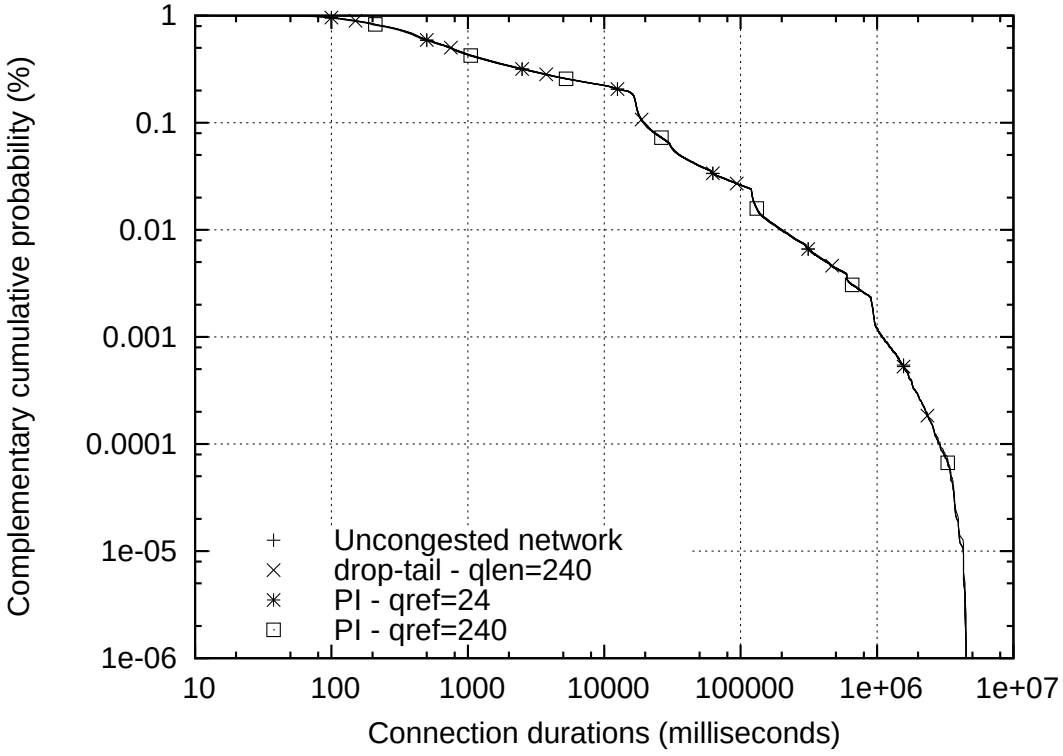

Figure 6.12: PI performance at 80% load (CCDF)

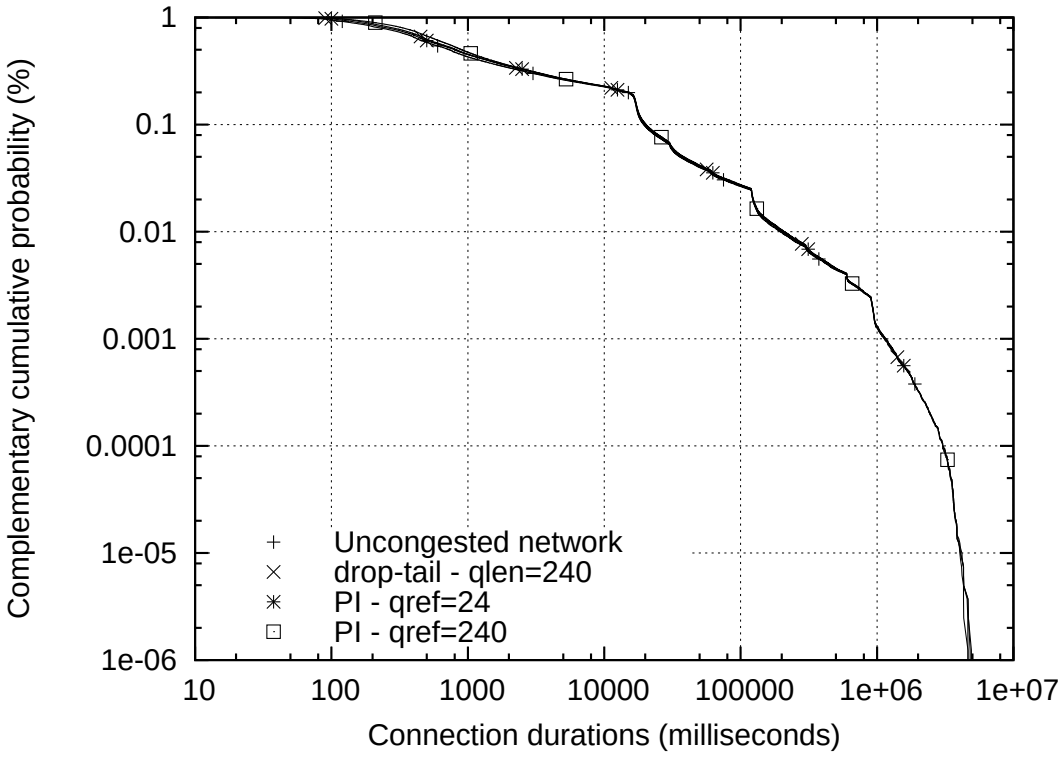

Figure 6.13: PI performance at 90% load (CCDF)

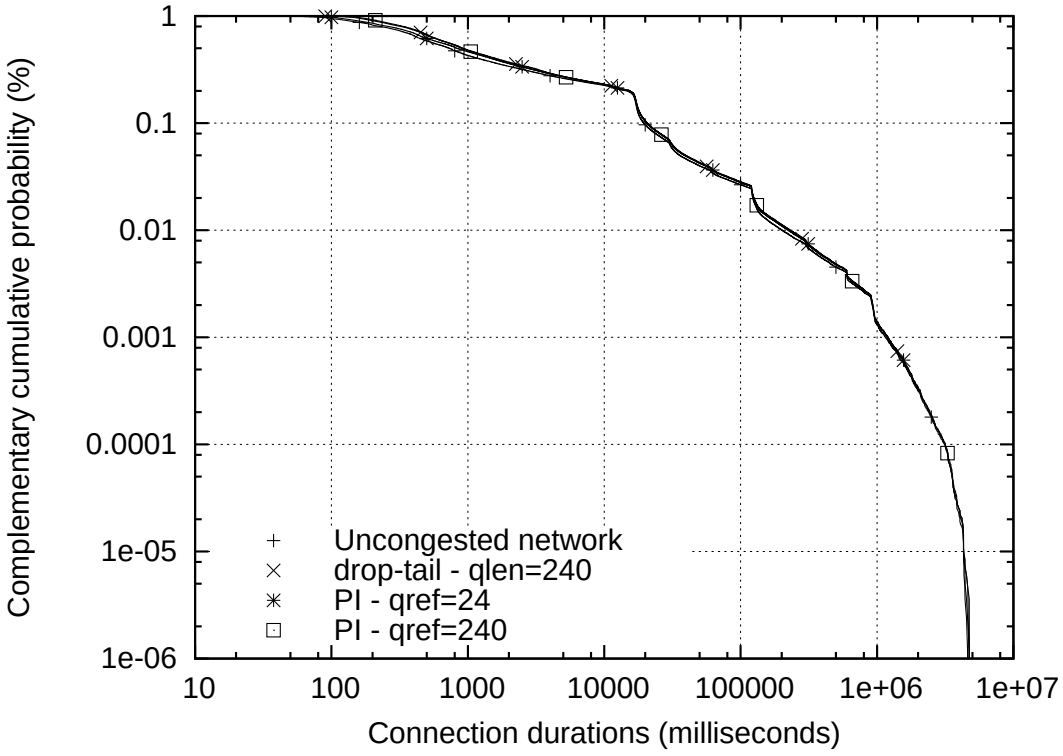

Figure 6.14: PI performance at 95% load (CCDF)

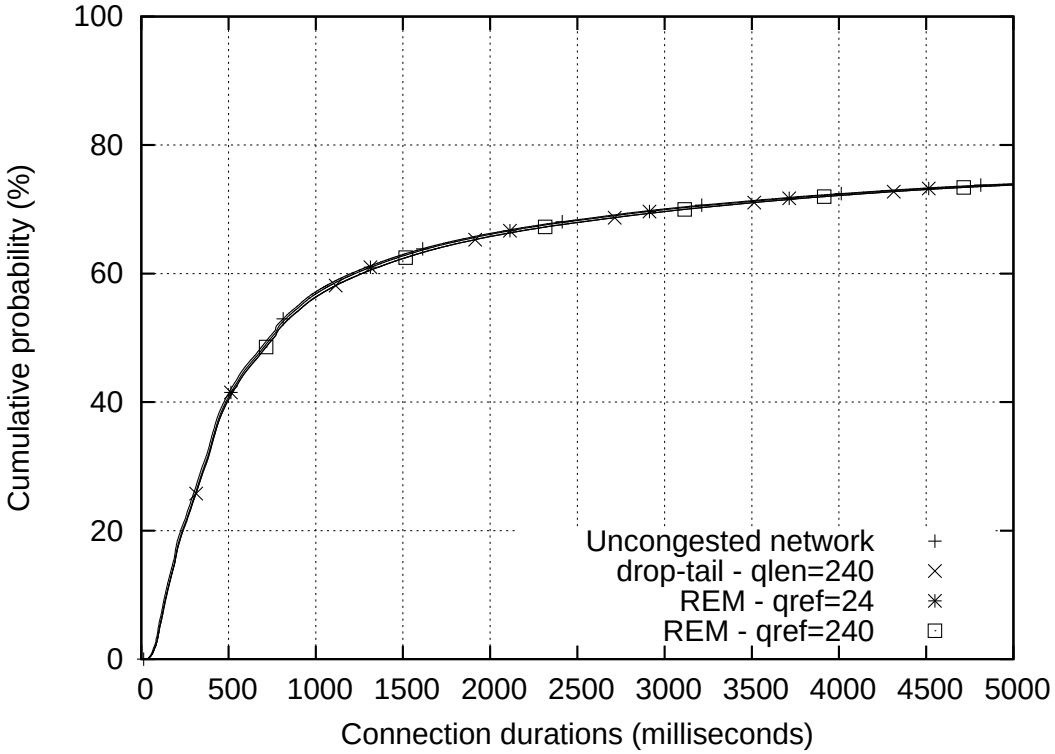

Figure 6.15: REM performance at 80% load

70% of flows that finished within 4 seconds. The longest 30% of flows experienced the same performance when experiments were performed for drop-tail and REM with a queue reference of 24 packets. REM with a queue reference of 240 packets delivered similar performance as drop-tail overall. However, REM with a queue reference of 240 packets gave slightly better performance than drop-tail for approximately 35% of flows that needed more than 500 milliseconds but less than 4 seconds to complete.

6.2.3 Results for ARED

Figures 6.21, 6.22, 6.23, 6.27, 6.28, and 6.29 show experimental results for ARED "packet mode" and "byte mode" with general TCP applications. These results were obtained with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). These parameter settings for ARED resulted in an implicit target queue reference of 24 and 240n packets respectively.

At 80% offered load, the original ARED algorithm in "packet mode" gave identical performance as drop-tail and the uncongested network with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets).

As the offered load increased to 90%, ARED "packet mode" with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) delivered the same performance as drop-tail

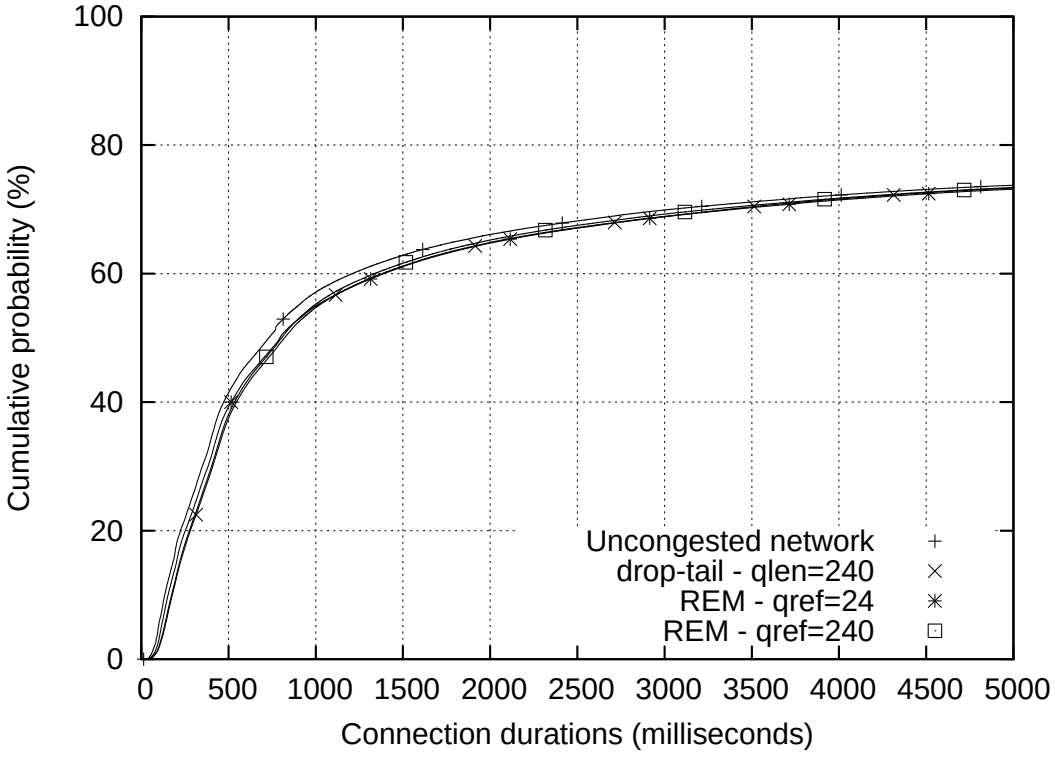

Figure 6.16: REM performance at 90% load

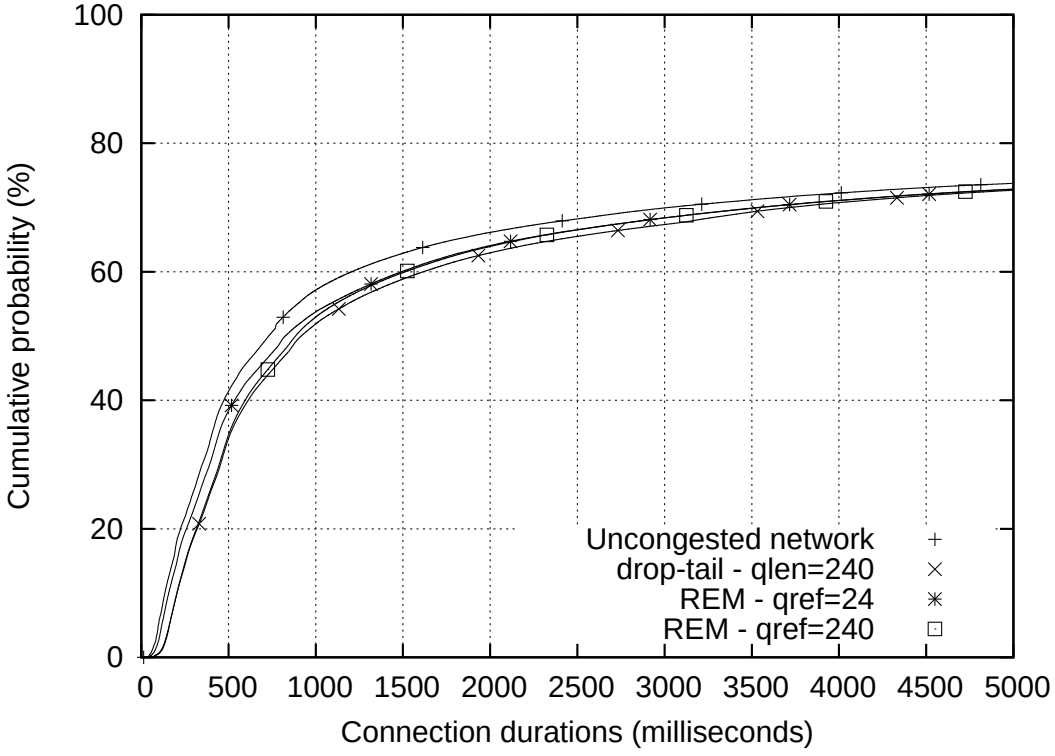

Figure 6.17: REM performance at 95% load

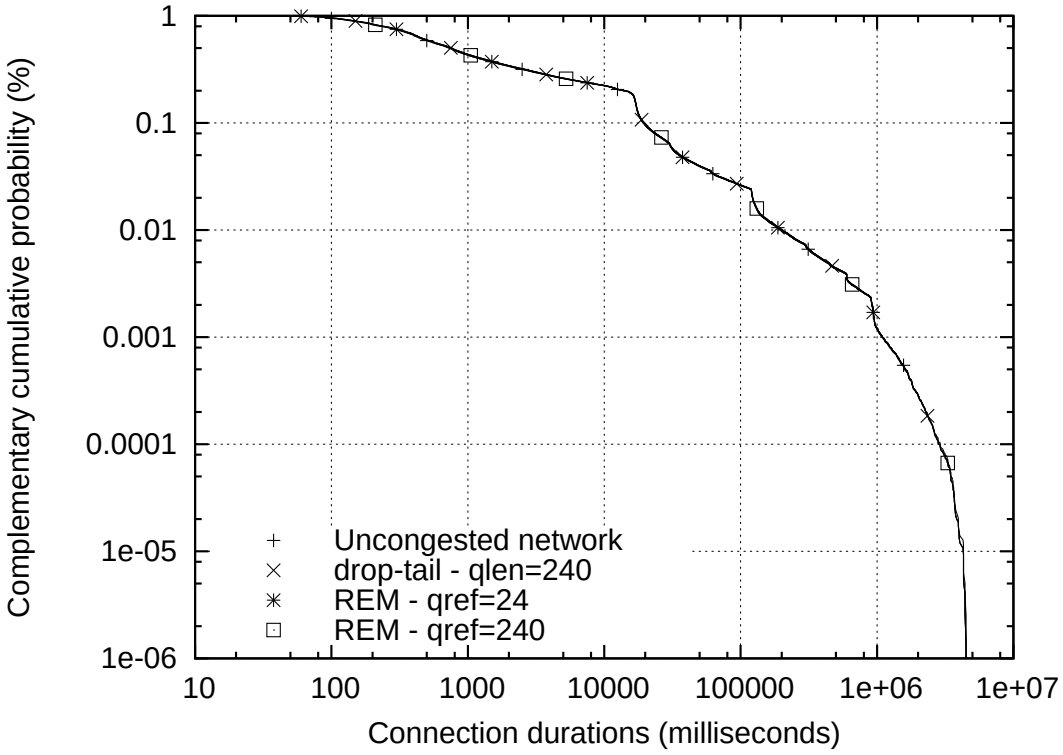

Figure 6.18: REM performance at 80% load (CCDF)

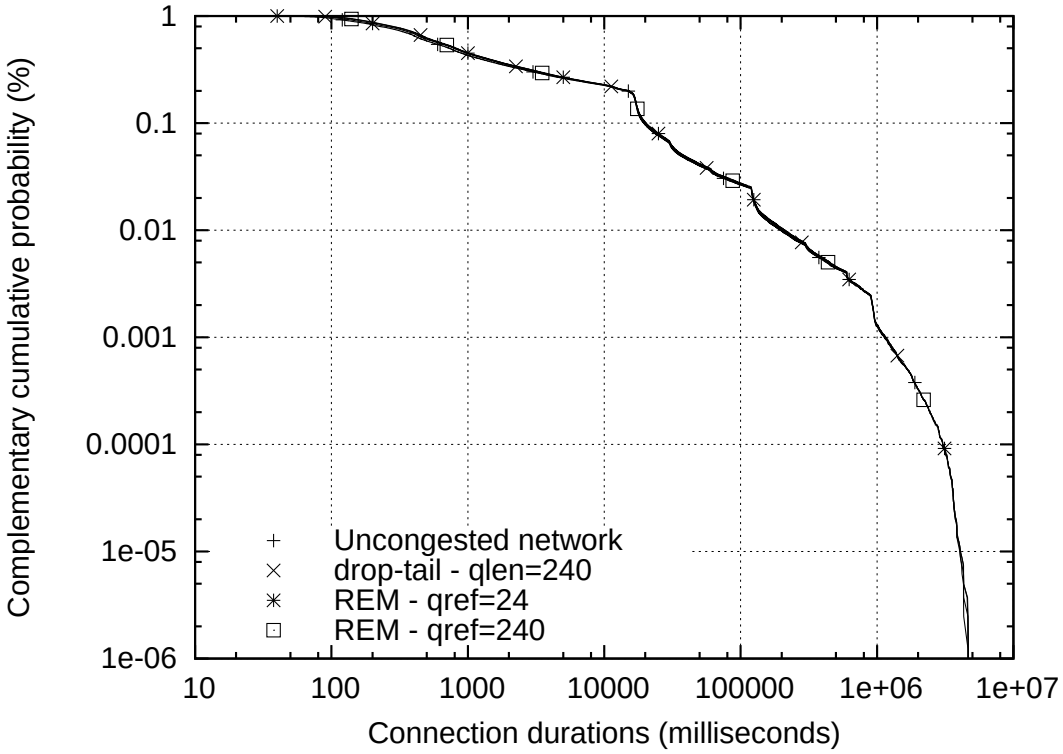

Figure 6.19: REM performance at 90% load (CCDF)

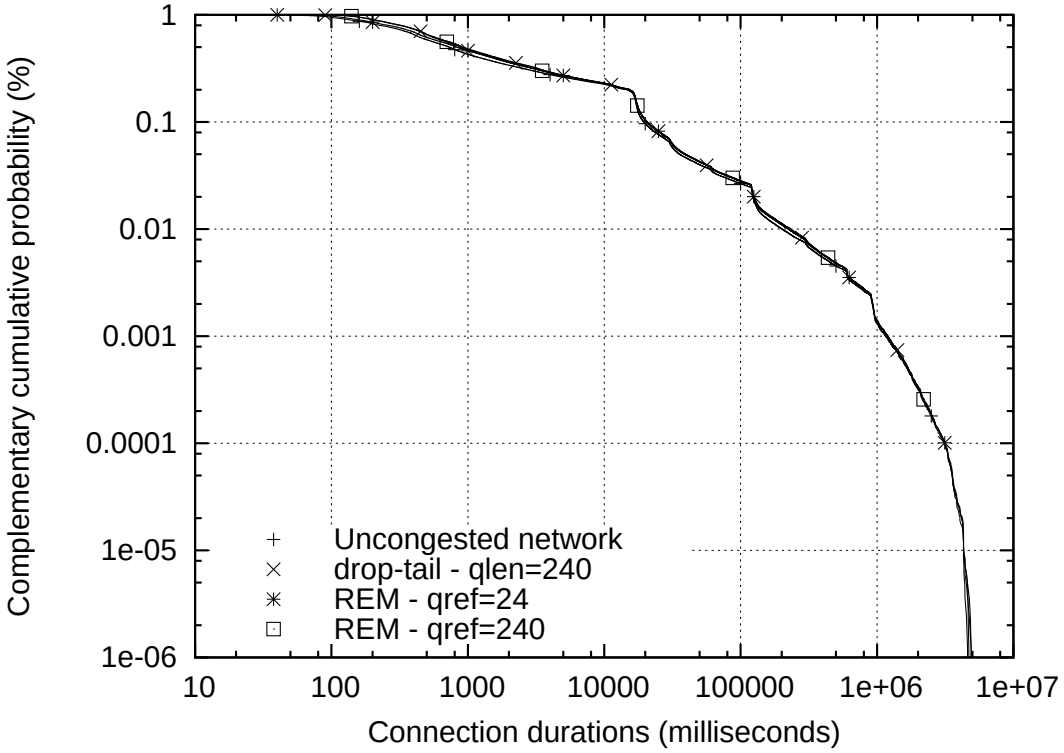

Figure 6.20: REM performance at 95% load (CCDF)

and suffered a small performance degradation when compared to the uncongested network. On the other hand, ARED "packet mode" with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) suffered noticeable performance degradation and underperformed drop-tail considerably.

At 95% load, ARED "packet mode" with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) suffered further performance degradation but it continued to identical performance as drop-tail. ARED "packet mode" with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) experienced even more performance degradation and gave poorer performance than drop-tail again.

At 80% load, ARED "byte mode" with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) obtained the same performance as drop-tail, ARED "packet mode" and the uncongested network.

At 90% offered load, ARED "byte mode" obtained indistinguishable performance with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). The performance for ARED "byte mode" at this load was identical to that of drop-tail and closely approximated the performance of the uncongested network. When used with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets), ARED "byte mode" outperformed ARED "packet mode". However, when used with pa-

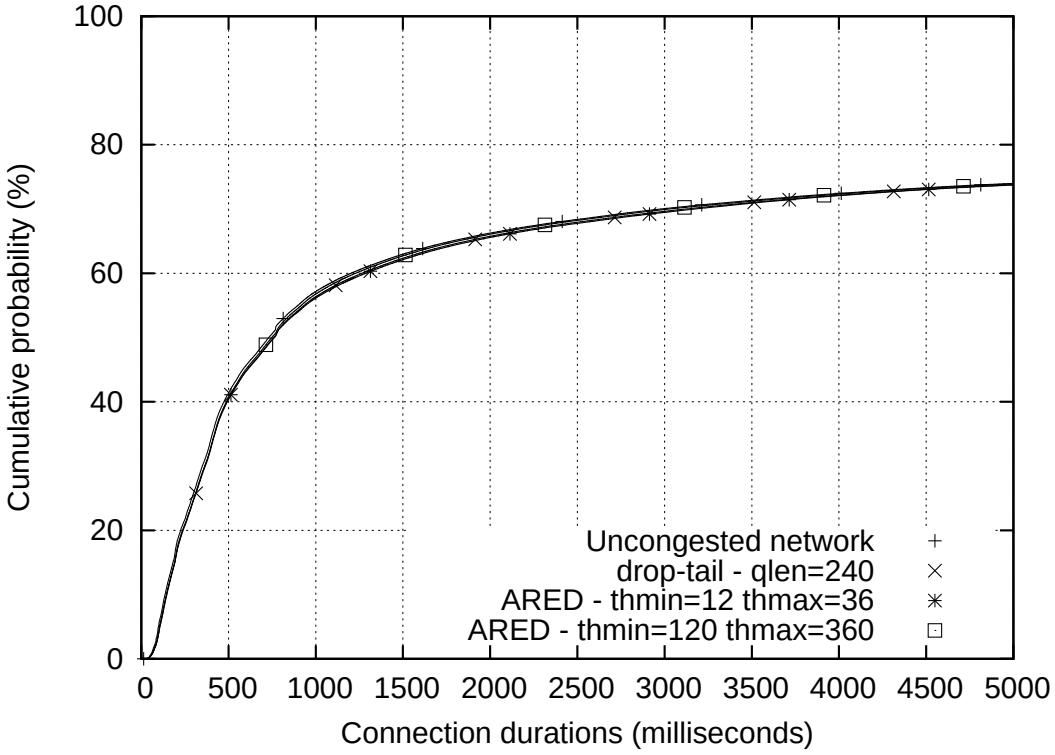

Figure 6.21: ARED performance at 80% load

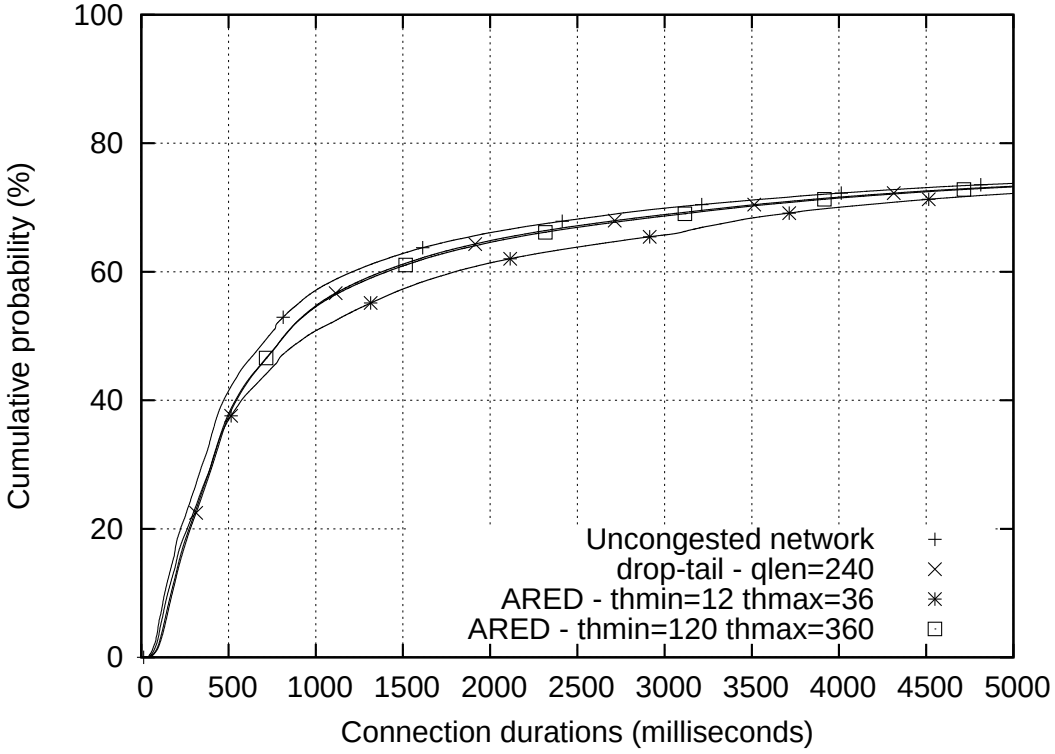

Figure 6.22: ARED performance at 90% load

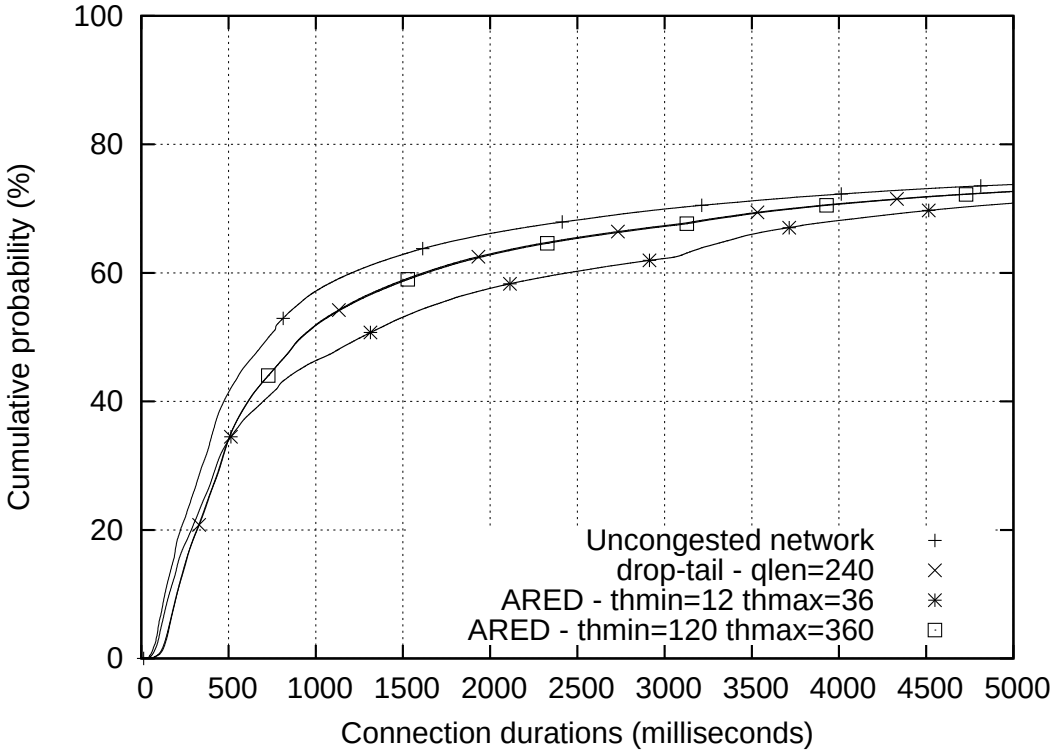

Figure 6.23: ARED performance at 95% load

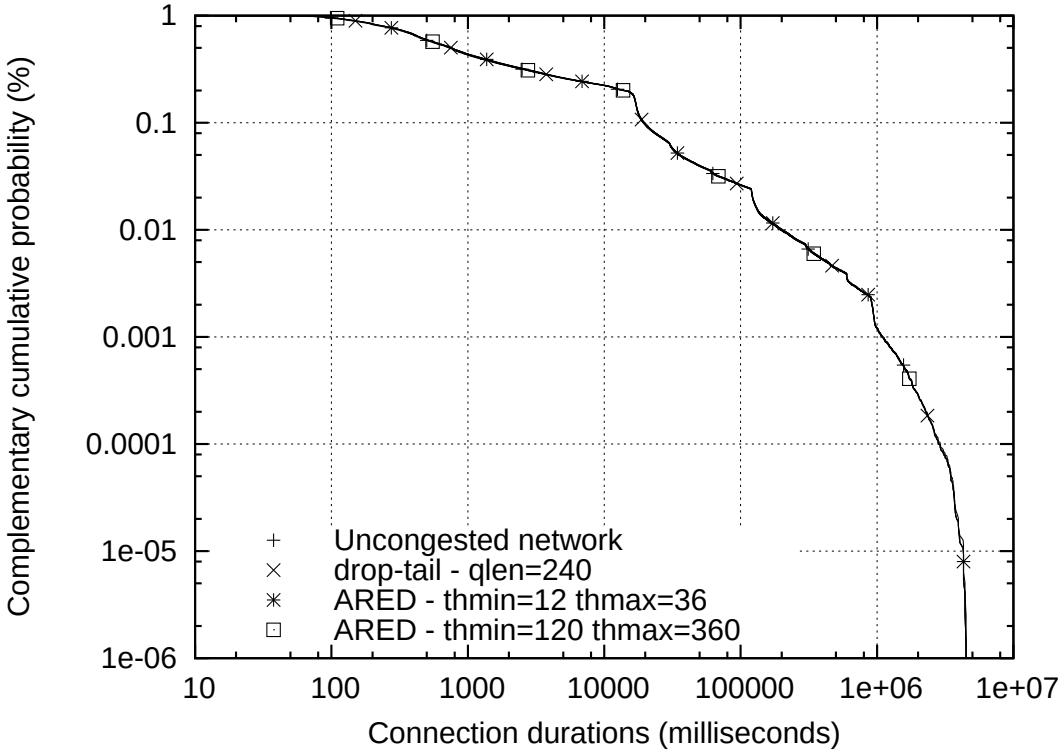

Figure 6.24: ARED performance at 80% load (CCDF)

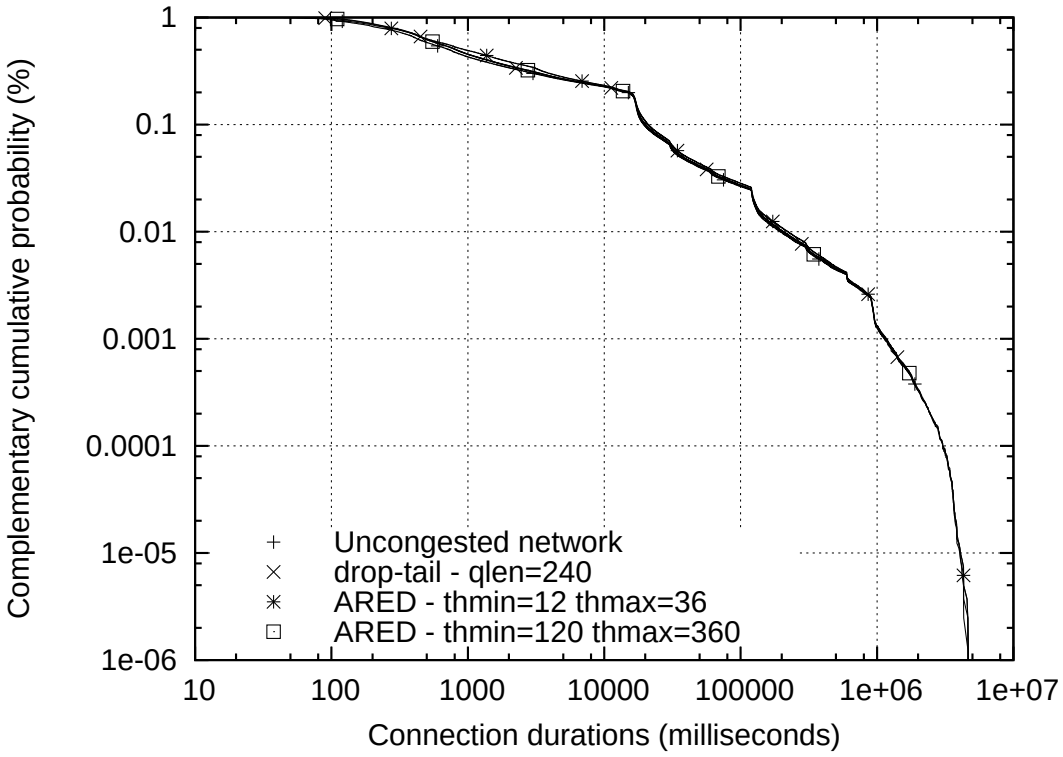

Figure 6.25: ARED performance at 90% load (CCDF)

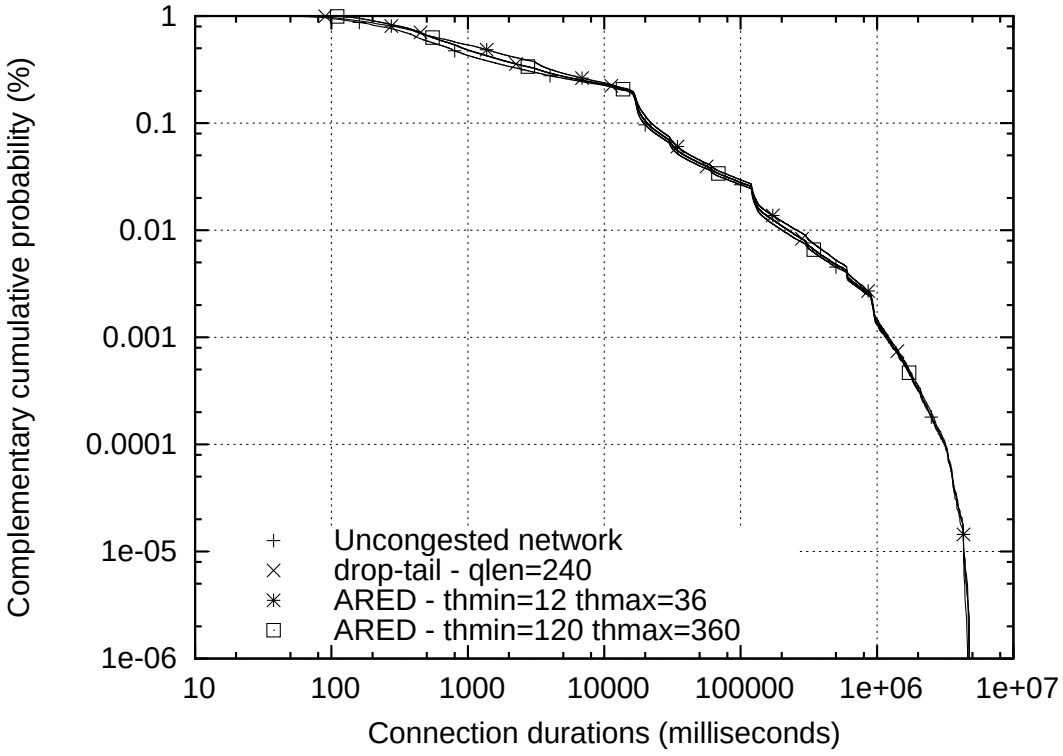

Figure 6.26: ARED performance at 95% load (CCDF)

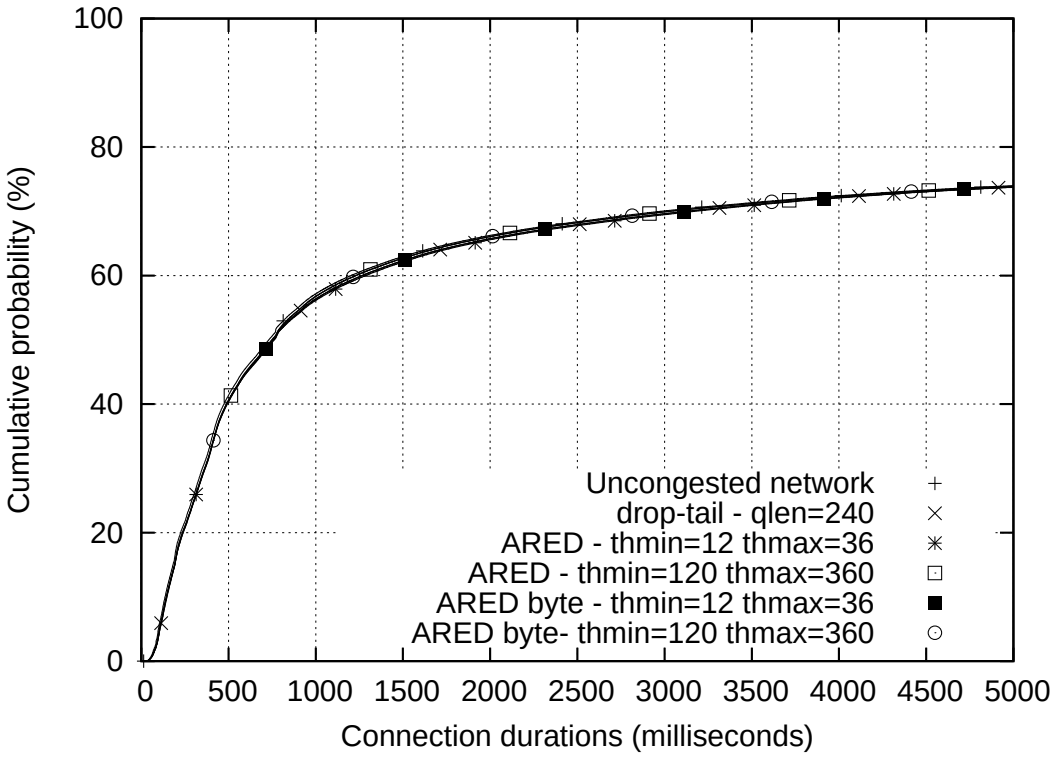

Figure 6.27: ARED byte mode performance at 80% load

rameter settings (th_{min} = 120 packets, th_{max} = 360 packets), the performance for ARED "packet mode" and "byte mode" was essentially the same.

At 95% load, ARED "byte mode" delivered slightly better performance with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) than with (th_{min} = 120 packets, th_{max} = 360 packets) for approximately 50% of flows that completed within 1 second or less. For the other 50% of flows, ARED "byte mode" gave slightly better performance with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) than with (th_{min} = 12 packets, th_{max} = 36 packets). Overall, ARED "packet mode" with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) significantly underperformed ARED "byte mode" with both parameter settings. However, drop-tail and ARED "packet mode" with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) only gave slightly worse performance than ARED "byte mode" with both parameter settings.

6.2.4 Results for LQD

Figures 6.33, 6.34, and 6.35 show experimental results for LQD with general TCP traffic when it was used with packet drops. These results were obtained with a queue reference of 24 and 240 packets.

At 80% offered load, LQD gave identical performance with both queue references of 24

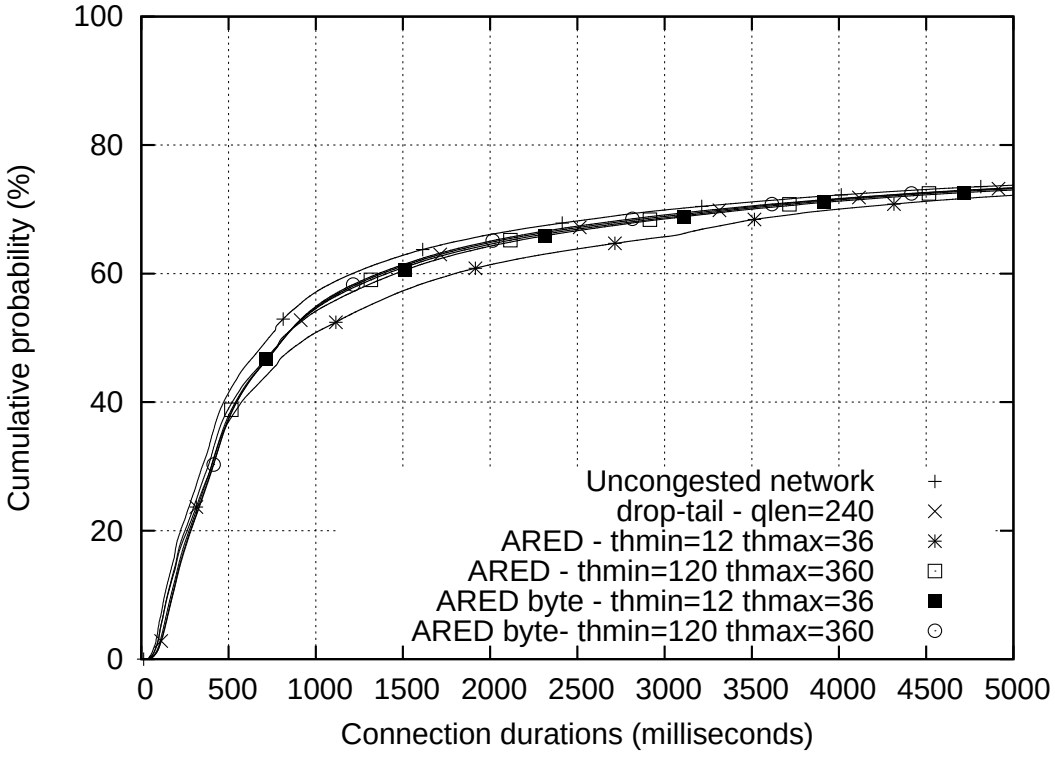

Figure 6.28: ARED byte mode performance at 90% load

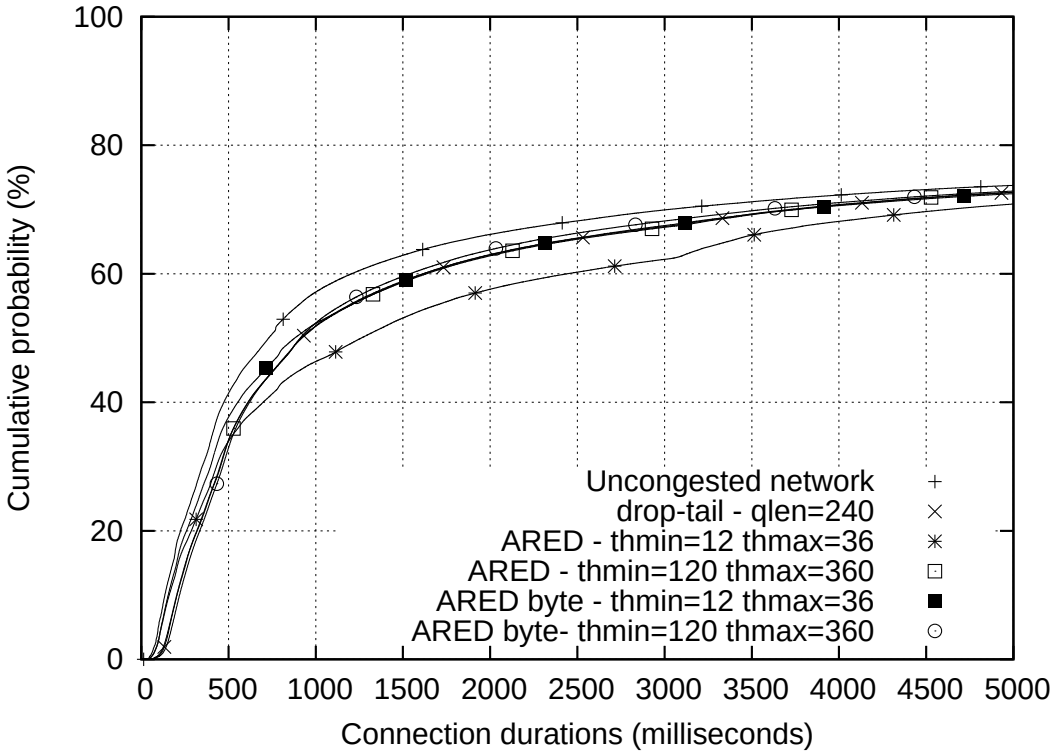

Figure 6.29: ARED byte mode performance at 95% load

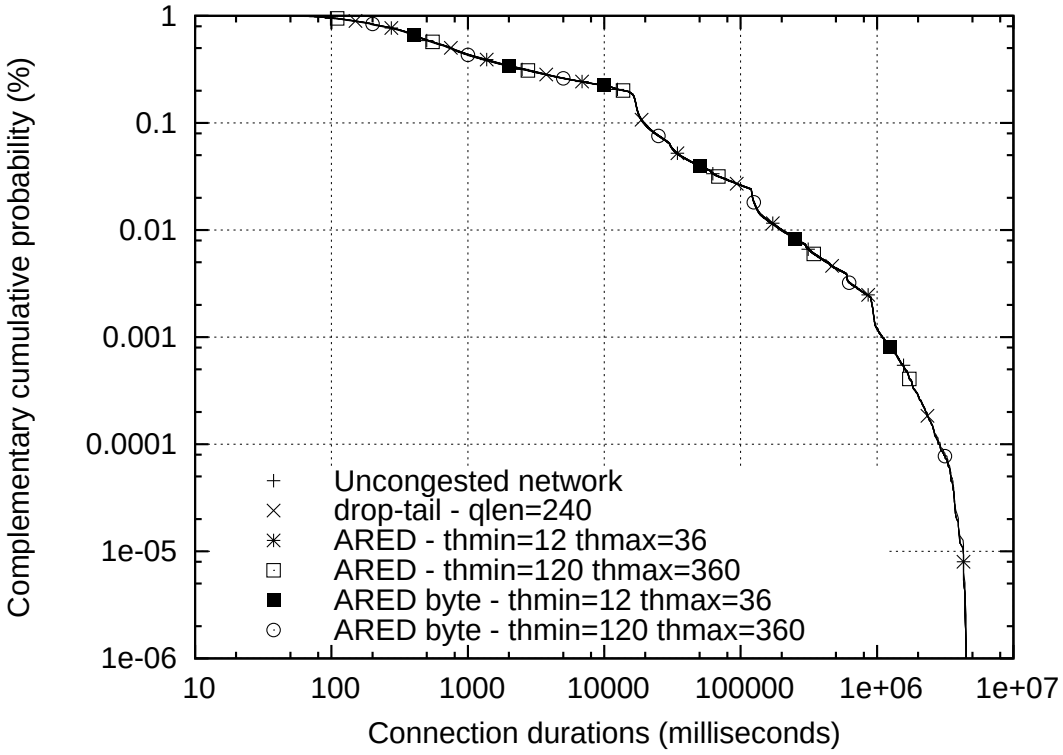

Figure 6.30: ARED byte mode performance at 80% load (CCDF)

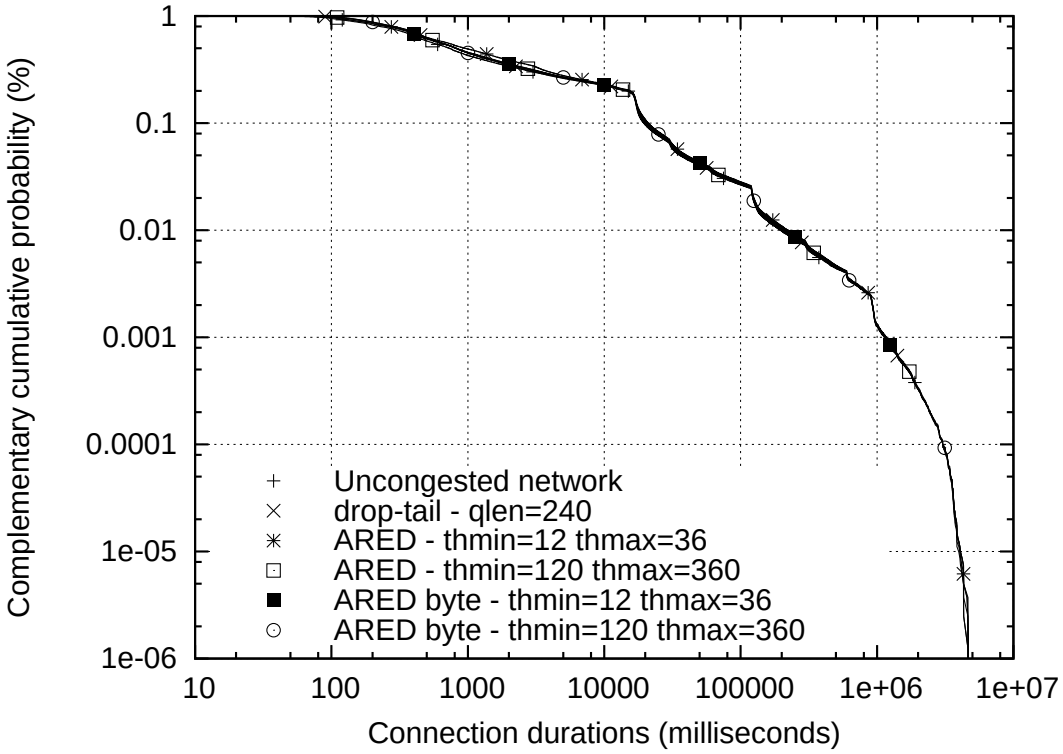

Figure 6.31: ARED byte mode performance at 90% load (CCDF)

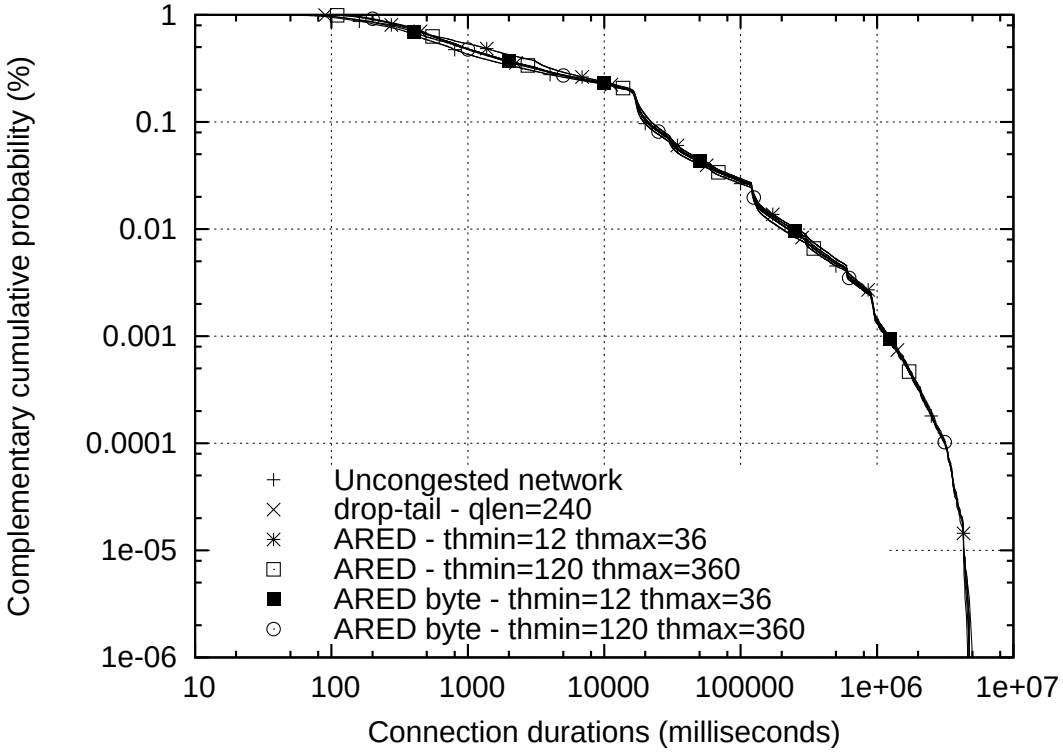

Figure 6.32: ARED byte mode performance at 95% load (CCDF)

and 240 packets. The performance for LQD at this load was essentially the same as that of drop-tail and the uncongested network.

As the offered load increased to 90%, LQD with a queue reference of 240 packets obtained identical performance as drop-tail. However, LQD with a queue reference of 24 packets gave slightly better performance than drop-tail and closely approximated the performance of the uncongested network.

At 95% offered load, LQD obtained slightly better performance for approximately 60% of flows with a queue reference of 24 packets than with a queue reference of 240 packets. LQD gave approximately the same performance for the other 40% of flows with both queue references. With both parameter settings, LQD outperformed drop-tail at this load and once again came close to the performance of the uncongested network.

6.2.5 Results for DCN

Figures depict experimental results for DCN with general TCP traffic when DCN was used with packet drops. These experimental results were obtained for DCN with a queue reference of 24 and 240 packets.

At 80% offered load, DCN gave similarly good performance with both queue references of 24 and 240 packets. The performance for DCN was identical to that of drop-tail and of

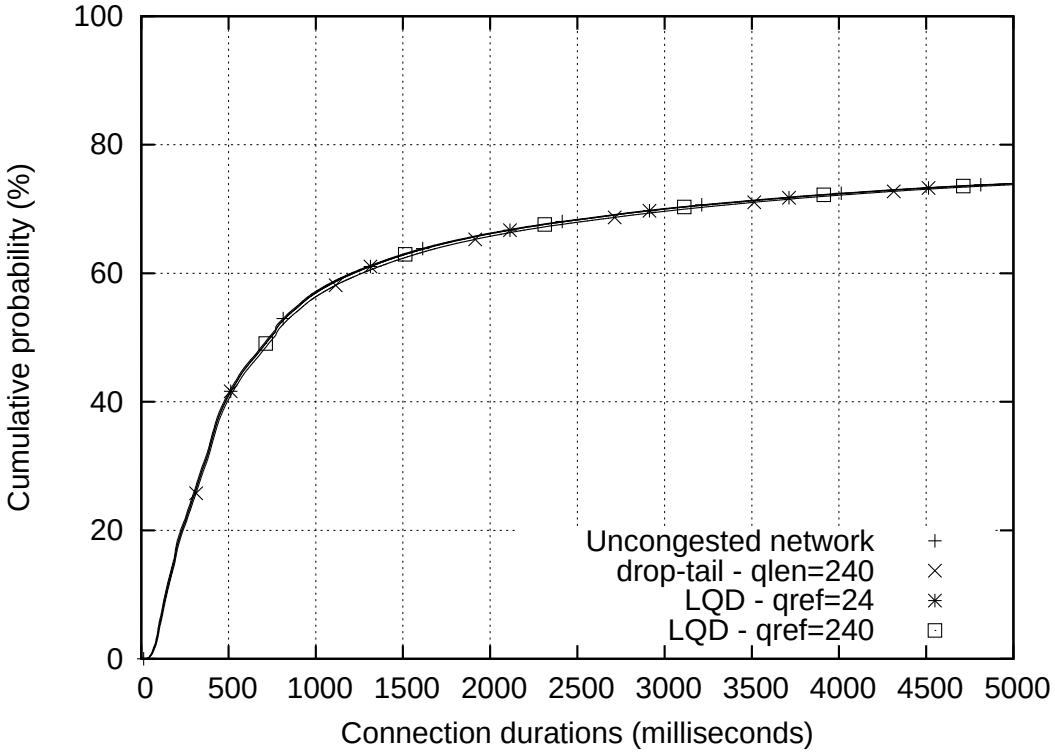

Figure 6.33: LQD performance at 80% load

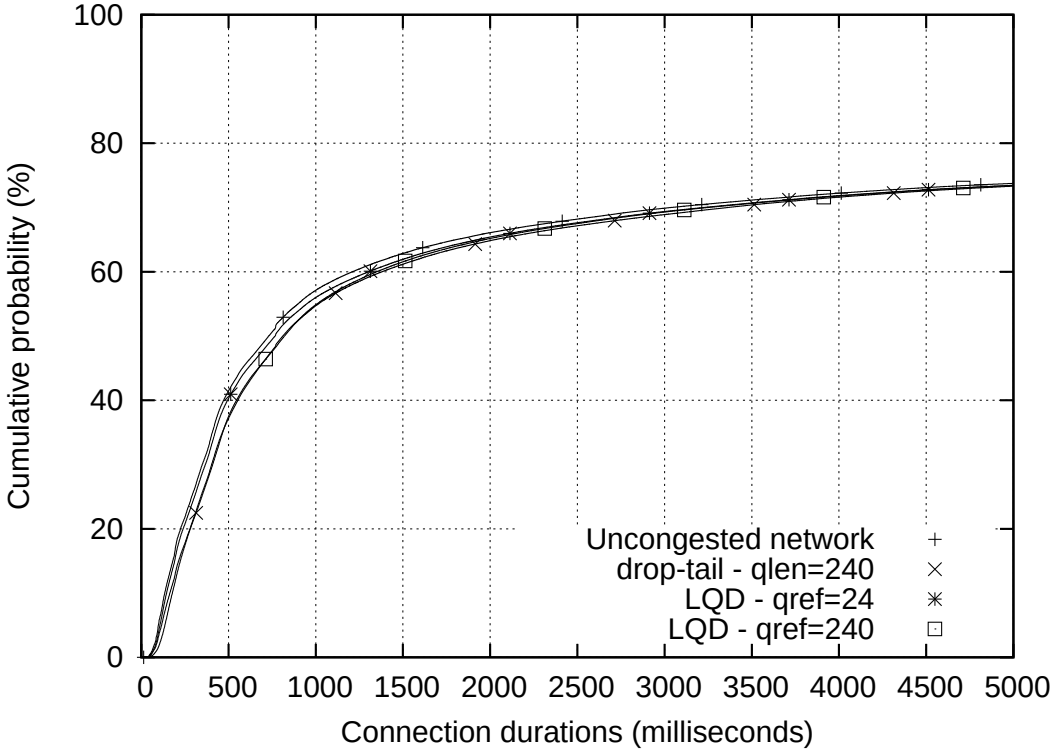

Figure 6.34: LQD performance at 90% load

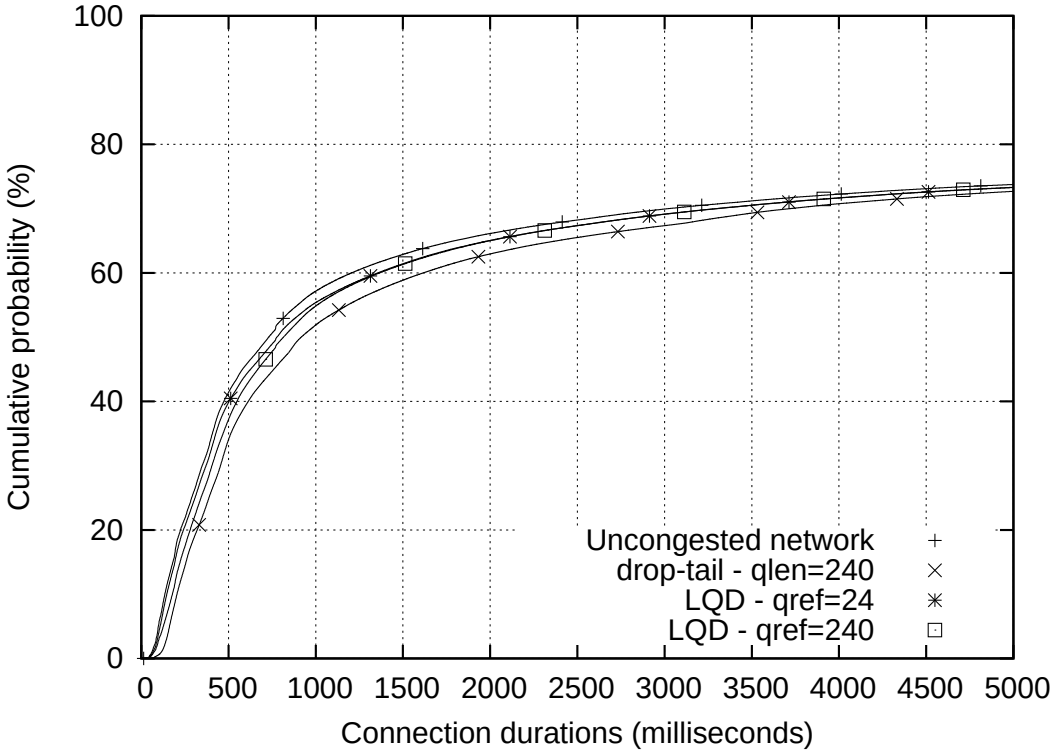

Figure 6.35: LQD performance at 95% load

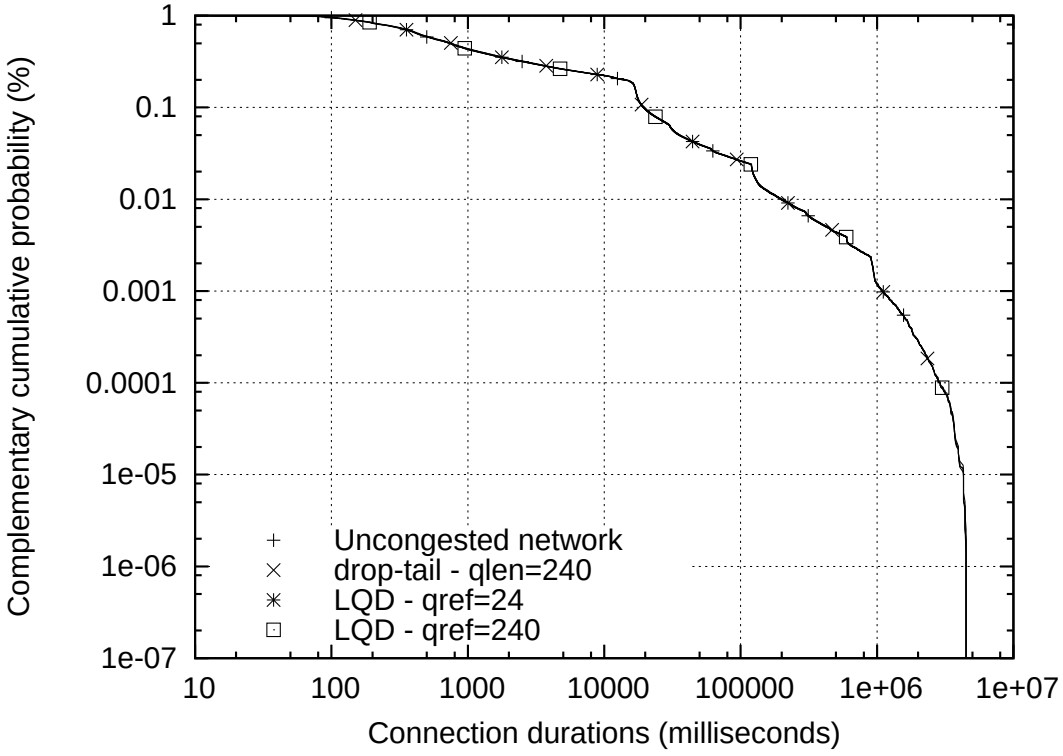

Figure 6.36: LQD performance at 80% load (CCDF)

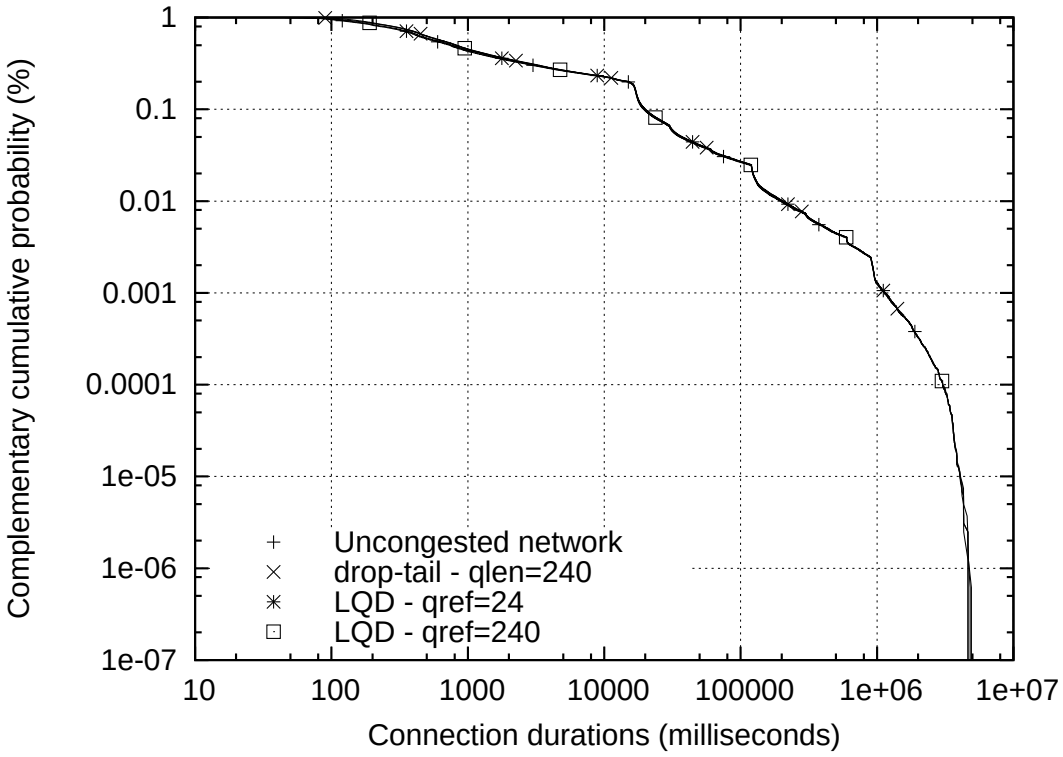

Figure 6.37: LQD performance at 90% load (CCDF)

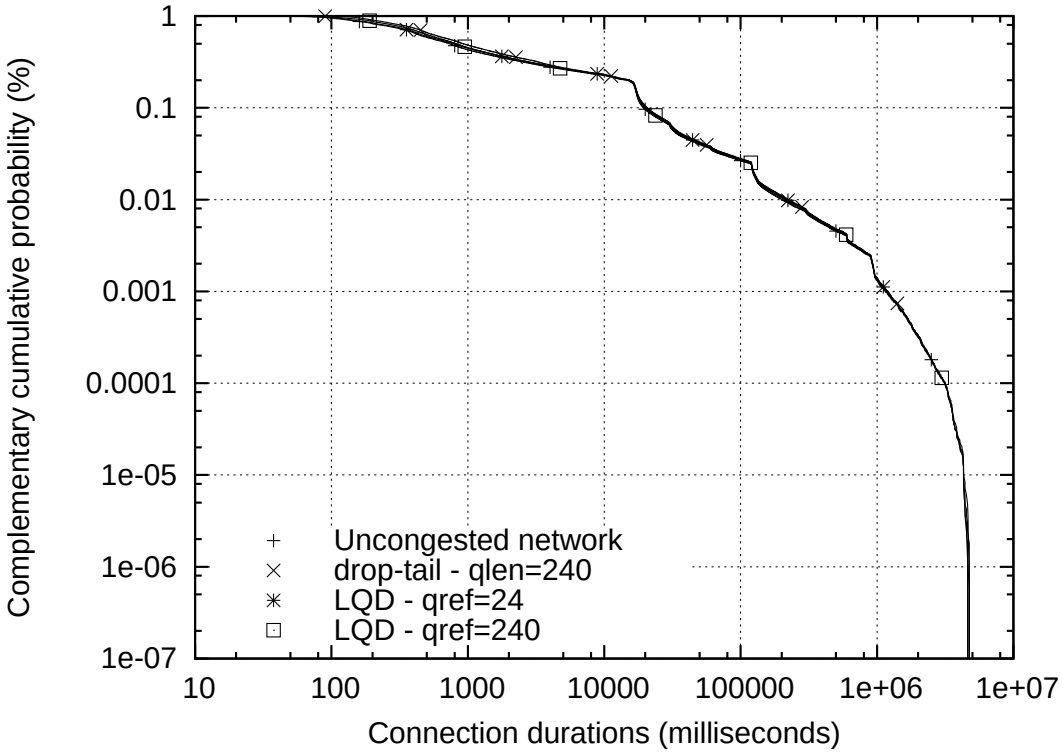

Figure 6.38: LQD performance at 95% load (CCDF)

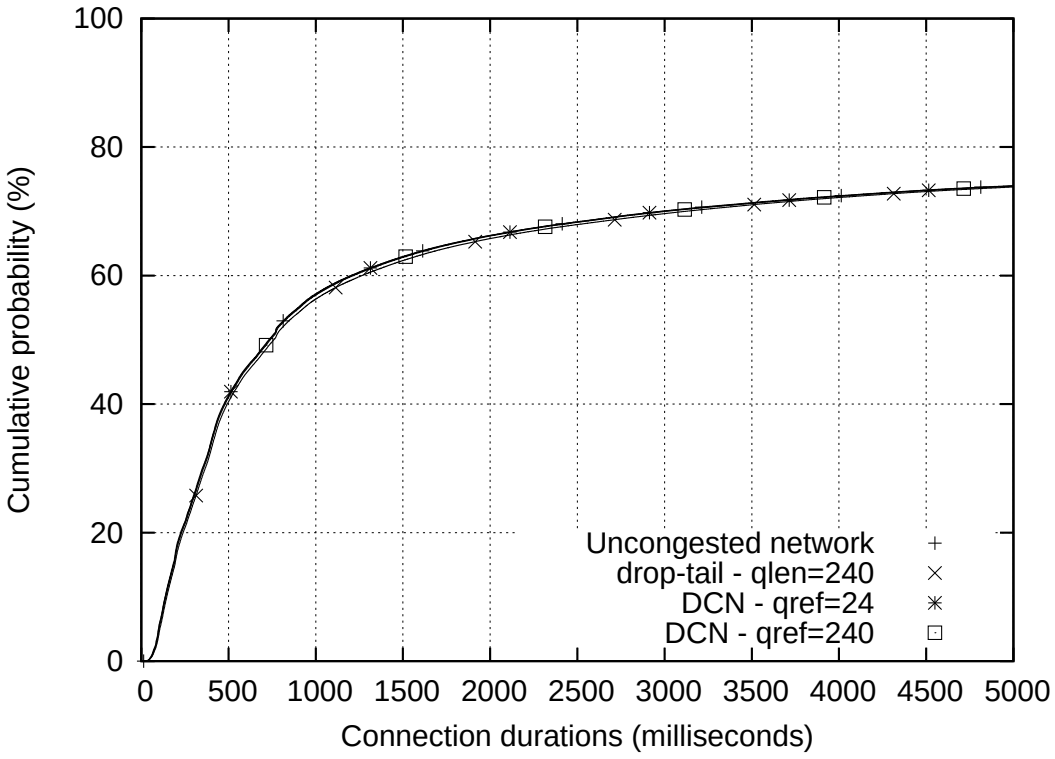

Figure 6.39: DCN performance at 80% load

the uncongested network at this load.

As the offered load increased to 90%, DCN obtained identical performance with both queue references. Further, DCN outperformed drop-tail and gave performance that was indistinguishable from that of the uncongested network.

At 95% offered load, DCN again obtained essentially the same performance with both queue references. DCN provided considerable performance improvement over drop-tail and gave identical performance as the uncongested network at this load. These results again demonstrated the benefits of differential treatment of flows in AQM algorithms.

6.3 Results for ARED, PI, LQD, and REM with ECN

Experimental results for AQM algorithms with general TCP applications presented in section 6.2 were obtained when AQM algorithms were used with packet drops. In order to evaluate the effects of the ECN signaling protocol on general TCP applications, experiments for AQM algorithms were also performed when ECN was turned on at both routers and end systems.

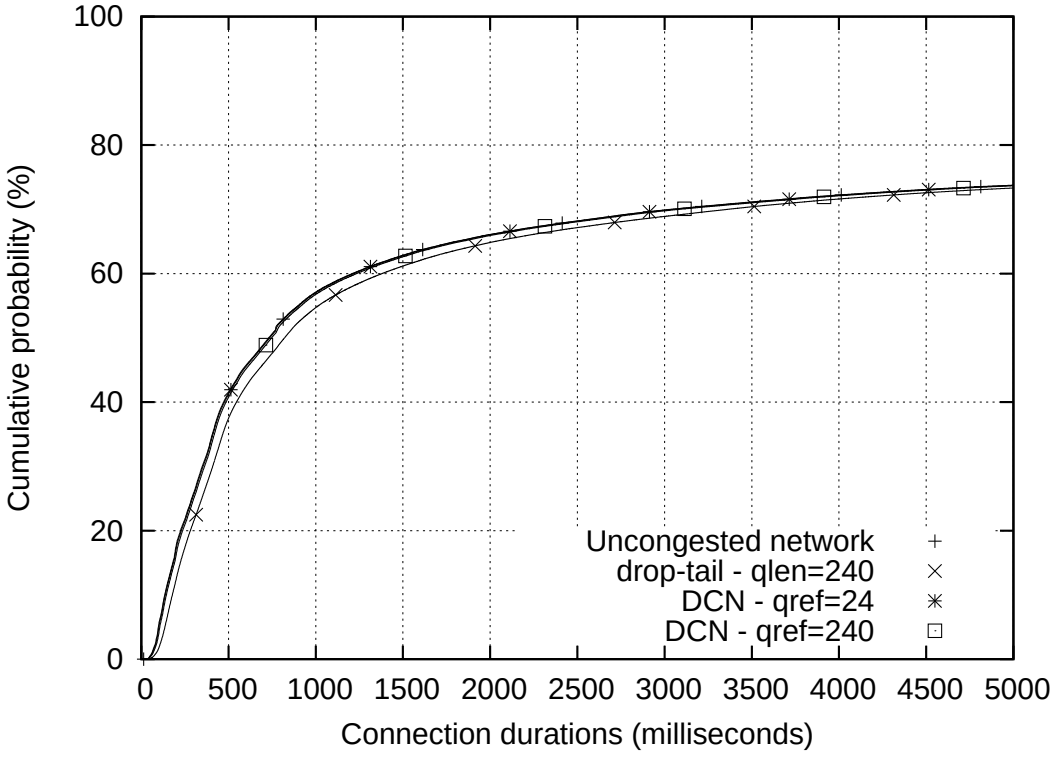

Figure 6.40: DCN performance at 90% load

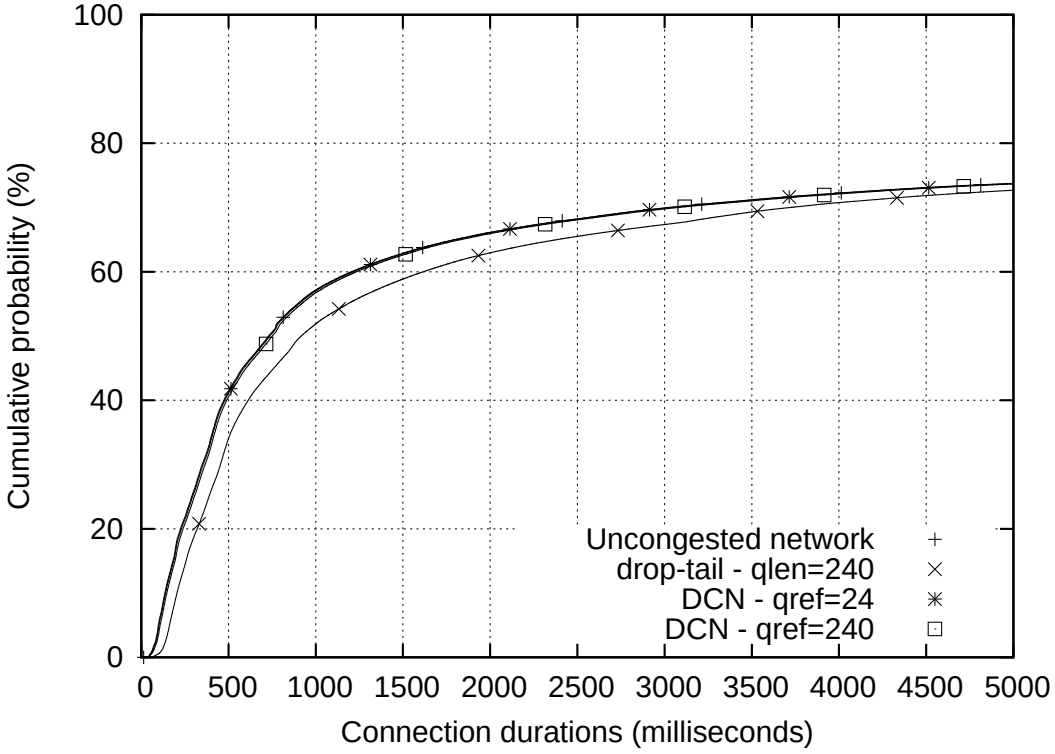

Figure 6.41: DCN performance at 95% load

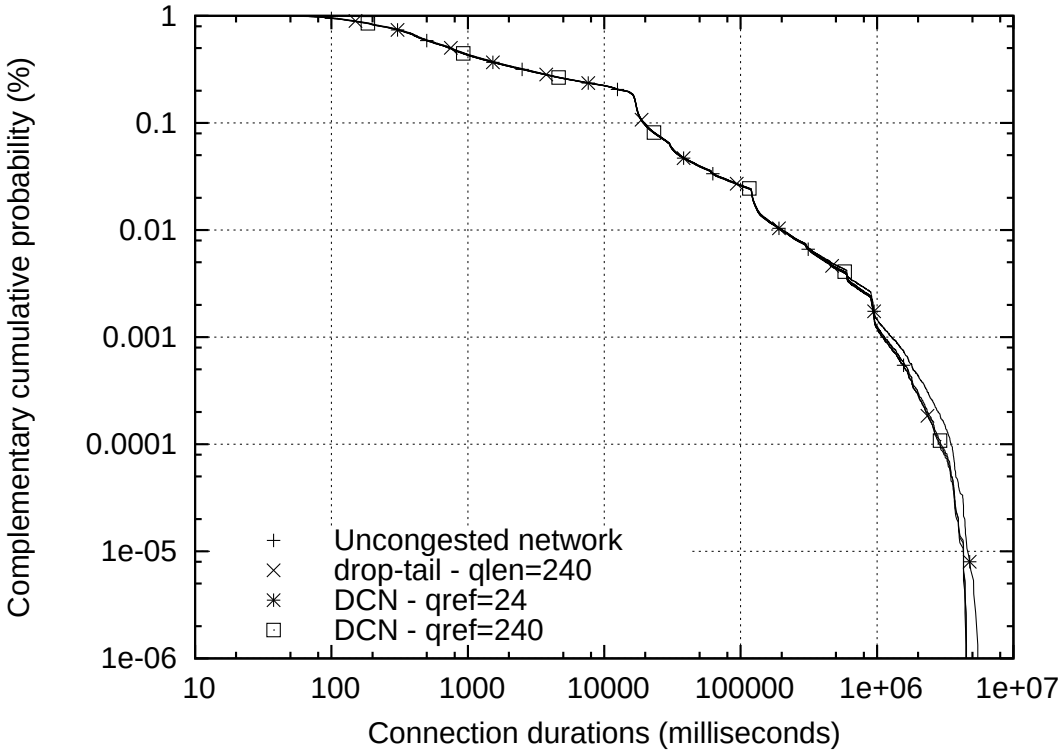

Figure 6.42: DCN performance at 80% load (CCDF)

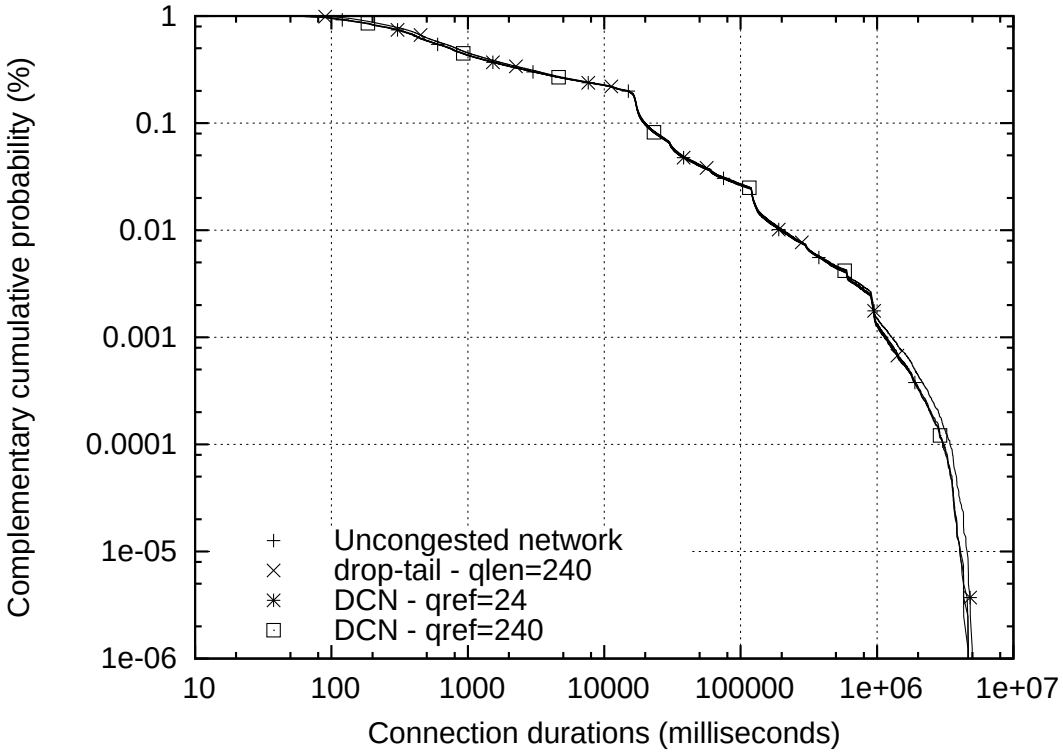

Figure 6.43: DCN performance at 90% load (CCDF)

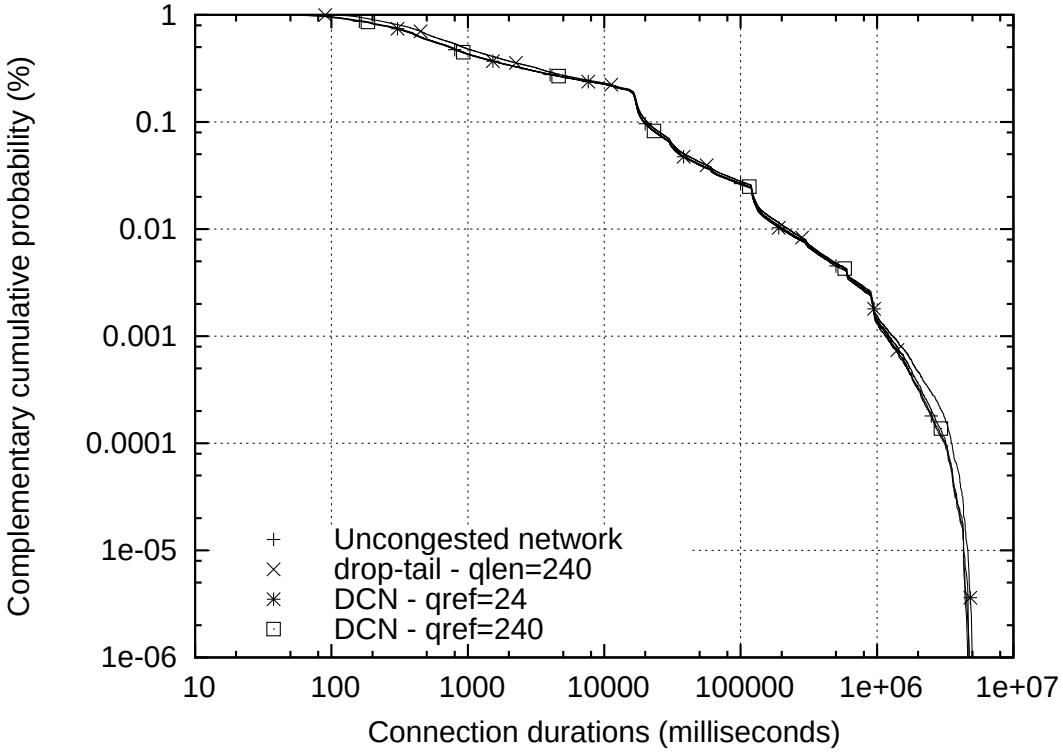

Figure 6.44: DCN performance at 95% load (CCDF)

6.3.1 Results for PI/ECN

Figures 6.45, 6.46, and 6.47 show experimental results for PI with general TCP traffic when PI was used with and without ECN. These results were obtained when PI operated with a queue reference of 24 and 240 packets.

At 80% offered load, ECN did not deliver any performance improvement for PI with both queue references of 24 and 240 packets at all. This is because PI already gave indistinguishable performance from the uncongested network at this load.

At 90% load, PI did not obtain any performance improvement with ECN when PI was used with a queue reference of 24 packets. With and without ECN, PI with a queue reference of 24 packets delivered approximately the same performance as drop-tail and came close to that of the uncongested network. However, ECN improved the performance for PI with a queue reference of 240 packets slightly. Without ECN, PI with a queue reference of 240 packets underperformed drop-tail slightly. When used with ECN, PI with a queue reference of 240 packets gave identical performance as drop-tail.

At 95% offered load, ECN gave a slight performance improvement for PI with a queue reference of 24 packets. However, the performance for PI with a queue reference of 240 packets was not improve by the addition of ECN at all. With or without ECN, PI gave slightly

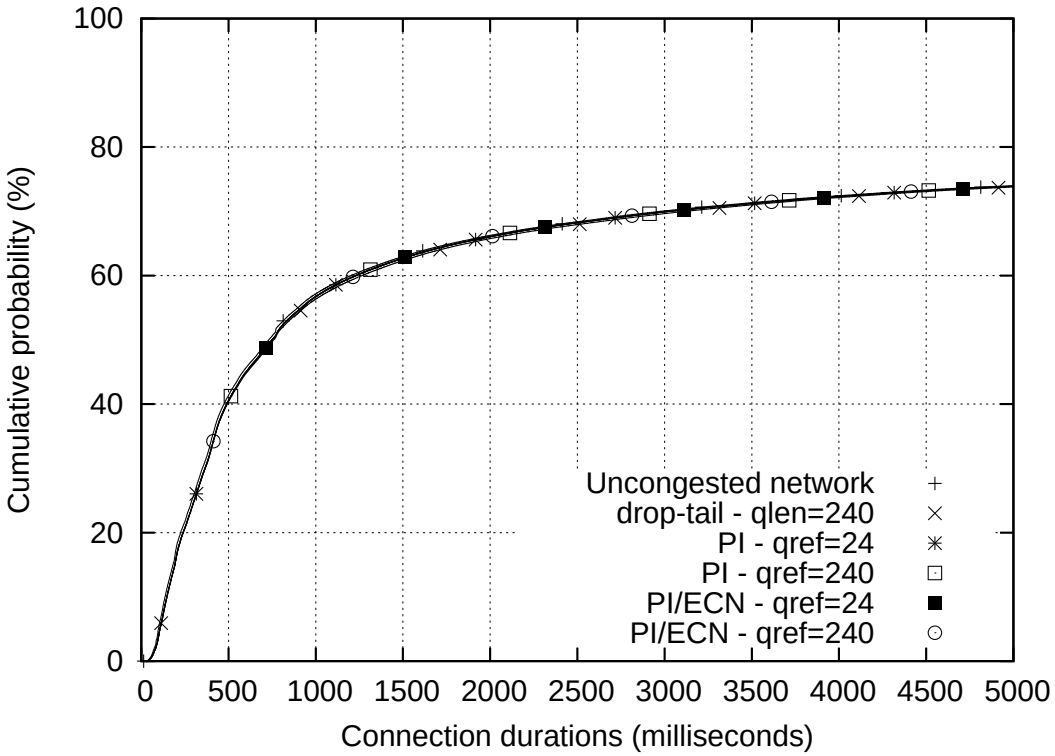

Figure 6.45: PI/ECN performance at 80% load

better performance than drop-tail with both queue references and closely approximated the performance of the uncongested network.

6.3.2 Results for REM/ECN

Figures 6.51, 6.52, and 6.53 depict experimental results for REM with general TCP applications when REM was used with or without ECN at 80%, 90%, and 95% loads. These results were obtained with a queue reference of 24 and 240 packets.

At 80% load, REM with ECN did not gain any performance improvement over packet drops when it was used with a queue reference of 24 and 240 packets. With or without ECN, the performance for REM was equal to that of drop-tail and uncongested network at this load.

At 90% load, REM also did not benefit from the ECN signaling protocol. This is because the performance for REM without ECN for both queue references was already very good and closely approximated the performance of the uncongested network at this load. The performance for REM (with or without ECN) was about the same as that of drop-tail.

At 95% offered load, REM obtained a small performance improvement over packet drops with both queue references. With both queue references, REM without ECN only showed a small advantage over drop-tail. However, when REM was used in combination with ECN,

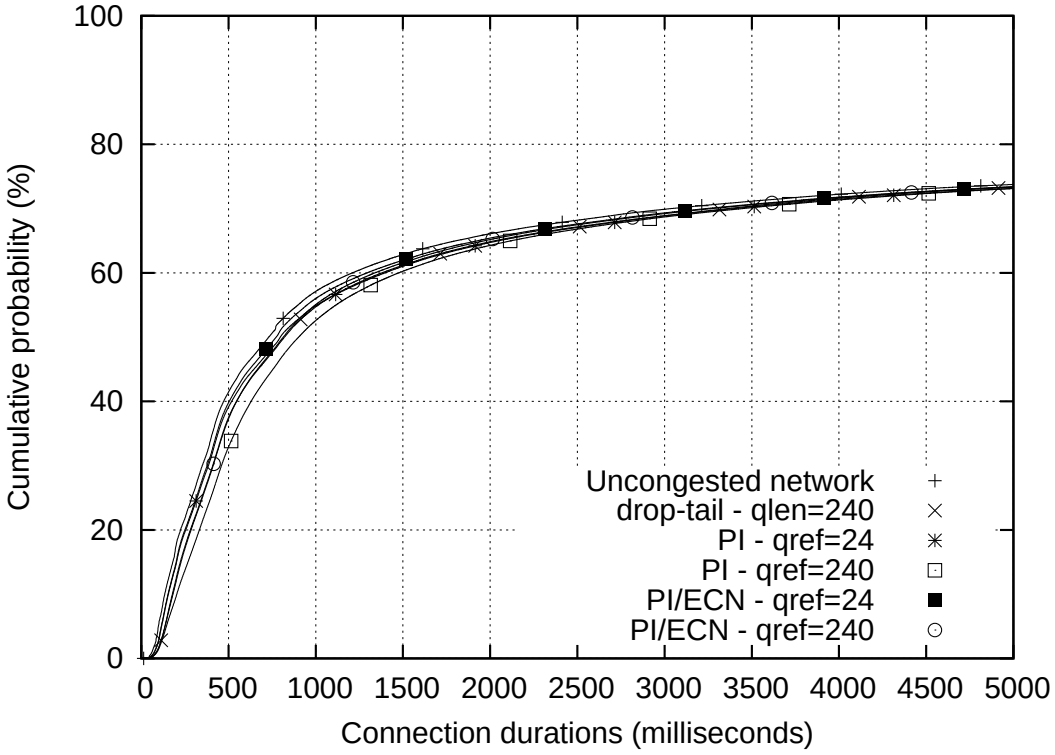

Figure 6.46: PI/ECN performance at 90% load

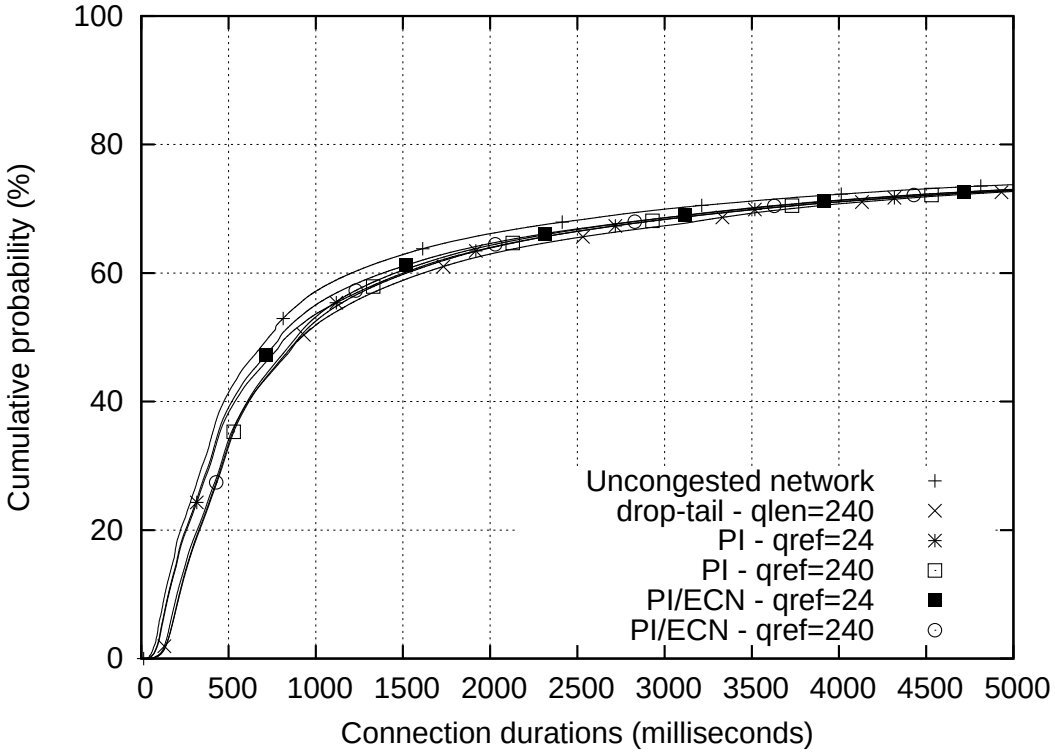

Figure 6.47: PI/ECN performance at 95% load

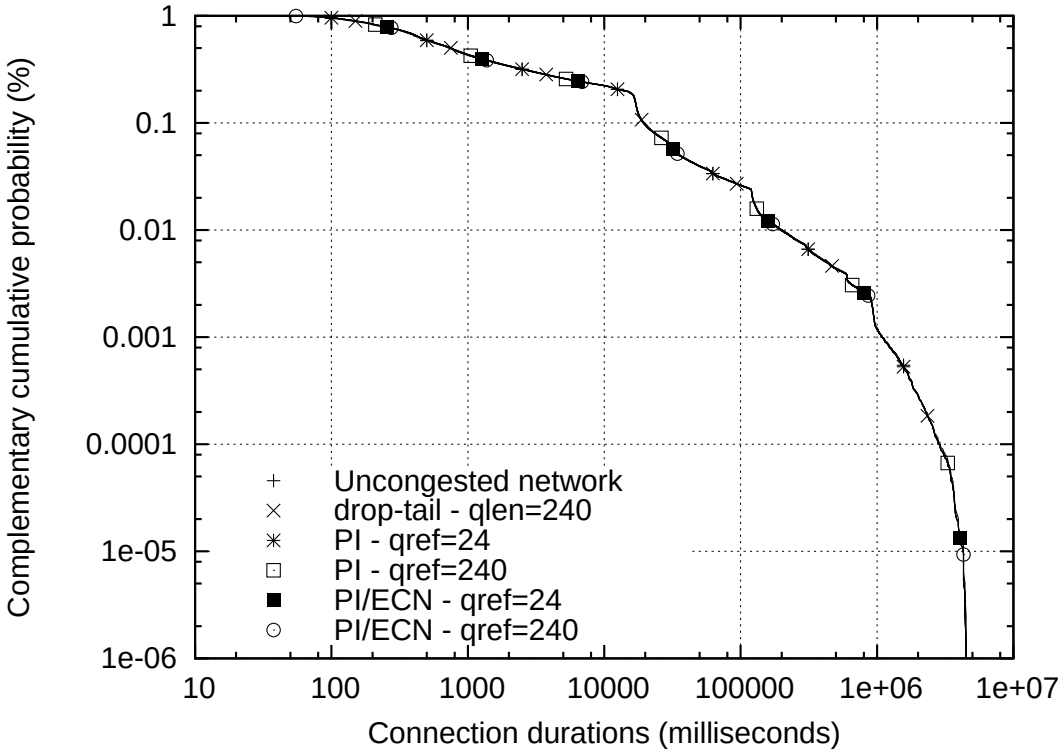

Figure 6.48: PI/ECN performance at 80% load (CCDF)

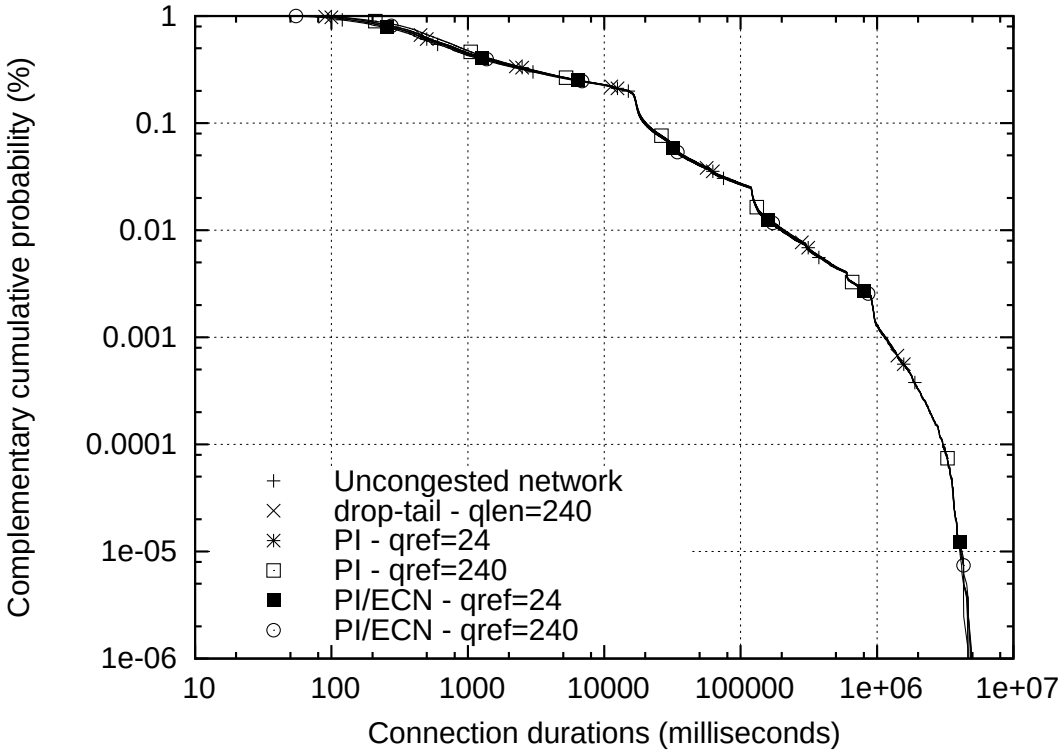

Figure 6.49: PI/ECN performance at 90% load (CCDF)

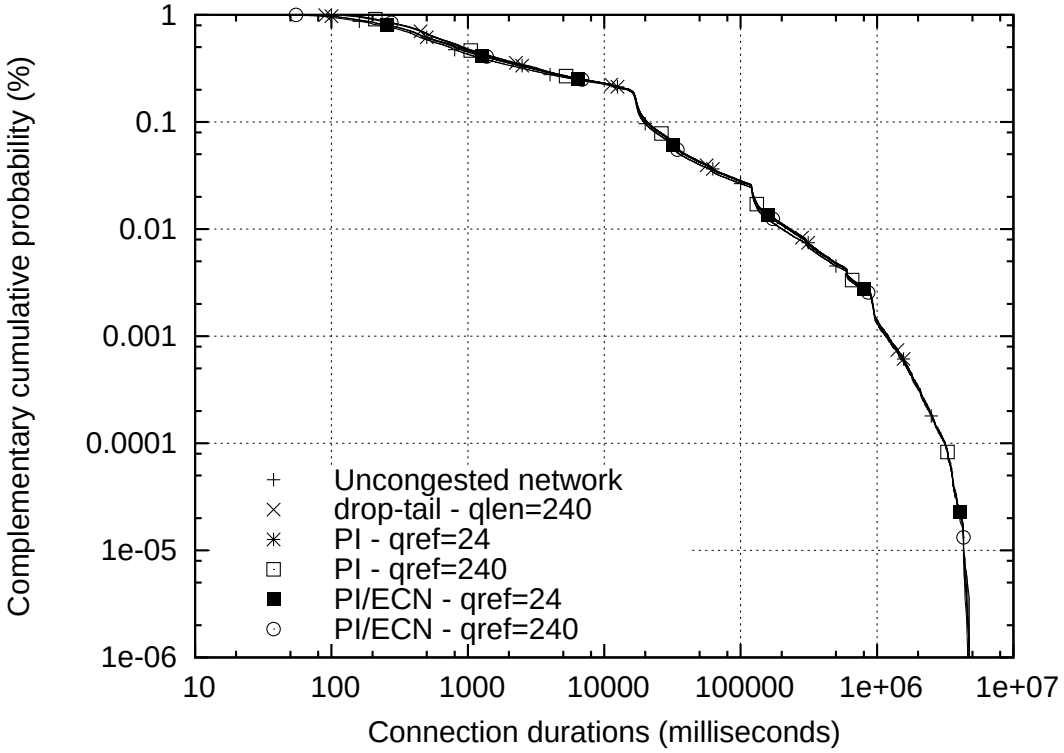

Figure 6.50: PI/ECN performance at 95% load (CCDF)

REM with both queue references clearly outperformed drop-tail and came close to the performance of the uncongested network. This result once again demonstrated the benefits of the ECN marking protocol. This is because ECN helps AQM algorithms avoid packet drops and thus improves network and application performance.

6.3.3 Results for ARED/ECN

Figures 6.57, 6.58, and 6.59 show results for ARED with and without ECN. These results were obtained with general TCP applications and with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) for ARED. These parameters were chosen to achieve an implicit target queue reference of 24 and 240 packets respectively. Further, figures 6.63, 6.64, and 6.65 show results for ARED when it operated using the "new gentle" mode. The results for ARED "new gentle" algorithm demonstrate the effects of dropping packets in ECN mode.

At 80% offered load, ARED with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) did not benefit from the ECN marking protocol. This is because ARED with both parameter settings already delivered performance identical with that of the uncongested network at this load. With or without ECN, ARED with both parameter settings gave performance that was indistinguishable

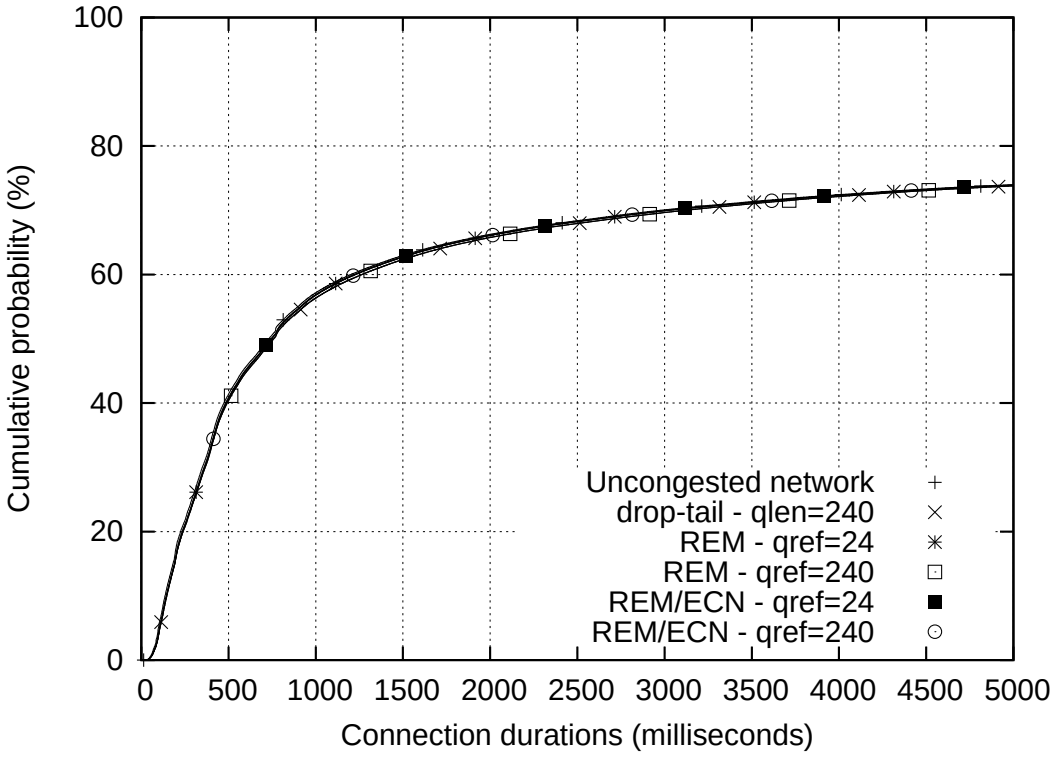

Figure 6.51: REM/ECN performance at 80% load

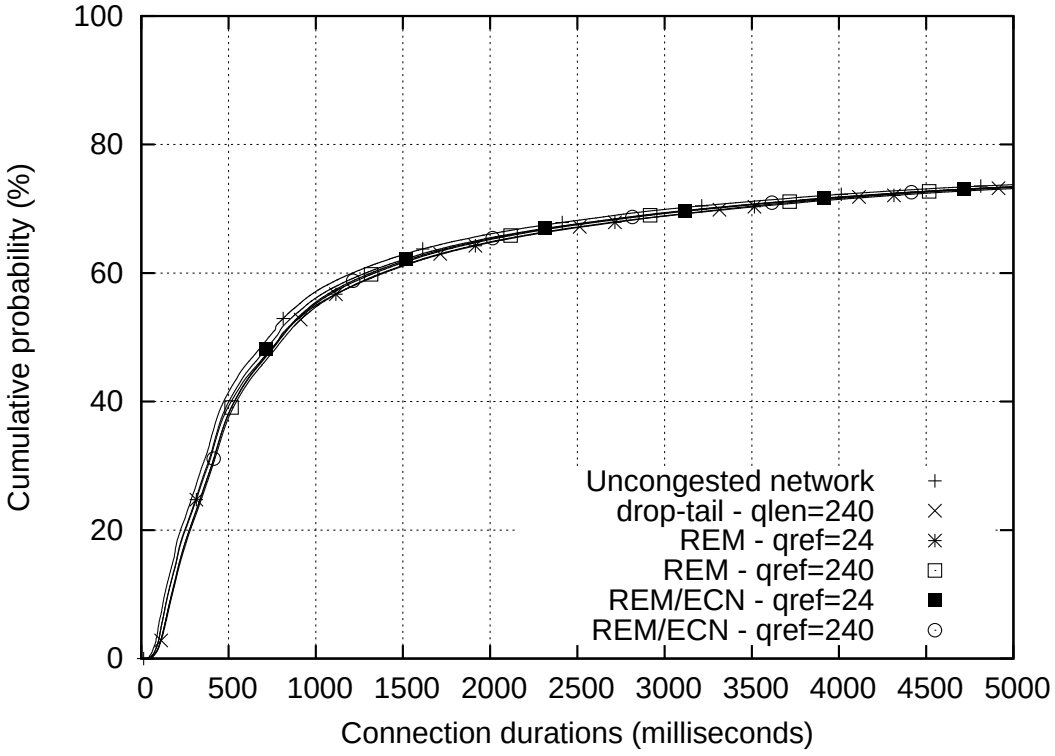

Figure 6.52: REM/ECN performance at 90% load

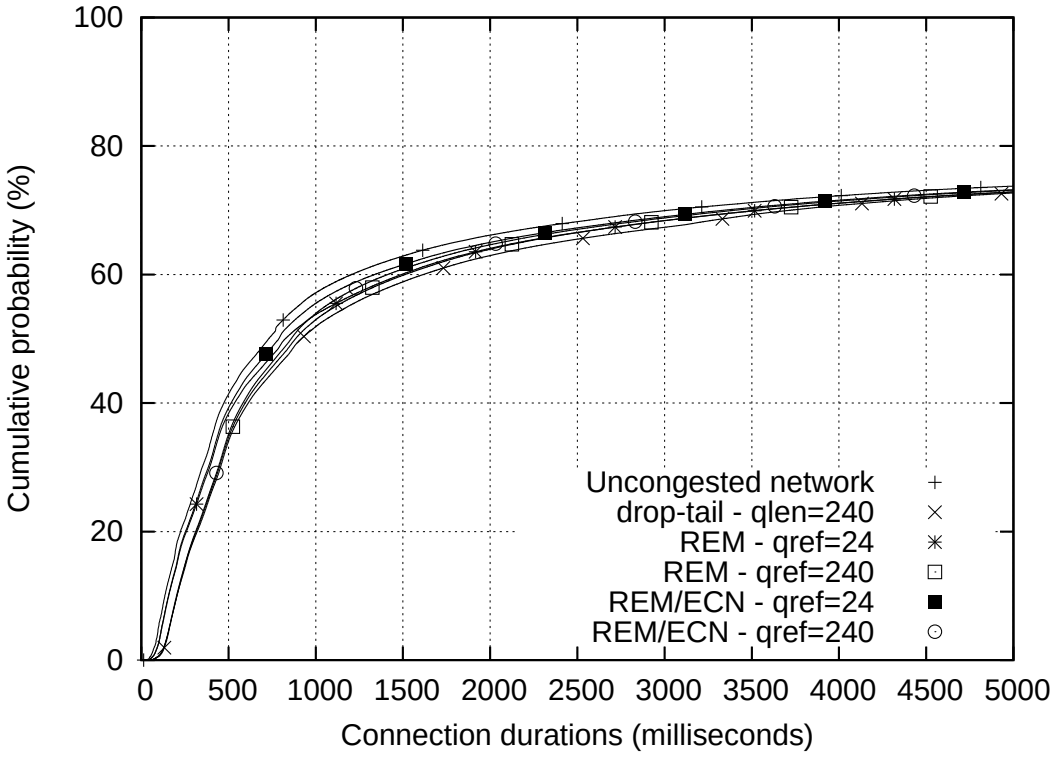

Figure 6.53: REM/ECN performance at 95% load

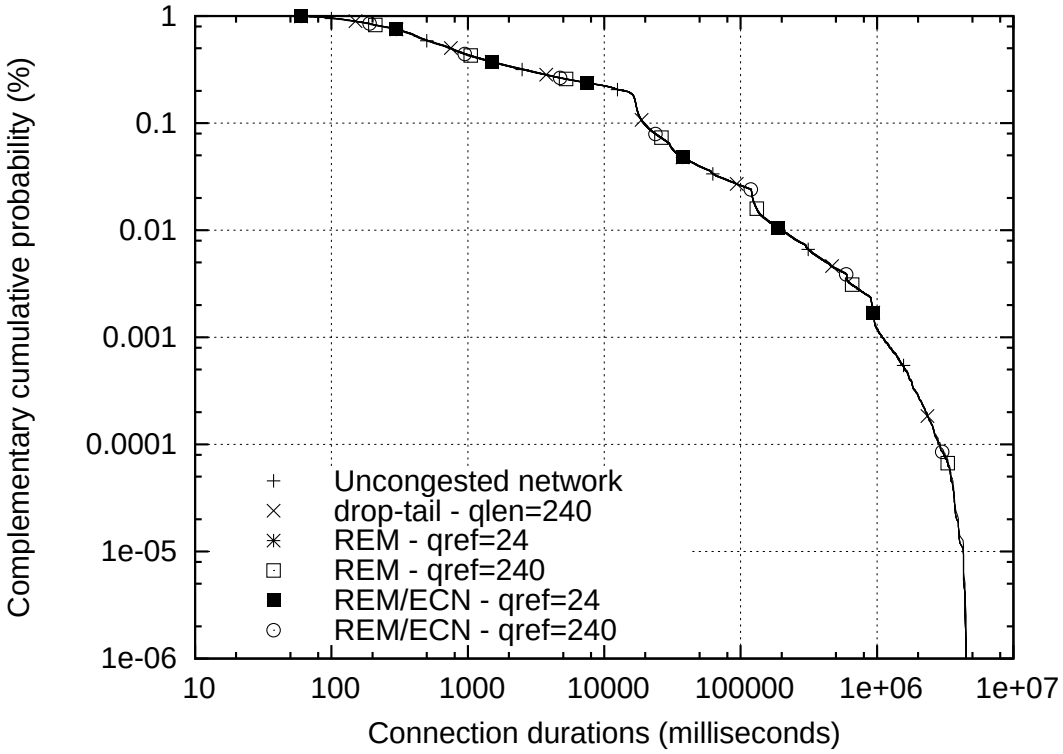

Figure 6.54: REM/ECN performance at 80% load (CCDF)

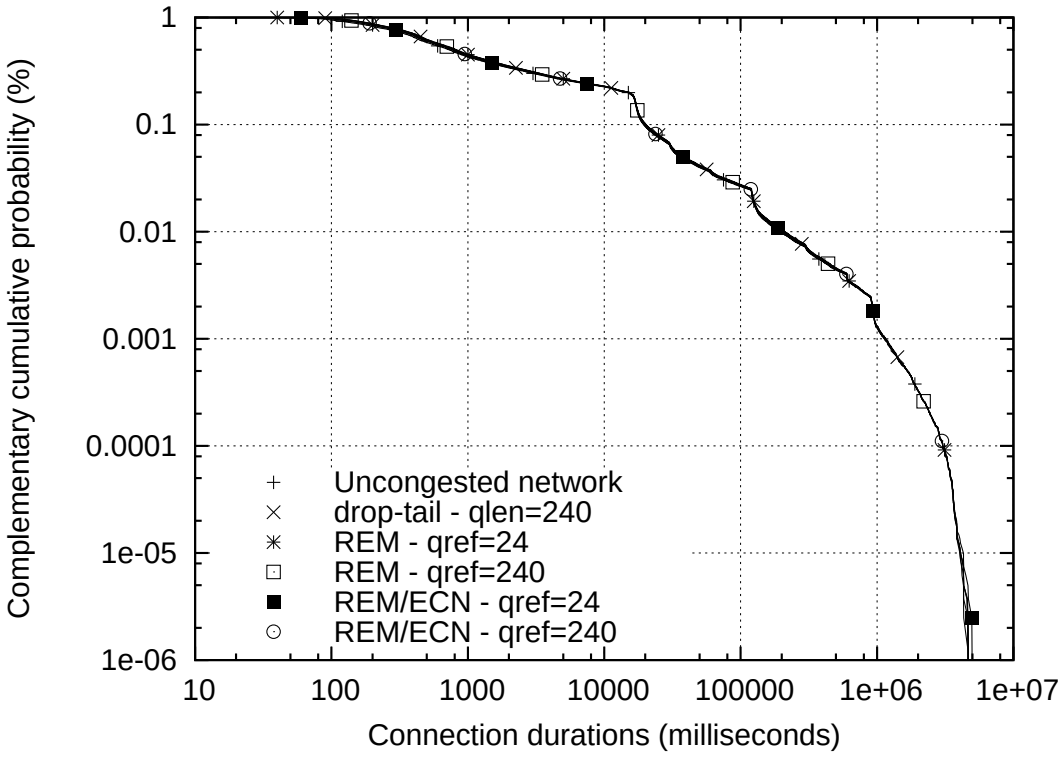

Figure 6.55: REM/ECN performance at 90% load (CCDF)

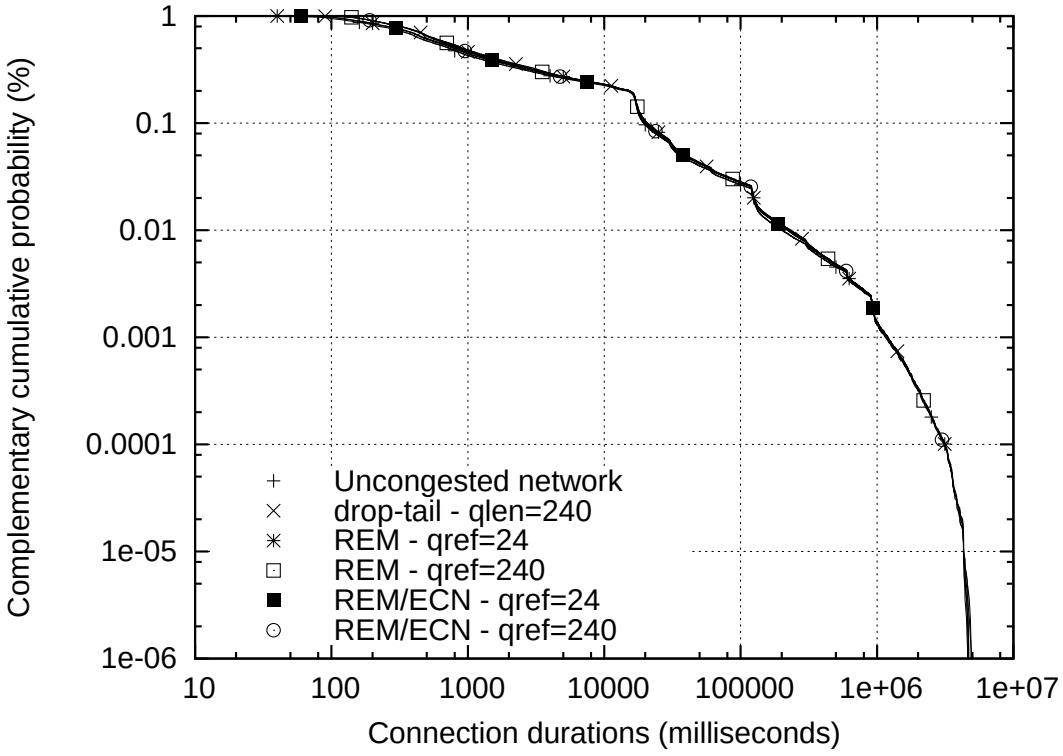

Figure 6.56: REM/ECN performance at 95% load (CCDF)

from that of drop-tail and of the uncongested network.

At 90% offered load, ARED with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) also did not obtain any performance improvement over packet drops when it was used with ECN. With or without ECN, ARED with the parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) delivered the same performance as drop-tail and came close to the performance of the uncongested network. However, when ARED was used with the parameter settings (th_{min} = 12 packets, th_{max} = 36 packets), it delivered equally poor performance both with and without ECN. ARED with the parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) suffered considerable performance degradation from the uncongested network and underperformed drop-tail noticeably.

As the offered load increased to 95%, ARED with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) once again did not gain any performance improvement from the addition of ECN. When ARED was used with the parameter settings (th_{min} = 120 packets, th_{max} = 360 packets), it delivered about the same performance as drop-tail both with and without ECN and showed noticeable performance degradation from the uncongested network. When used with the parameter settings (th_{min} = 12 packets, th_{max} = 36 packets), ARED with ECN surprisingly gave slightly poorer performance for about 55% of flows. These were flows that needed more than 1 second to complete. With or without ECN, ARED with the parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) underperformed drop-tail considerably.

Figure 6.63 shows a comparison of the original ARED algorithm and the ARED "new gentle" algorithm at 80% load when they were used with ECN. At this load, the ARED "new gentle" algorithm did not show any improvement over the original ARED algorithm. When used in combination with ECN, both the original ARED algorithm and the ARED "new gentle" algorithm delivered essentially the same performance as drop-tail and the uncongested network with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets).

At 90% offered load, both ARED/ECN and ARED/ECN "new gentle" gave identical performance as drop-tail with the parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) and closely approximated the performance of the uncongested network. However, when used with the parameter settings (th_{min} = 12 packets, th_{max} = 36 packets), ARED/ECN "new gentle" obtained considerable performance improvement over ARED/ECN. Nevertheless, ARED/ECN "new gentle" with the parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) delivered slightly worse performance than drop-tail for approximately 40% of flows that needed more than 1.5 seconds to complete.

At 95% offered load, both ARED/ECN and ARED/ECN "new gentle" again obtained essentially the same performance as drop-tail with the parameter settings (th_{min} = 120

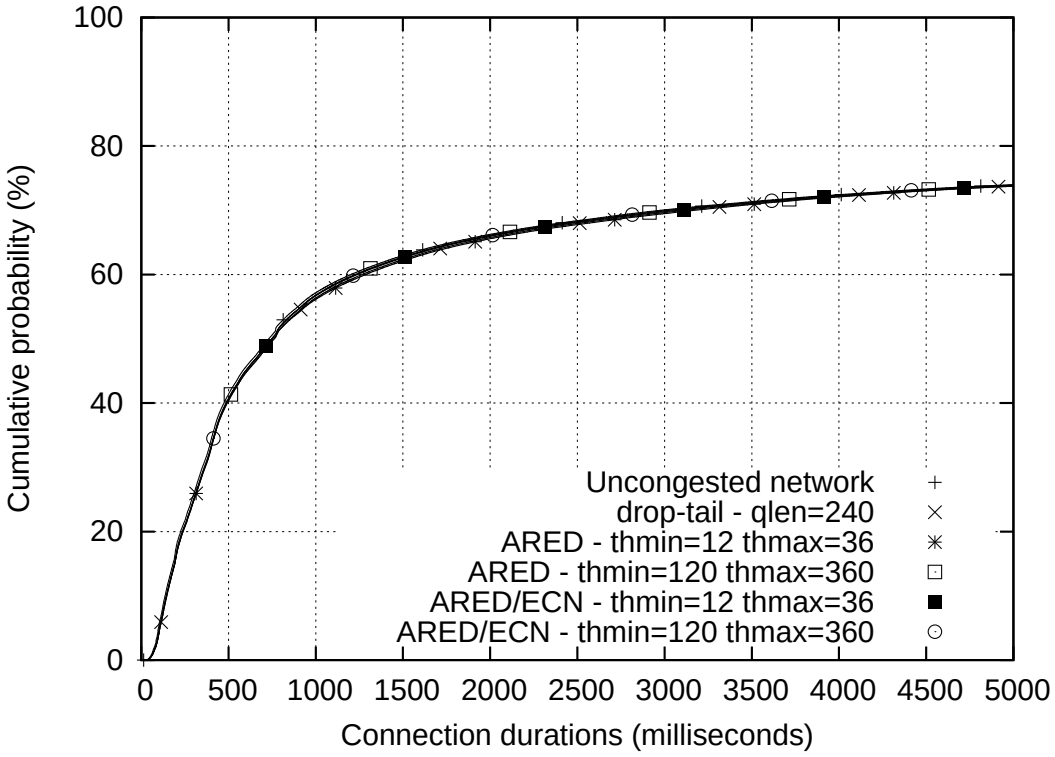

Figure 6.57: ARED/ECN performance at 80% load

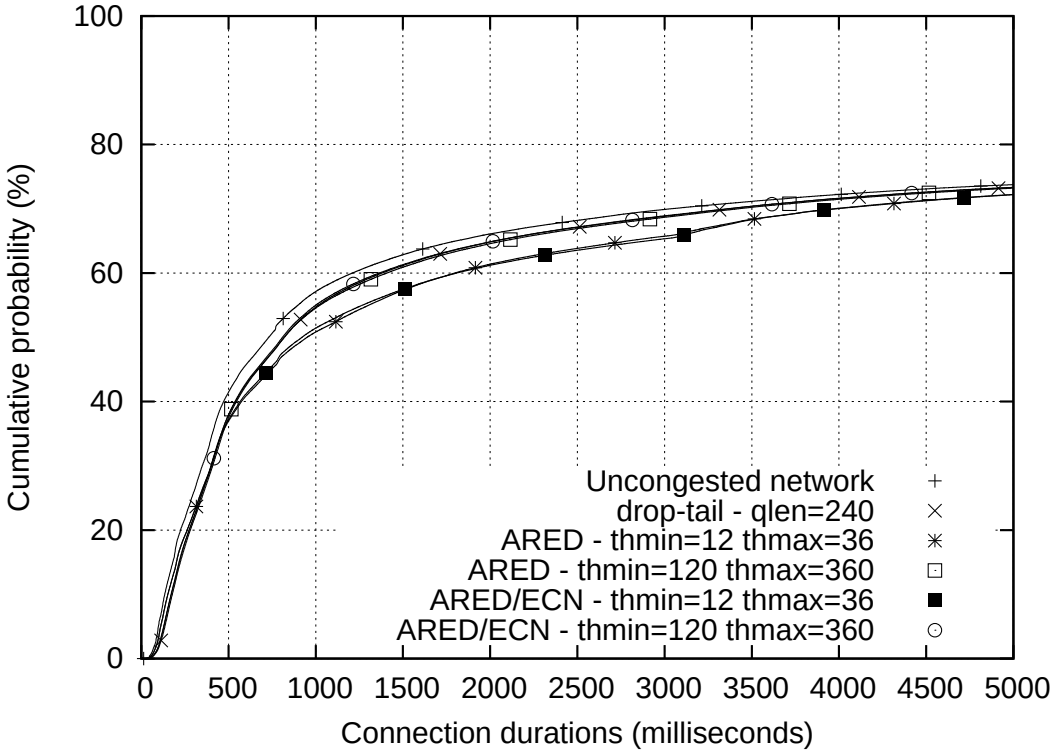

Figure 6.58: ARED/ECN performance at 90% load

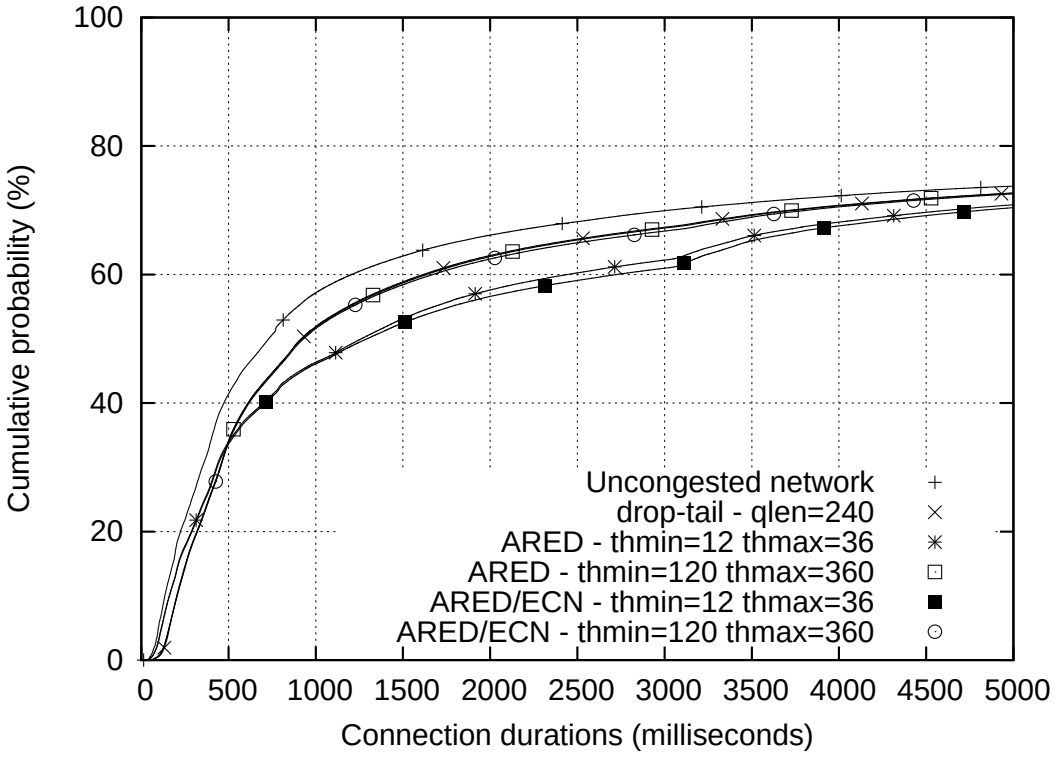

Figure 6.59: ARED/ECN performance at 95% load

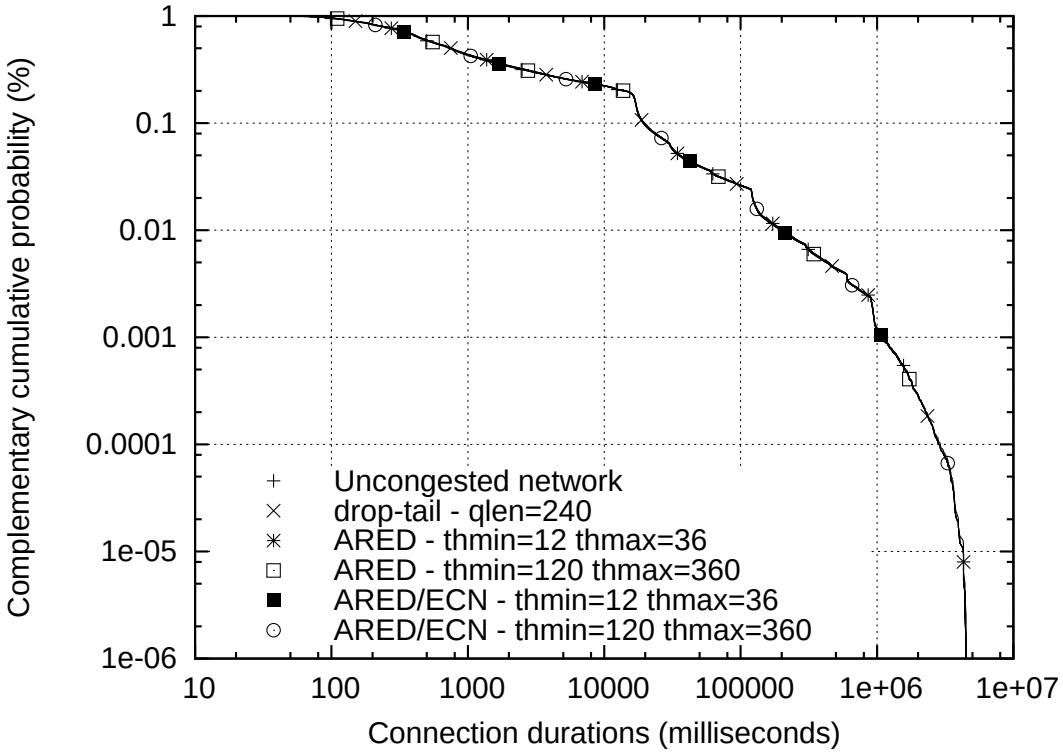

Figure 6.60: ARED/ECN performance at 80% load (CCDF)

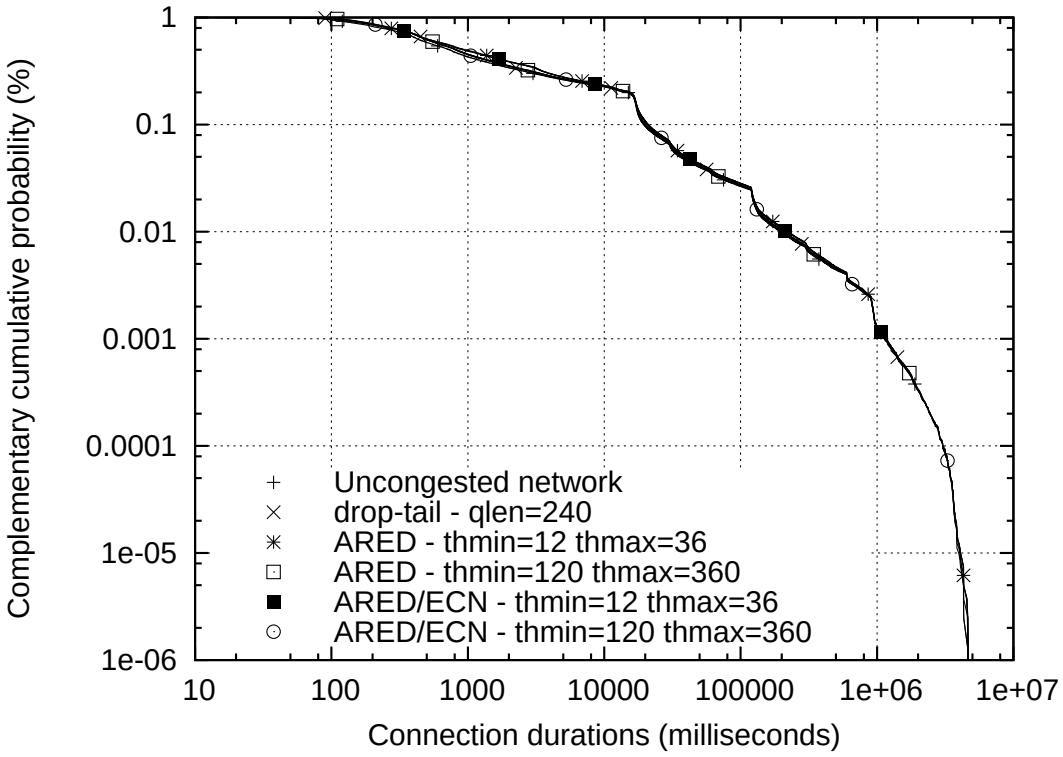

Figure 6.61: ARED/ECN performance at 90% load (CCDF)

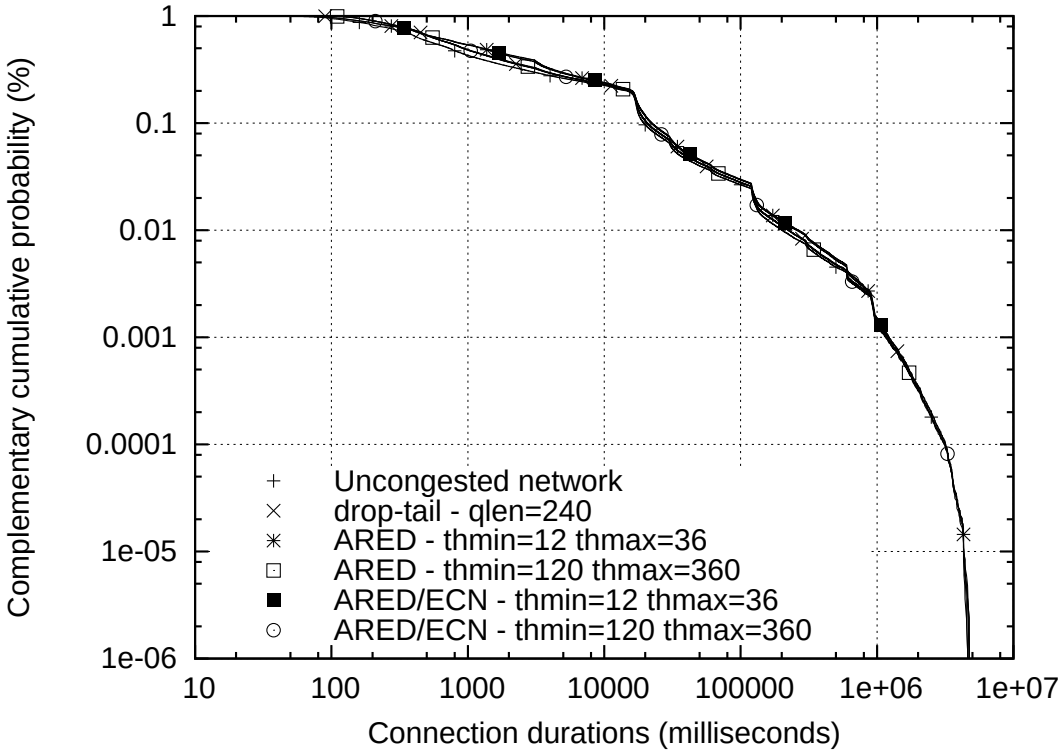

Figure 6.62: ARED/ECN performance at 95% load (CCDF)

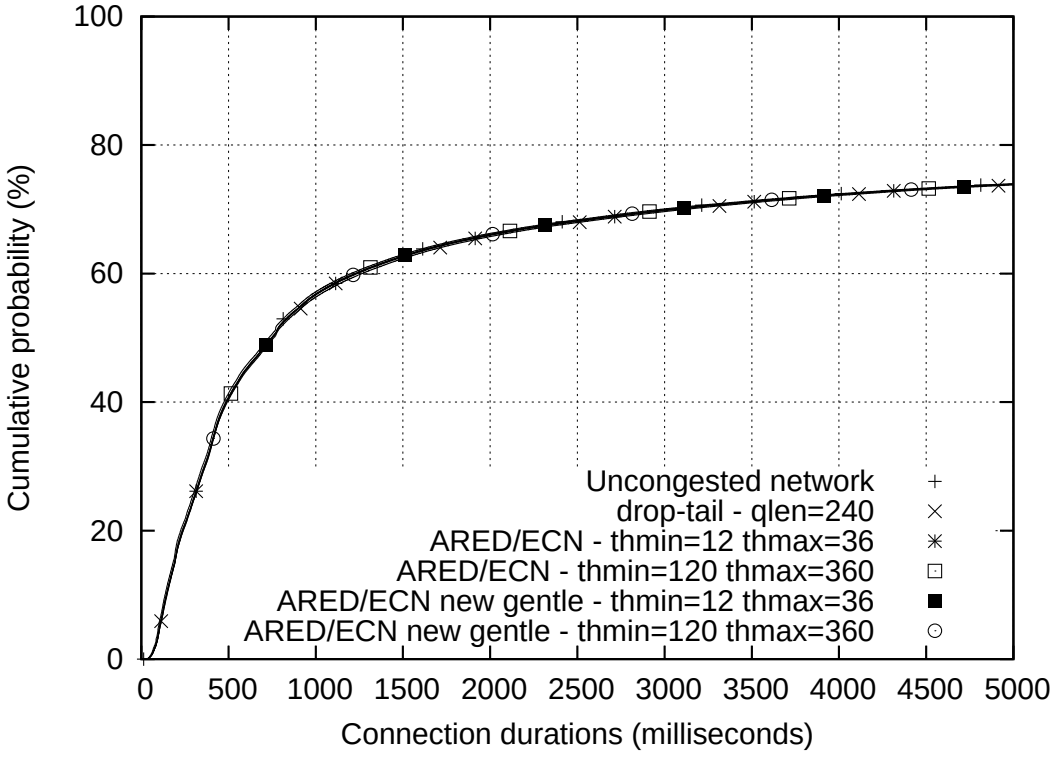

Figure 6.63: ARED/ECN new gentle performance at 80% load

packets, $th_{max} = 360$ packets). However, the performance for ARED/ECN and ARED/ECN "new gentle" with the parameter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets) degraded considerably at this load. When used with the parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets), ARED/ECN "new gentle" gave considerable better performance than ARED/ECN. However, ARED/ECN "new gentle" with the parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets) slightly underperformed drop-tail.

6.3.4 Results for LQD/ECN

Figures 6.69, 6.70, and 6.71 show experimental results for LQD with general TCP traffic when LQD was used with and without ECN at 80%, 90%, and 95% loads. These results were obtained with a queue reference of 24 and 240 packets.

At 80% offered load, LQD did not benefit from the addition of the ECN marking protocol. This is because LQD with both queue references of 24 and 240 packets already obtained identical performance as the uncongested network when LQD was used with packet drops.

At 90% offered load, LQD once again did not obtain any performance improvement from ECN. With or without ECN, LQD with both queue references delivered performance that was slightly better than that of drop-tail and almost indistinguishable from the performance of the uncongested network.

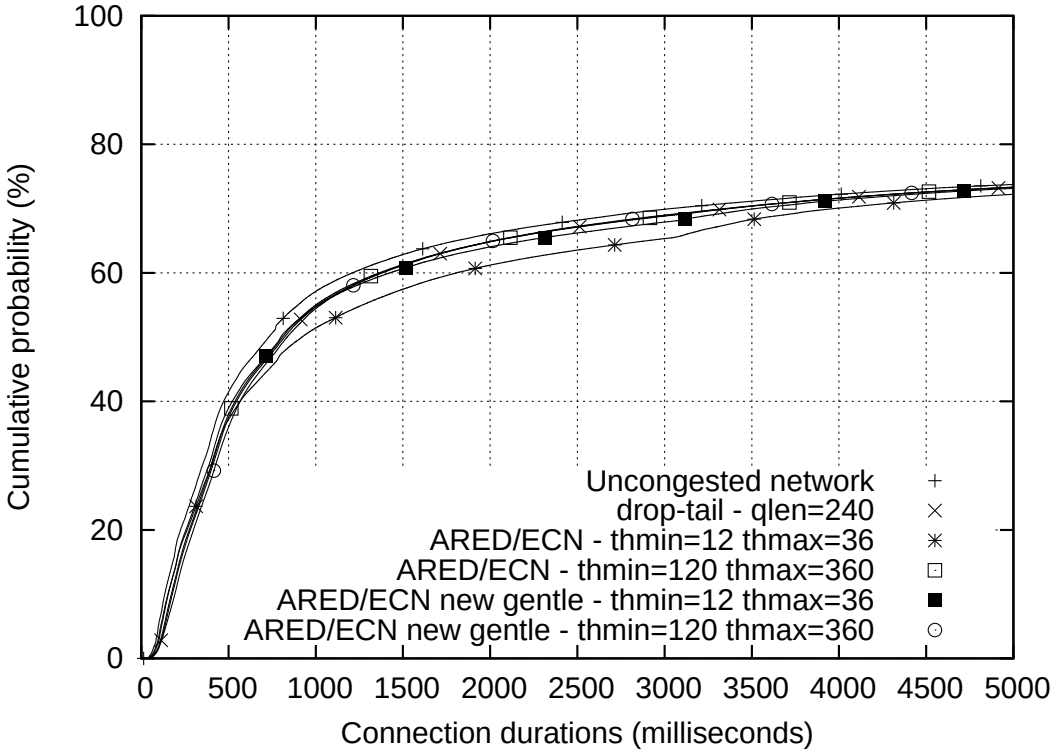

Figure 6.64: ARED/ECN new gentle performance at 90% load

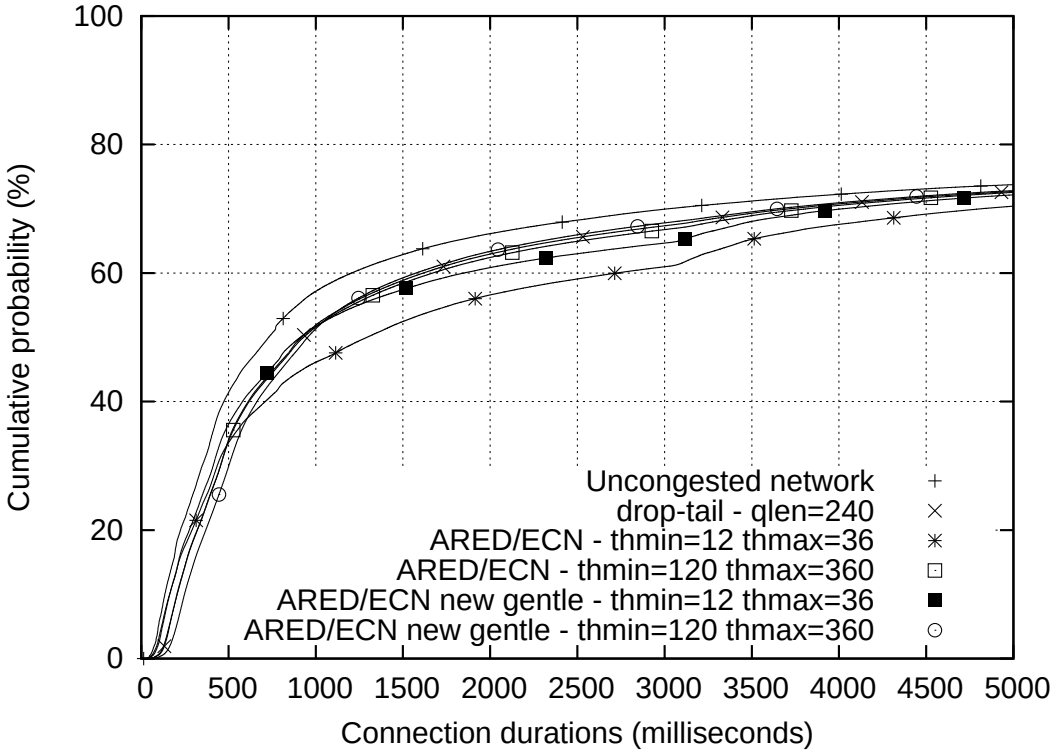

Figure 6.65: ARED/ECN new gentle performance at 95% load

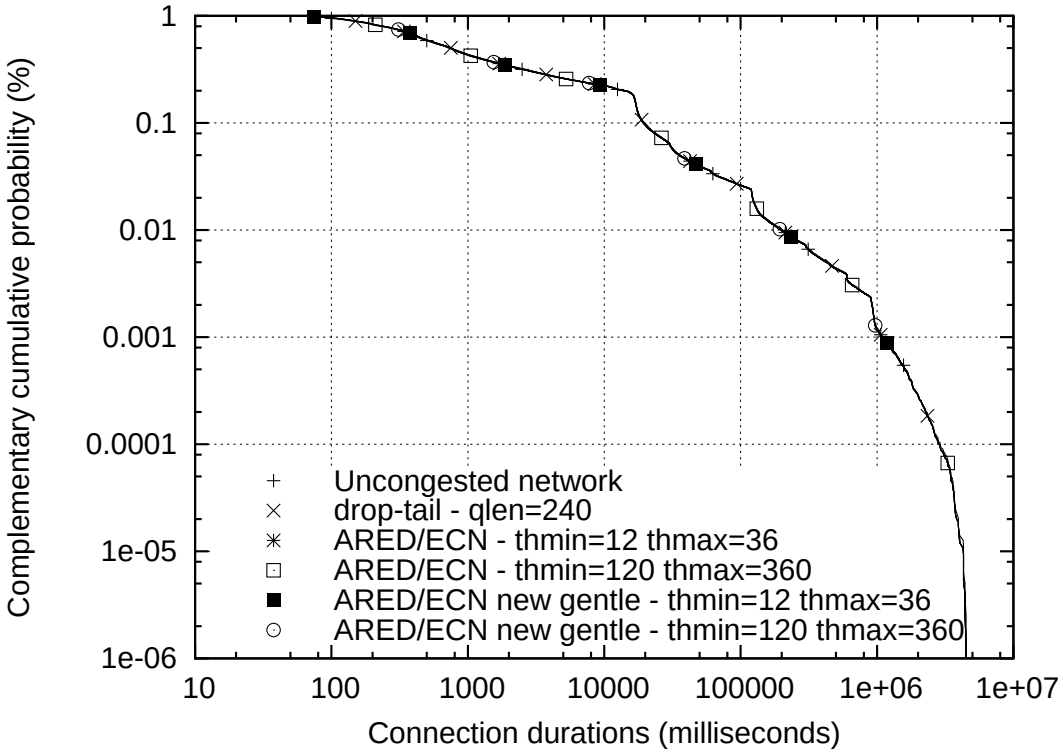

Figure 6.66: ARED/ECN new gentle performance at 80% load (CCDF)

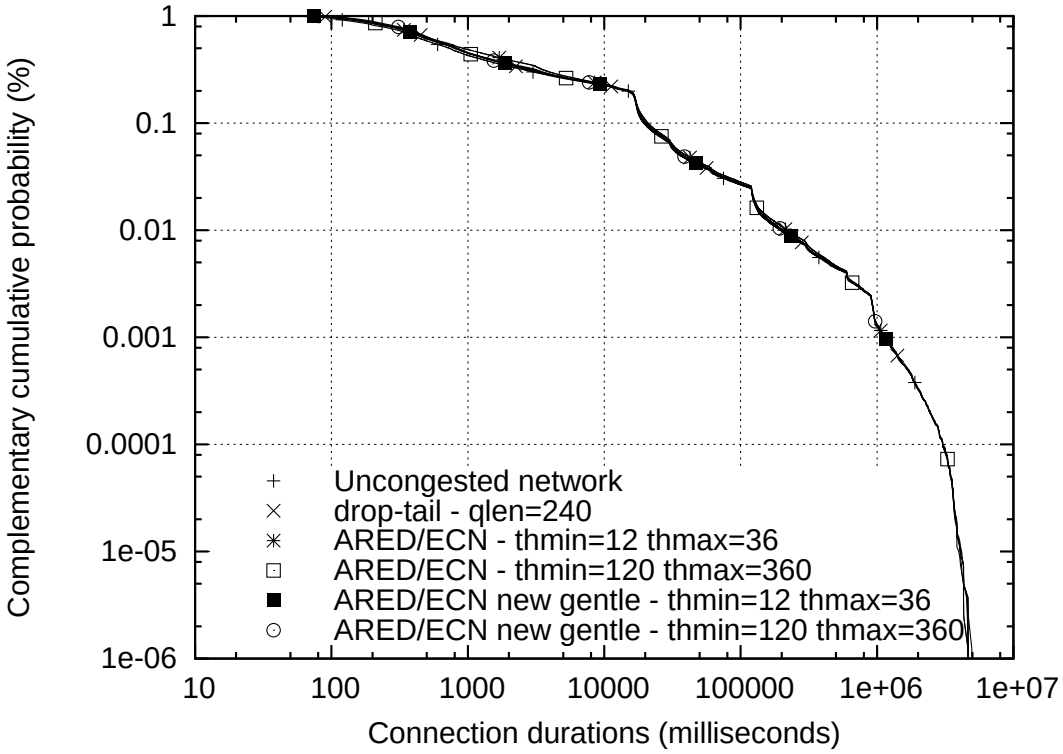

Figure 6.67: ARED/ECN new gentle performance at 90% load (CCDF)

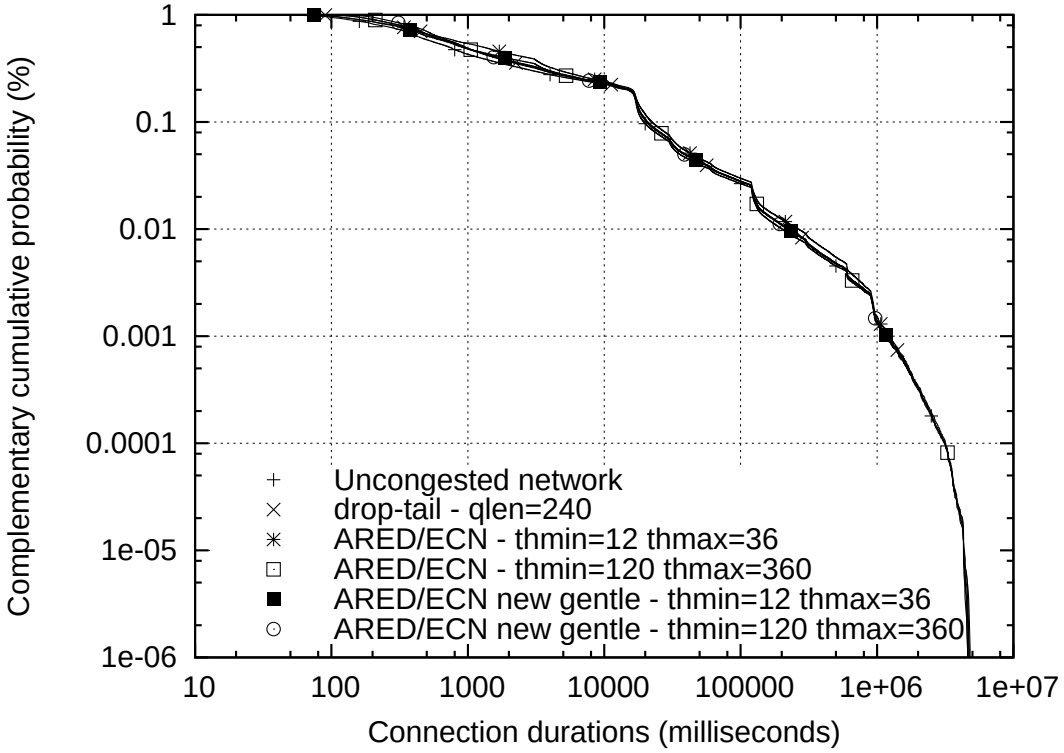

Figure 6.68: ARED/ECN new gentle performance at 95% load (CCDF)

At 95% offered load, ECN also did not improve the performance for LQD with both queue references of 24 and 240 packets. When used with either queue reference, the performance for LQD with or without ECN was slightly better than that of drop-tail and came very close to the performance of the uncongested network.

6.3.5 Results for DCN/ECN

Figures 6.75, 6.76, and 6.77 show experimental results for DCN with general TCP applications when DCN operated with and without ECN at 80%, 90%, and 95% loads. These results were achieved with a queue reference of 24 and 240 packets for DCN.

At 80% offered load, DCN obtained the same performance for both queue references of 24 and 240 packets when it was used with and without ECN. The performance for DCN was identical to drop-tail and the uncongested network at this load.

At 90% offered load, DCN also delivered the same performance for both queue references with and without ECN. The performance for DCN was slightly better than drop-tail and indistinguishable from that of the uncongested network.

As the offered load increased to 95%, DCN still did not suffer any performance degradation. With or without ECN, DCN with both queue references outperformed drop-tail and gave essentially the same performance as the uncongested network. This result once again

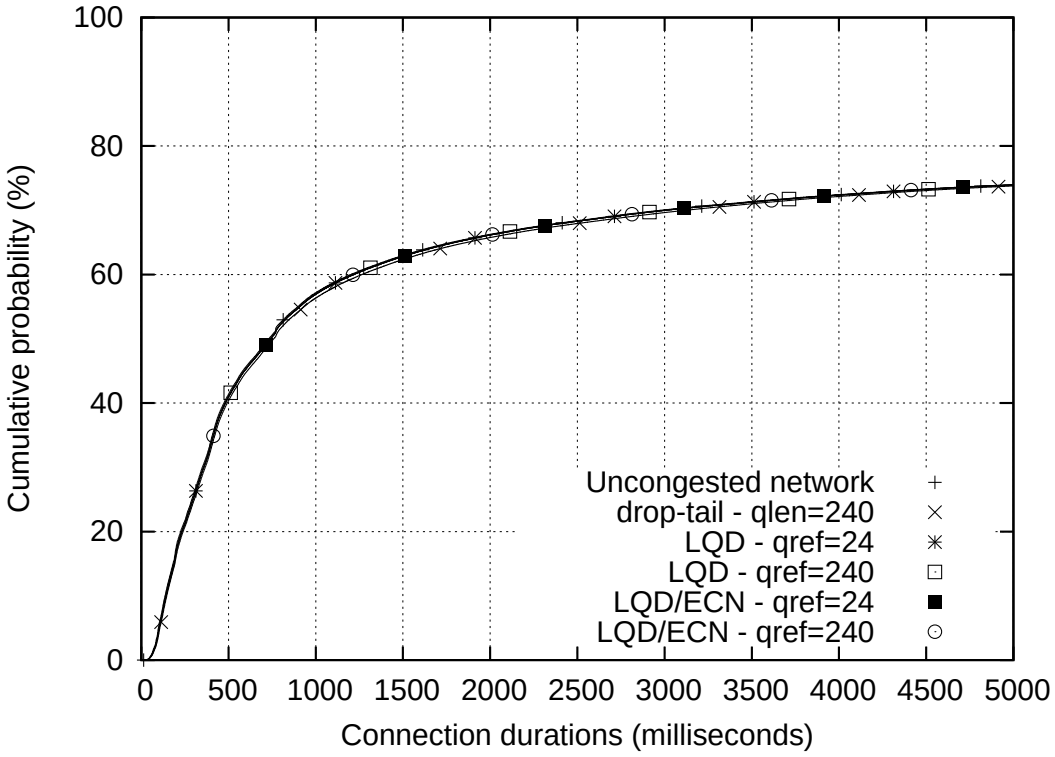

Figure 6.69: LQD/ECN performance at 80% load

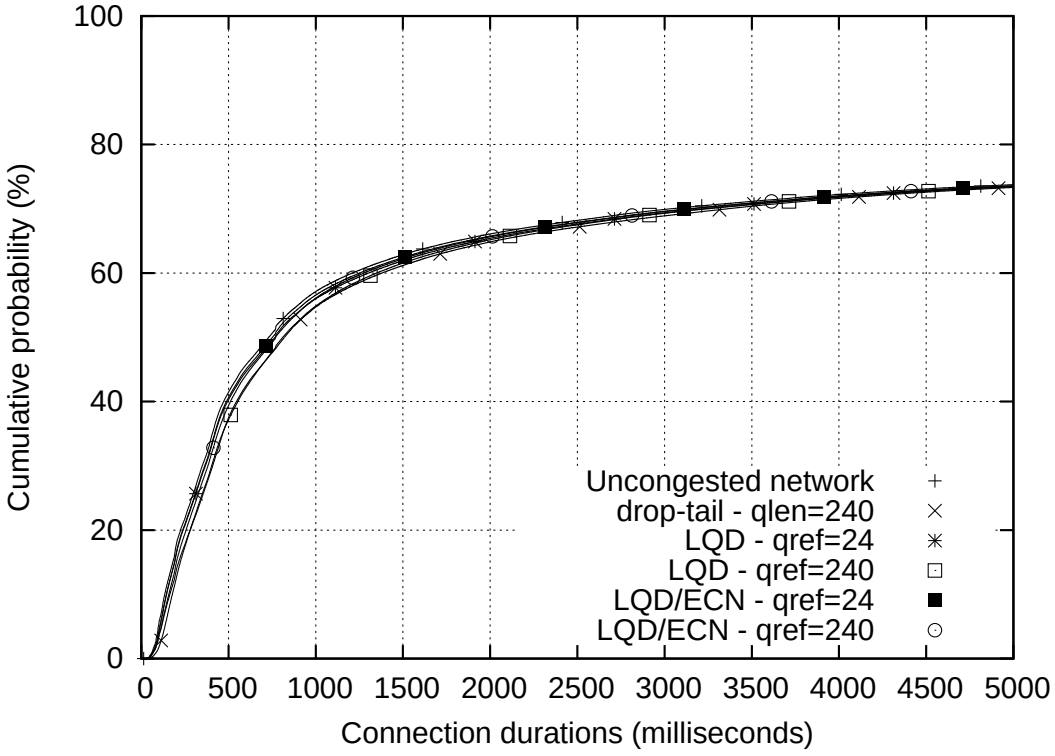

Figure 6.70: LQD/ECN performance at 90% load

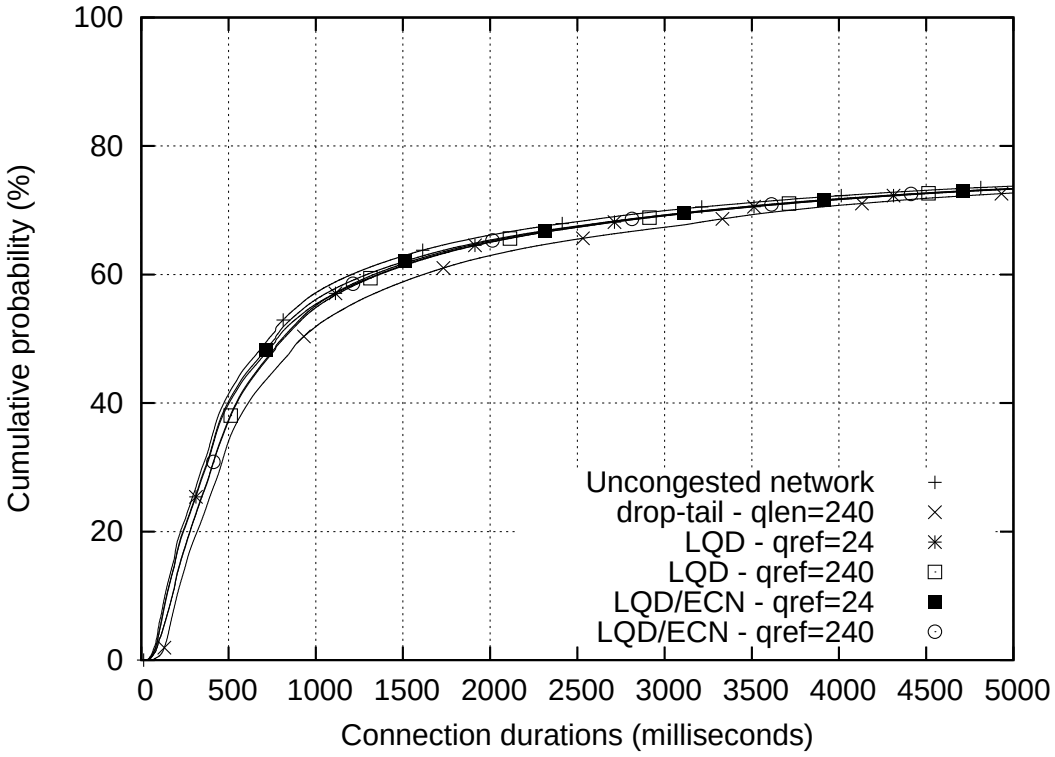

Figure 6.71: LQD/ECN performance at 95% load

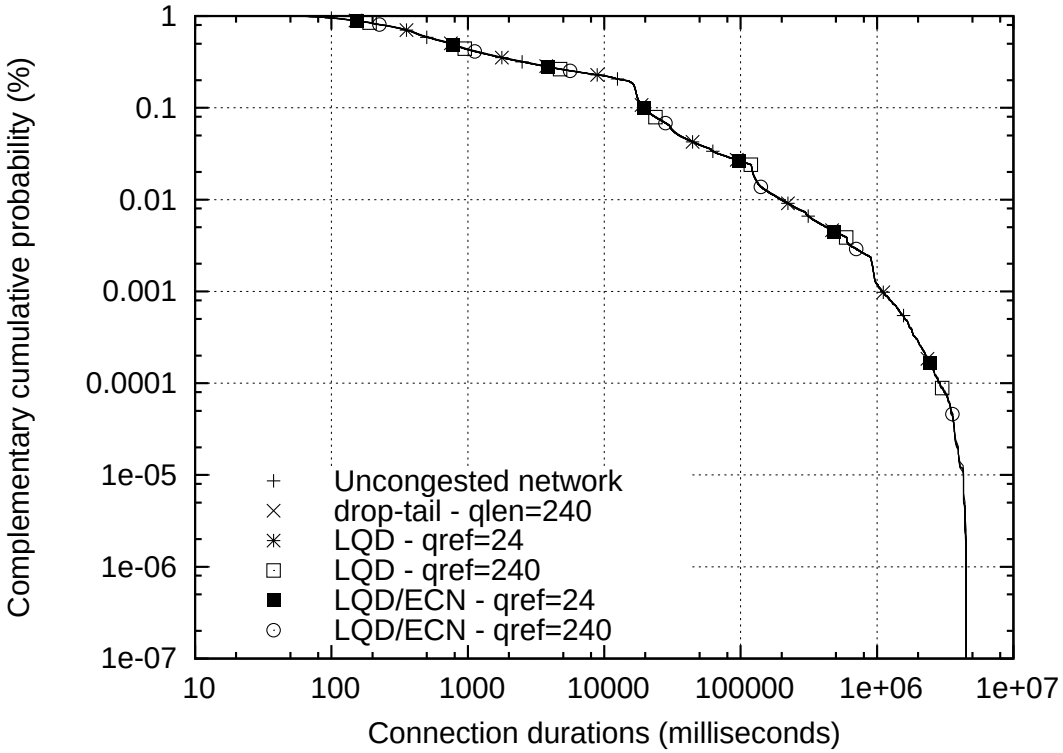

Figure 6.72: LQD/ECN performance at 80% load (CCDF)

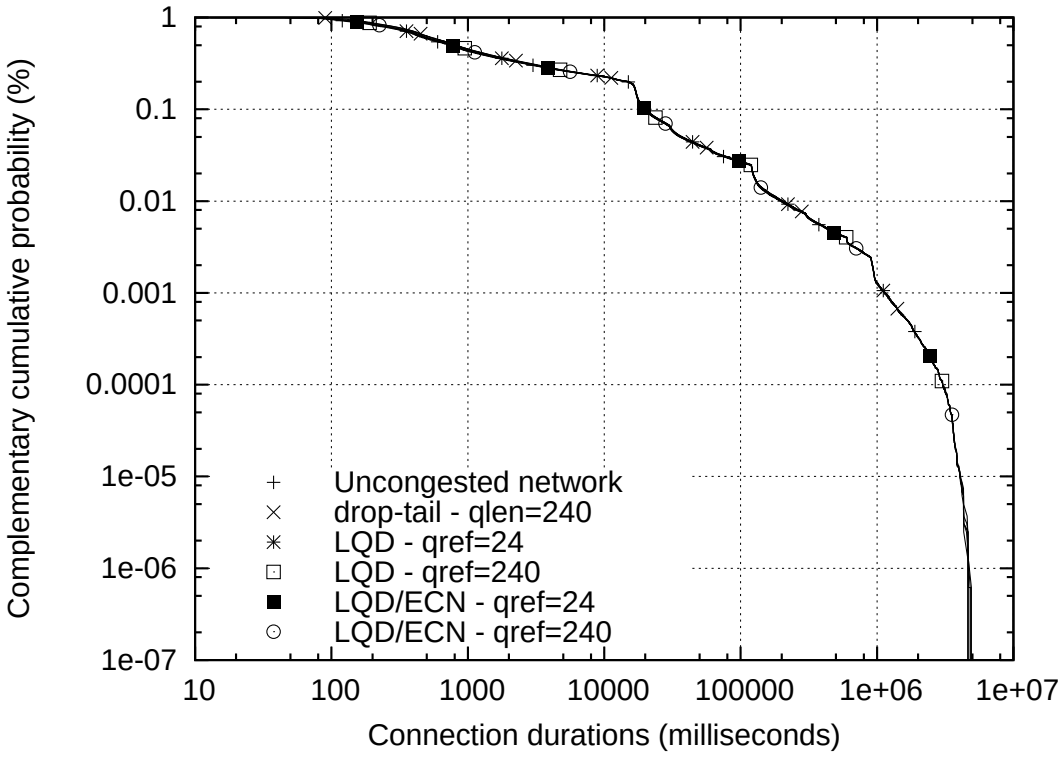

Figure 6.73: LQD/ECN performance at 90% load (CCDF)

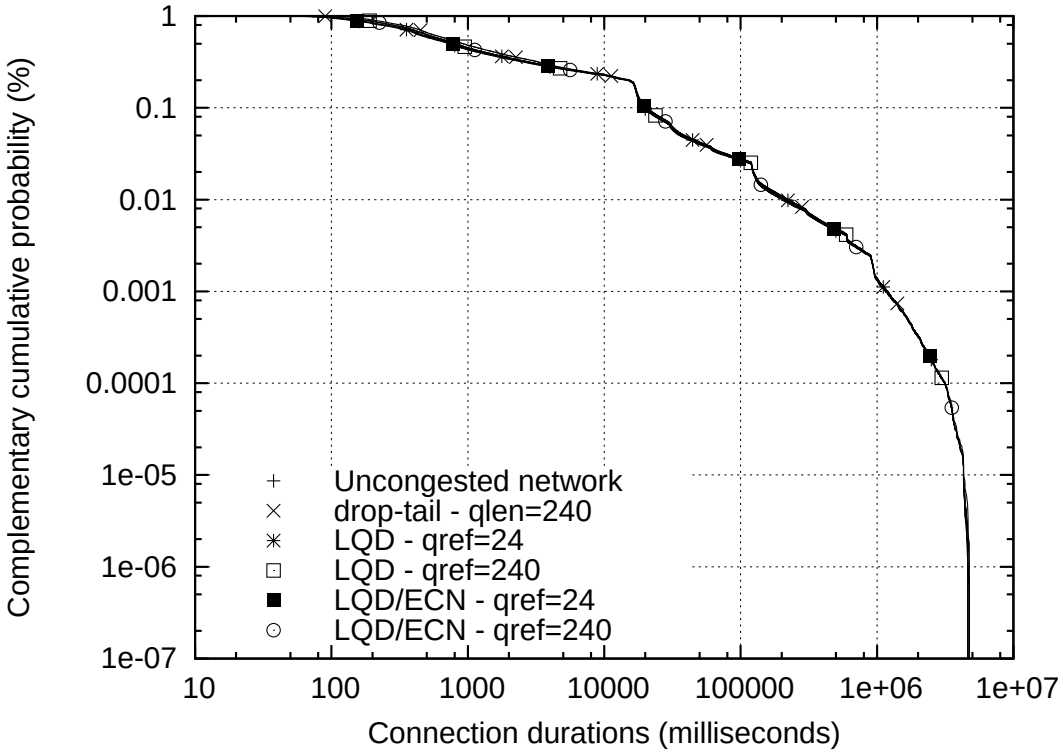

Figure 6.74: LQD/ECN performance at 95% load (CCDF)

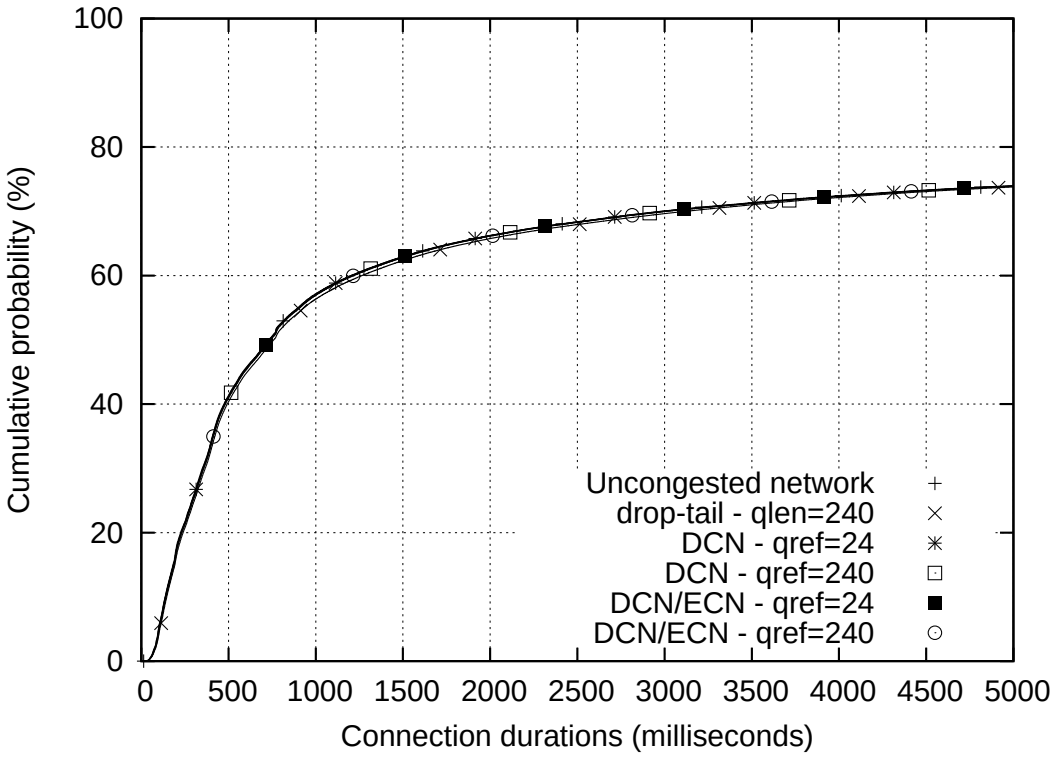

Figure 6.75: DCN/ECN performance at 80% load

demonstrates the benefits of differential treatment of flows in AQM algorithms.

6.4 Comparison of All Results

Figures 6.81, 6.82, and 6.83 show a comparison of results for all AQM algorithms with general TCP applications. These results were obtained with the best parameter settings for each of the AQM algorithms. Experimental results for DCN were shown when DCN operated without ECN to demonstrate the benefits of differential treatment of flows (recall from section 6.3.5 that DCN obtained the same performance with and without ECN).

At 80% offered load, drop-tail with a queue length of 240 packets achieved performance that was comparable with that of all AQM algorithms and with the performance of the uncongested network. Thus, it appears that Internet Service Providers can operate their networks for general TCP traffic at 80% without risking performance degradation for their customers' applications. Further, AQM seems to have no advantage over drop-tail for general TCP applications at 80% offered load or lower.

At 90% load, DCN delivered the best performance among all AQM algorithms even when DCN was used without ECN. The performance for DCN was equal to that of the uncongested network at this load. When used in combination with ECN, PI and REM

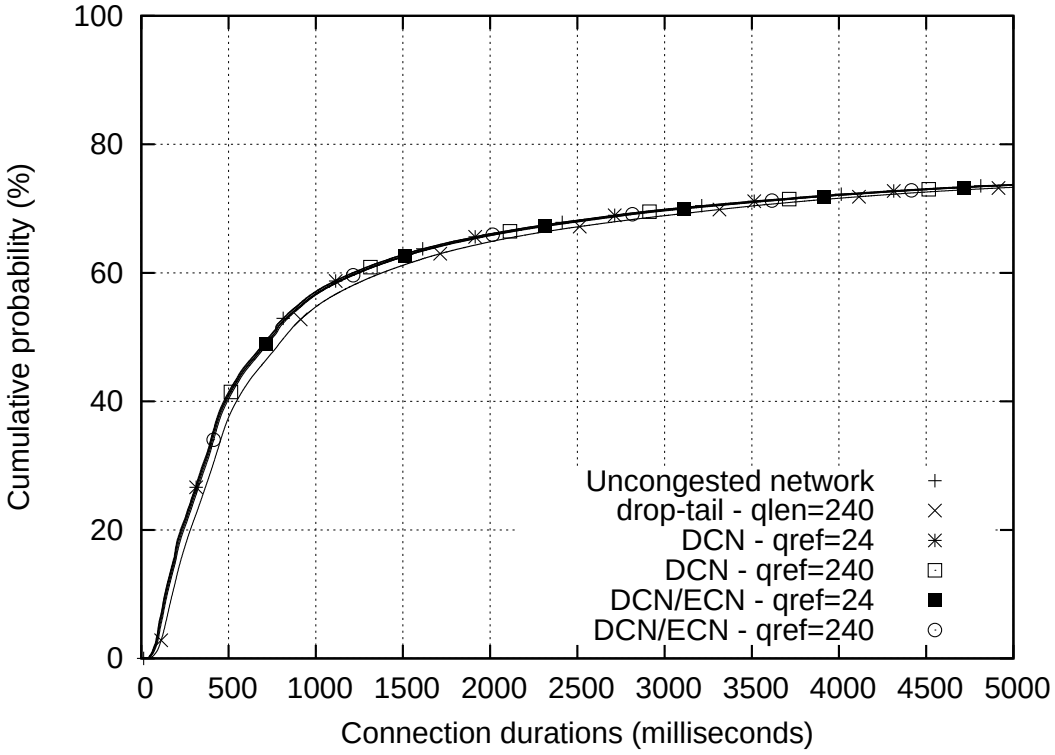

Figure 6.76: DCN/ECN performance at 90% load

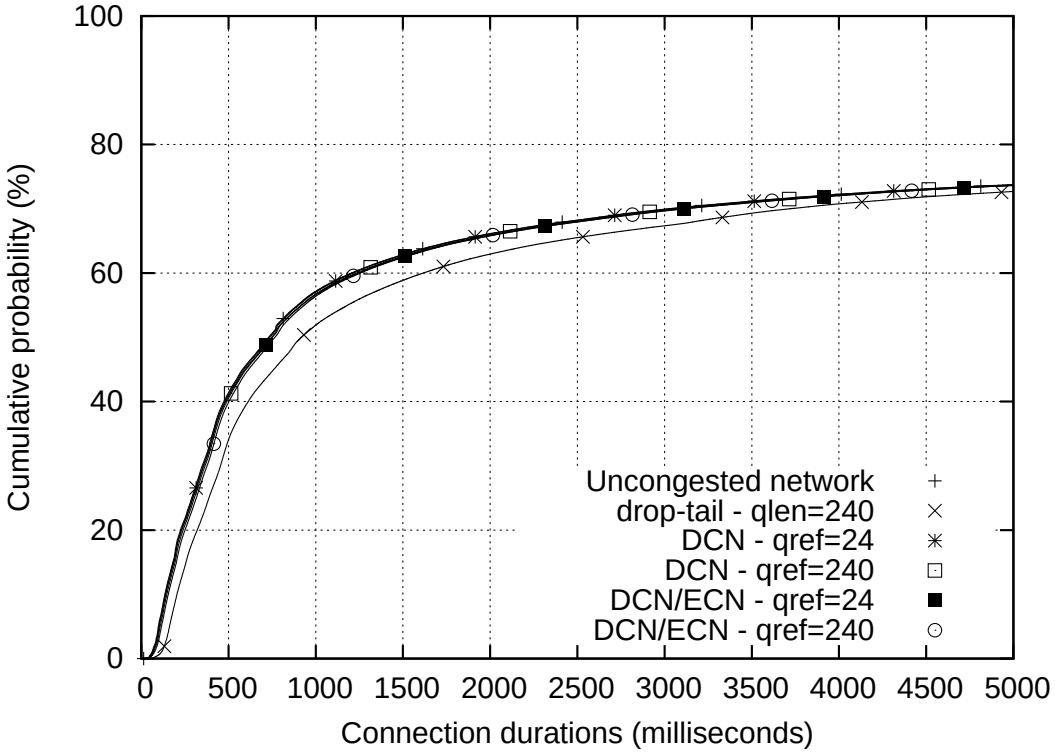

Figure 6.77: DCN/ECN performance at 95% load

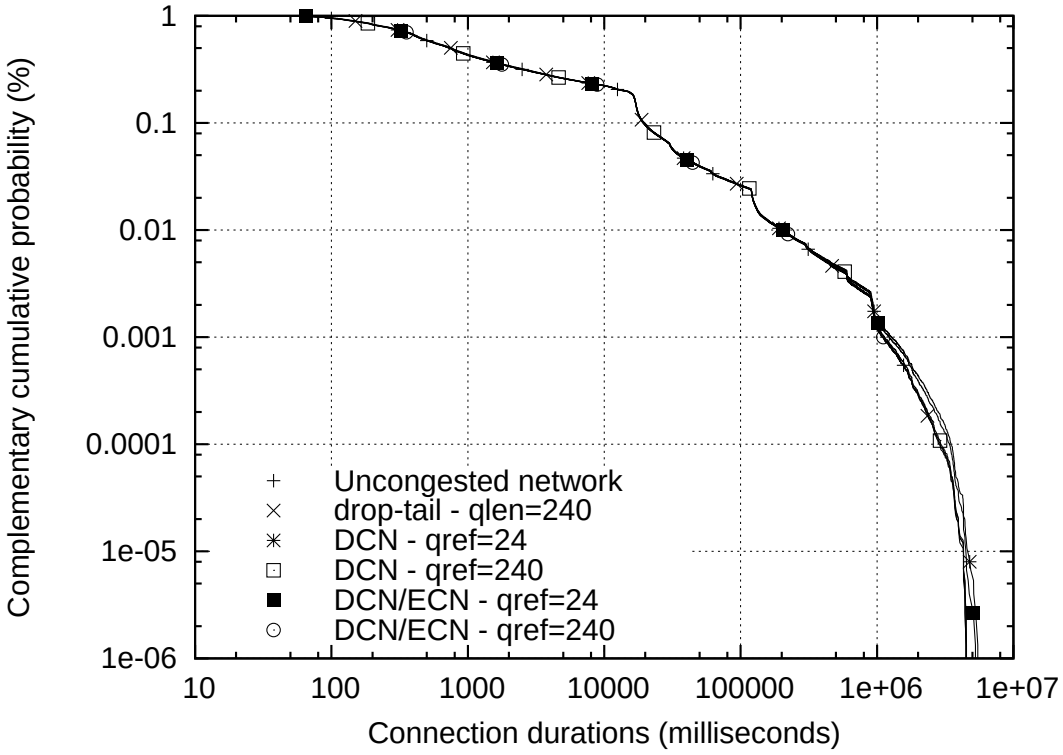

Figure 6.78: DCN/ECN performance at 80% load (CCDF)

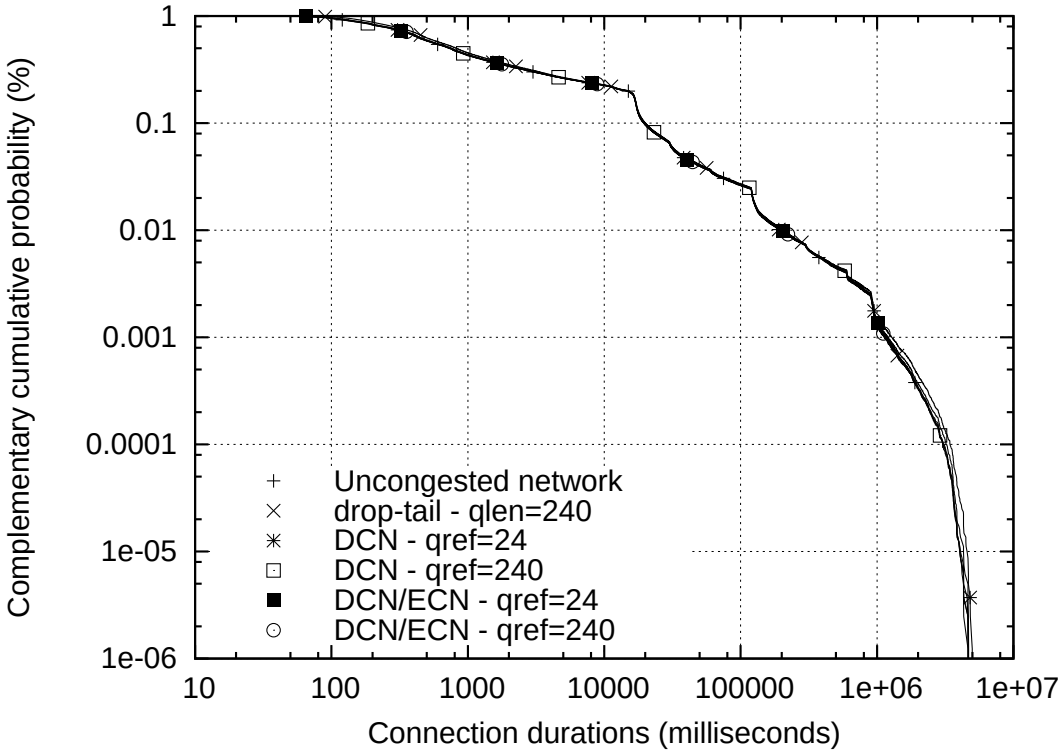

Figure 6.79: DCN/ECN performance at 90% load (CCDF)

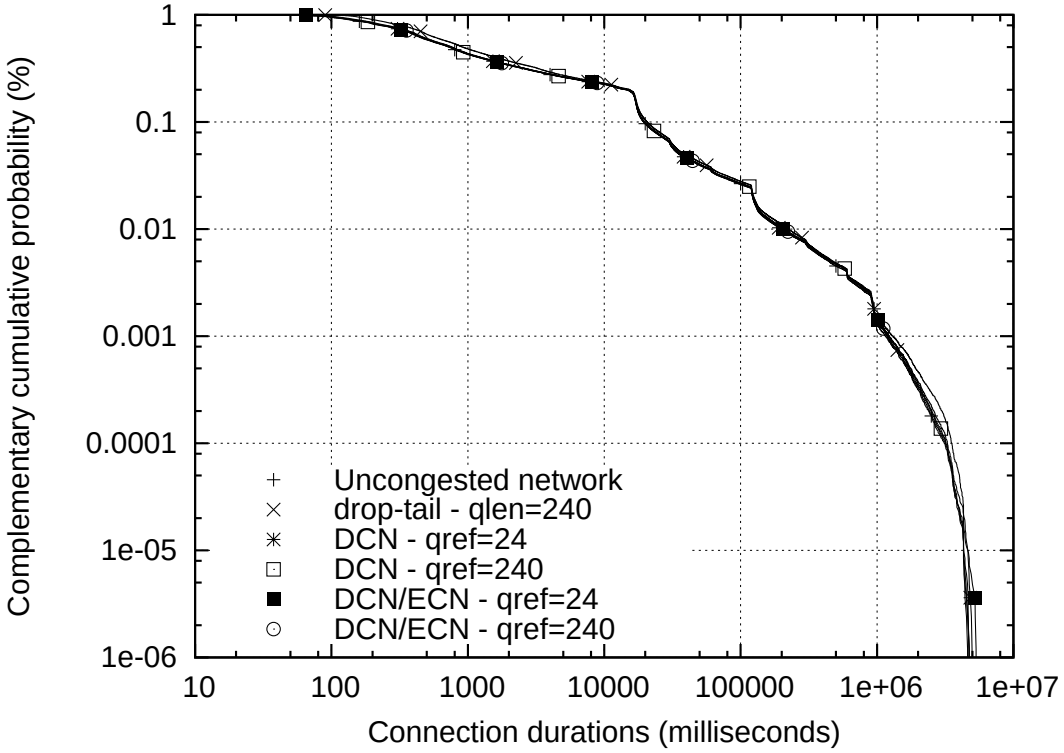

Figure 6.80: DCN/ECN performance at 95% load (CCDF)

obtained similar performance and slightly underperformed DCN. ARED/ECN "new gentle" gave about the same performance as drop-tail and slightly worse than PI and REM with ECN.

As the offered load increased to 95%, DCN outperformed all AQM algorithms even when it was used with packet drops. The performance for DCN was identical to that of the uncongested network. PI and REM with ECN gave about the same performance and slightly worse than drop-tail and DCN. ARED/ECN "new gentle" obtained slightly worse performance than drop-tail which in turn underperformed PI and REM with ECN.

6.5 Summary

This Chapter presents experimental results for PI, REM, ARED, LQD, and DCN with general TCP traffic. The traffic was synthetically generated in the laboratory network based on characteristics that were derived from packet traces of general TCP applications. Assuming that connection durations are the most important performance metric (and other performance metrics such as loss rate and link utilization are secondary) in evaluating AQM algorithms, following conclusions can be drawn from the results presented in this Chapter.

- At 90% offered load or lower, the performance for drop-tail with a queue length of

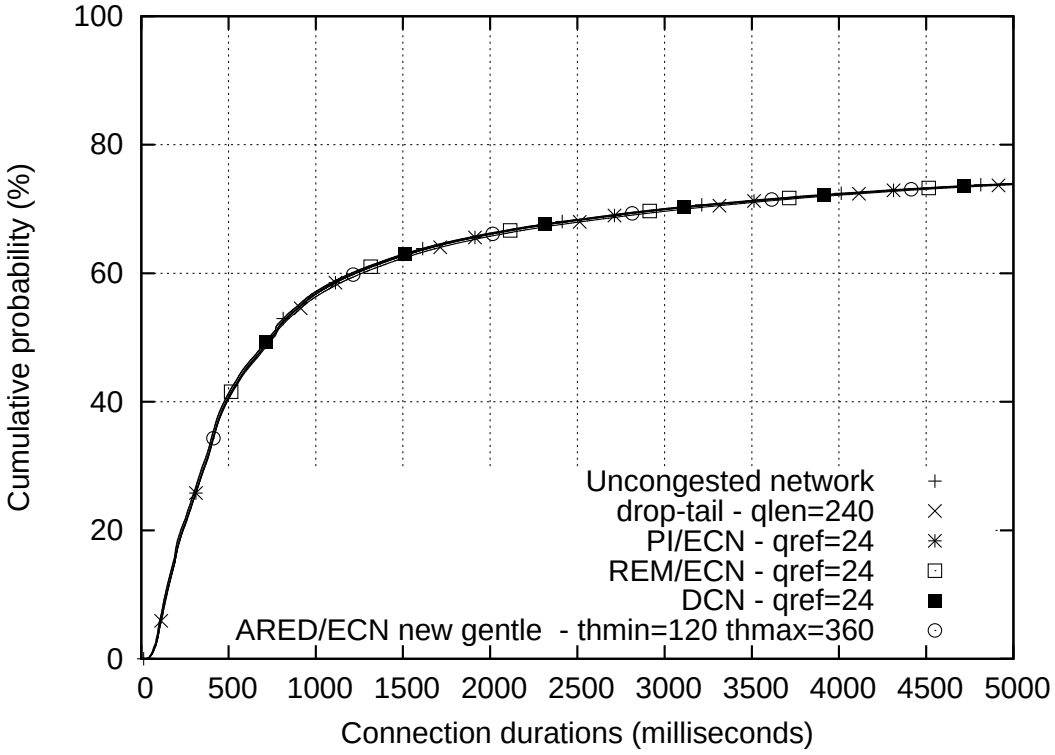

Figure 6.81: Comparison of all AQM algorithms at 80% load

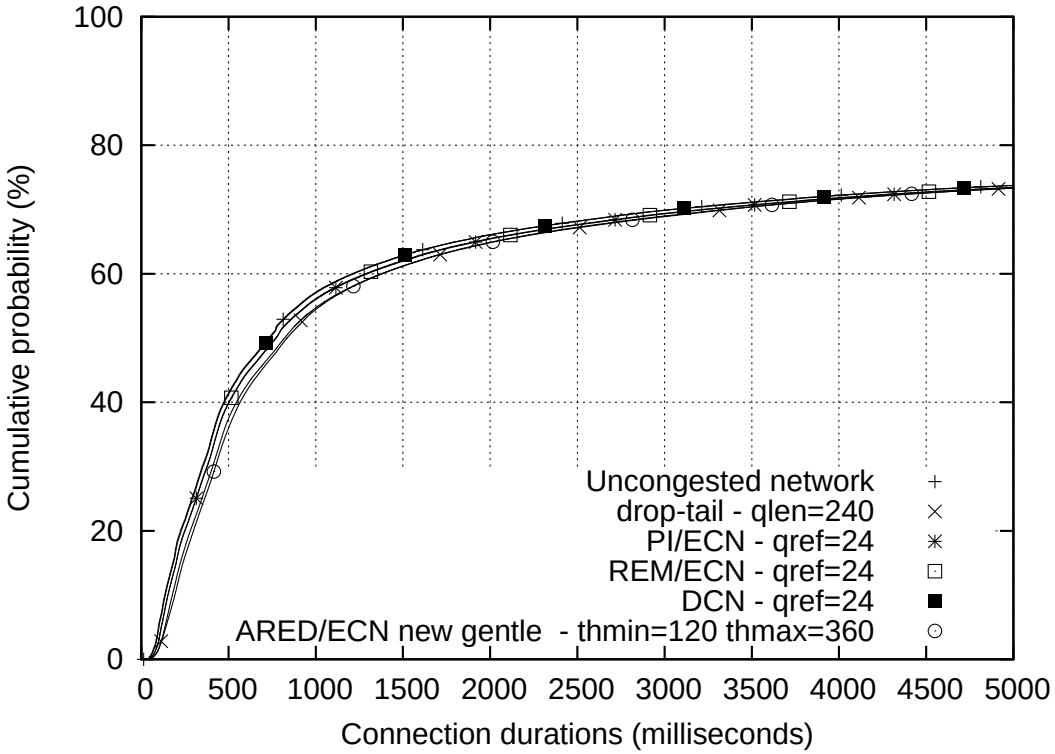

Figure 6.82: Comparison of all AQM algorithms at 90% load

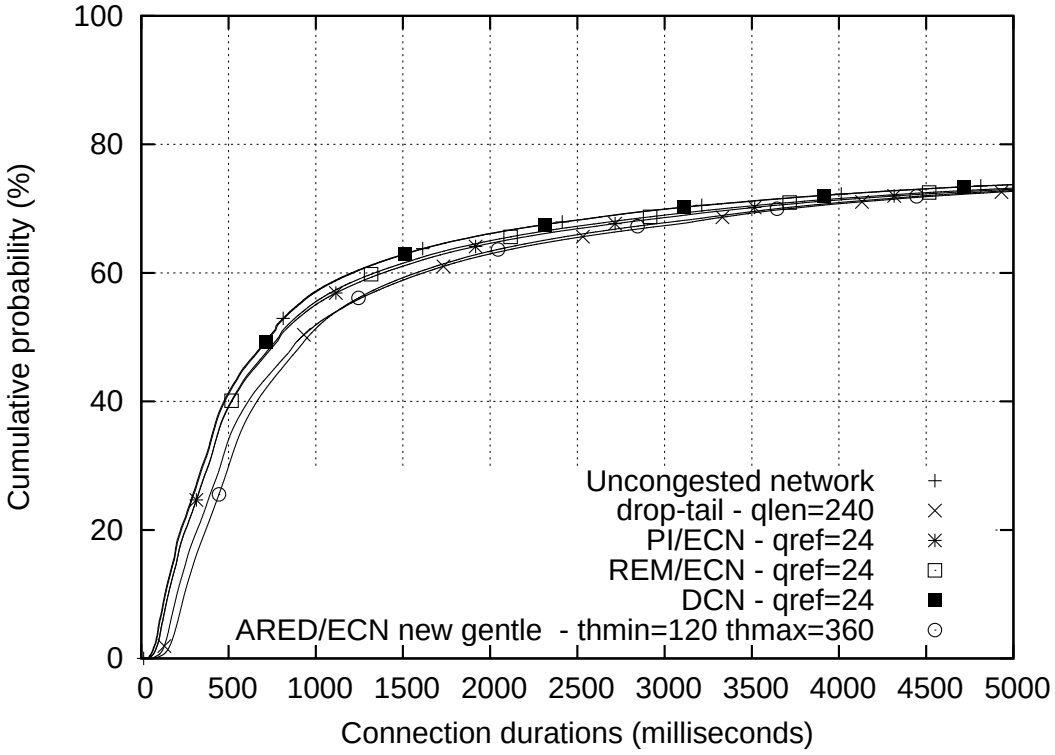

Figure 6.83: Comparison of all AQM algorithms at 95% load

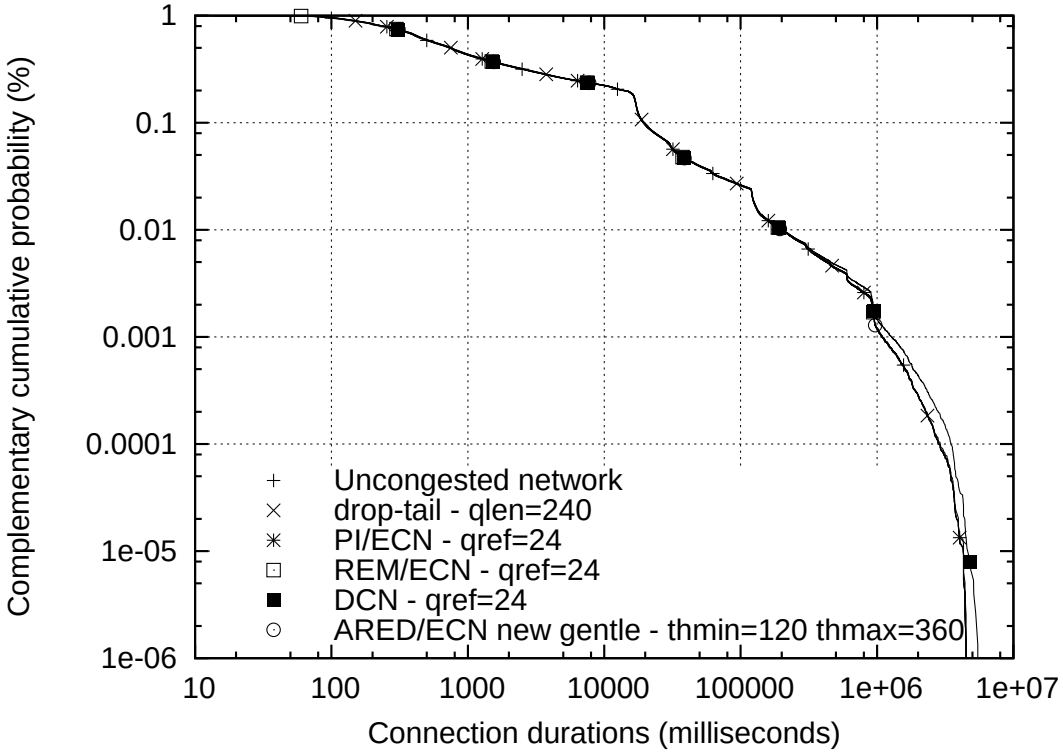

Figure 6.84: Comparison of all AQM algorithms at 80% load (CCDF)

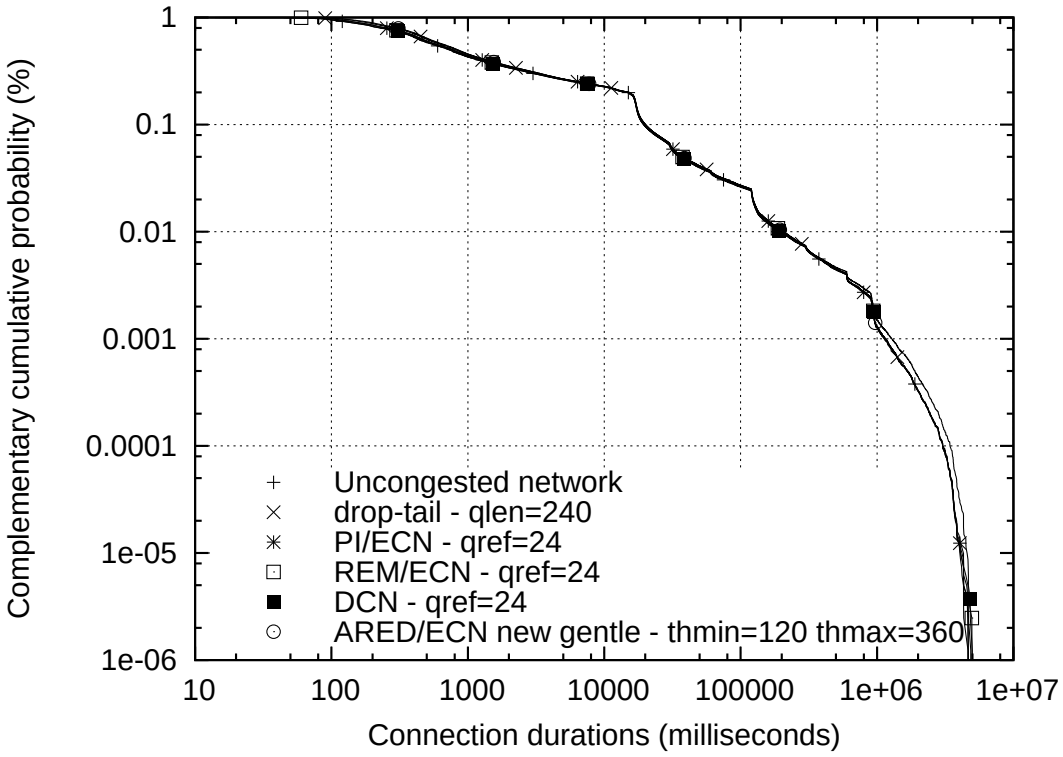

Figure 6.85: Comparison of all AQM algorithms at 90% load (CCDF)

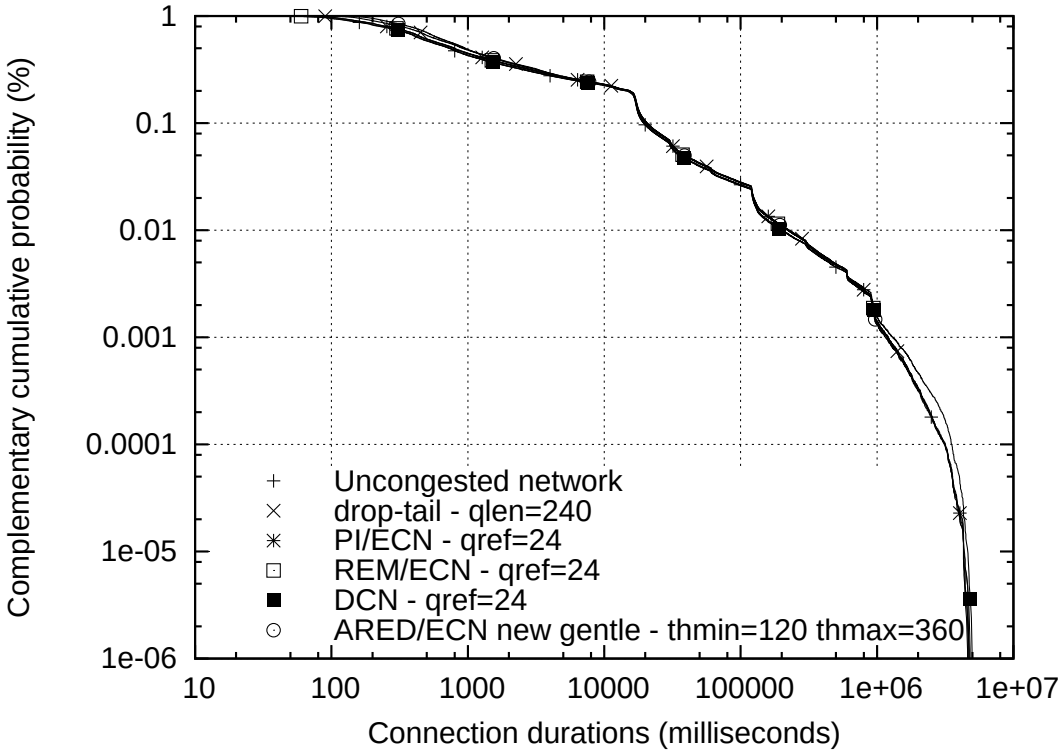

Figure 6.86: Comparison of all AQM algorithms at 95% load (CCDF)

240 packets was very close to that of all AQM algorithms and the performance of the uncongested network. Thus, it appears that Internet Service Providers could operate their network near saturation without causing noticeable performance degradation for their customers' applications. Further, AQM seems to have no advantage over drop-tail for general TCP applications at 90% offered load or below.

- In contrast to results with Web traffic presented in Chapters 4 and 5, ECN only improved the performance for AQM algorithms at 95% loads. Further, the performance improvement of ECN for AQM algorithms was significantly less impressive with general TCP applications than with Web applications.

- ARED was once again the worst performing AQM algorithm but the performance for ARED was improved considerably with the two modifications ARED "byte mode" and "new gentle" with ECN.

- The DCN algorithm demonstrated the power of differential treatment of flows. Even without the ECN marking protocol, DCN obtained performance that was indistinguishable from the performance of the uncongested network at 95% offered load.

Table 6.1: Loss rate, completed requests, and link utilization

	Offered load	Forward path loss rate (%)	Reverse path loss rate (%)	Completed connections (millions)	Forward path throughput (Mbps)	Reverse path throughput (Mbps)
Uncongested 1 Gbps network	80%	0.0	0.0	0.75	80.8	76.9
	90%	0.0	0.0	0.81	91.4	85.5
	95%	0.0	0.0	0.83	98.7	90.7
Drop-tail queue size = 24	80%	0.1	0.0	0.75	80.0	76.2
	90%	0.7	0.3	0.81	87.5	82.7
	95%	1.7	0.9	0.83	88.4	87.0
Drop-tail queue size = 240	80%	0.0	0.0	0.75	80.0	76.3
	90%	0.4	0.1	0.81	87.5	82.9
	95%	0.9	0.4	0.83	89.4	87.3
PI $q_{ref} = 24$	80%	0.1	0.0	0.75	75.7	72.8
	90%	0.7	0.1	0.81	85.3	82.6
	95%	1.3	0.7	0.83	86.1	84.9
PI/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.0	0.0	0.81	86.1	82.8
	95%	0.1	0.0	0.83	86.7	86.0
PI $q_{ref} = 240$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.3	0.0	0.81	87.5	80.7
	95%	0.7	0.2	0.83	87.2	87.0
PI/ECN $q_{ref} = 240$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.0	0.0	0.81	86.9	83.1
	95%	0.0	0.0	0.83	87.7	87.0
REM $q_{ref} = 24$	80%	0.1	0.0	0.75	75.7	72.8
	90%	0.6	0.2	0.81	85.6	82.2
	95%	1.2	0.6	0.83	85.5	84.8
REM/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.0	0.0	0.81	86.8	82.7
	95%	0.1	0.0	0.83	86.9	86.9

Continued on next page

	Offered load	Forward path loss rate (%)	Reverse path loss rate (%)	Completed connections (millions)	Forward path throughput (Mbps)	Reverse path throughput (Mbps)
REM	80%	0.1	0.0	0.75	75.6	71.0
$q_{ref} = 240$	90%	0.2	0.1	0.81	86.7	82.5
	95%	0.7	0.4	0.83	86.1	84.6
REM/ECN	80%	0.0	0.0	0.75	75.7	72.8
$q_{ref} = 240$	90%	0.0	0.0	0.81	86.8	82.7
	95%	0.1	0.0	0.83	86.9	86.9
ARED	80%	0.2	0.1	0.75	75.5	72.8
$th_{min} = 12$	90%	2.3	0.8	0.81	80.7	80.1
$th_{min} = 36$	95%	3.6	2.5	0.83	81.1	79.5
ARED/ECN	80%	0.1	0.0	0.75	75.6	72.7
$th_{min} = 12$	90%	2.2	0.7	0.81	80.5	80.2
$th_{min} = 36$	95%	3.3	2.2	0.83	81.3	79.5
ARED	80%	0.1	0.0	0.75	75.7	72.8
$th_{min} = 120$	90%	0.3	0.1	0.81	86.3	82.6
$th_{min} = 360$	95%	1.1	0.4	0.83	86.8	86.1
ARED/ECN	80%	0.0	0.0	0.75	75.7	72.8
$th_{min} = 120$	90%	0.3	0.0	0.81	86.6	82.8
$th_{min} = 360$	95%	1.1	0.4	0.83	86.4	84.8
ARED "byte	80%	0.2	0.0	0.75	75.5	72.4
mode" $th_{min} = 12$	90%	1.6	0.8	0.81	80.8	79.2
$th_{max} = 36$	95%	2.2	1.6	0.83	81.5	80.5
ARED "byte	80%	0.0	0.0	0.75	75.7	72.8
mode" $th_{min} = 120$	90%	0.4	0.1	0.81	86.1	82.1
$th_{max} = 360$	95%	1.4	0.9	0.83	86.8	82.4
ARED/ECN "new	80%	0.0	0.0	0.75	75.7	72.8
gentle" $th_{min} = 12$	90%	0.7	0.2	0.81	87.0	82.6
$th_{max} = 36$	95%	1.8	0.7	0.83	87.2	86.2
ARED/ECN "new	80%	0.0	0.0	0.75	75.7	72.8
gentle" $th_{min} = 120$	90%	0.1	0.0	0.81	87.5	82.5
$th_{max} = 360$	95%	0.3	0.1	0.83	89.8	86.6

Continued on next page

	Offered load	Forward path loss rate (%)	Reverse path loss rate (%)	Completed connections (millions)	Forward path throughput (Mbps)	Reverse path throughput (Mbps)
LQD $q_{ref} = 24$	80%	0.0	0.0	0.75	75.8	72.8
	90%	0.0	0.0	0.81	87.6	80.7
	95%	0.0	0.0	0.83	89.9	83.4
LQD/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	75.9	72.9
	90%	0.0	0.0	0.81	87.9	83.5
	95%	0.0	0.0	0.83	90.1	86.7
LQD $q_{ref} = 240$	80%	0.0	0.0	0.75	75.9	73.1
	90%	0.0	0.0	0.81	87.6	81.3
	95%	0.0	0.0	0.83	89.9	83.5
LQD/ECN $q_{ref} = 240$	80%	0.0	0.0	0.75	76.1	73.1
	90%	0.0	0.0	0.81	87.9	81.4
	95%	0.0	0.0	0.83	90.3	83.6
DCN $q_{ref} = 24$	80%	0.1	0.0	0.75	75.7	72.8
	90%	0.2	0.0	0.81	87.1	83.0
	95%	0.5	0.1	0.83	87.3	87.2
DCN/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.0	0.0	0.81	87.3	83.0
	95%	0.0	0.0	0.83	87.9	87.2
DCN $q_{ref} = 240$	80%	0.1	0.0	0.75	76.4	73.1
	90%	0.2	0.0	0.81	87.6	83.2
	95%	0.3	0.0	0.83	88.2	88.0
DCN/ECN $q_{ref} = 240$	80%	0.0	0.0	0.75	76.6	73.3
	90%	0.0	0.0	0.81	87.9	83.3
	95%	0.0	0.0	0.83	88.4	88.1

Table 6.2: Percentiles of response times

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
Uncongested 1 Gbps network	80%	0.730	5.897	19.365
	90%	0.730	6.152	19.505
	95%	0.730	6.166	19.513
Drop-tail queue size = 24	80%	0.757	6.004	19.498
	90%	0.810	6.589	20.074
	95%	0.913	7.084	20.649
Drop-tail queue size = 240	80%	0.759	6.083	19.563
	90%	0.820	6.525	19.988
	95%	0.916	7.016	20.584
PI $q_{ref} = 24$	80%	0.749	5.985	19.451
	90%	0.796	6.702	20.213
	95%	0.830	7.012	20.652
PI/ECN $q_{ref} = 24$	80%	0.749	5.978	19.448
	90%	0.773	6.473	19.941
	95%	0.794	6.823	20.365
PI $q_{ref} = 240$	80%	0.750	5.985	19.453
	90%	0.901	6.698	20.193
	95%	0.906	6.946	20.553
PI/ECN $q_{ref} = 240$	80%	0.750	5.984	19.453
	90%	0.813	6.479	19.943
	95%	0.888	6.828	20.399
REM $q_{ref} = 24$	80%	0.744	5.973	19.448
	90%	0.794	6.675	20.204
	95%	0.825	6.979	20.620
REM/ECN $q_{ref} = 24$	80%	0.744	5.968	19.437
	90%	0.769	6.444	19.900
	95%	0.788	6.688	20.206
REM $q_{ref} = 240$	80%	0.758	6.079	19.561
	90%	0.804	6.474	19.947
	95%	0.883	6.946	20.542
REM/ECN $q_{ref} = 240$	80%	0.748	5.980	19.445
	90%	0.799	6.435	19.879
	95%	0.859	6.747	20.264

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
ARED	80%	0.762	6.118	19.635
$th_{min} = 12$	90%	0.950	7.339	20.990
$th_{max} = 36$	95%	1.261	8.240	22.160
ARED/ECN	80%	0.747	5.992	19.476
$th_{min} = 12$	90%	0.921	7.282	20.881
$th_{max} = 36$	95%	1.293	8.501	22.372
ARED	80%	0.747	5.979	19.444
$th_{min} = 120$	90%	0.817	6.629	20.084
$th_{max} = 360$	95%	0.920	7.039	20.627
ARED/ECN	80%	0.748	5.979	19.445
$th_{min} = 120$	90%	0.809	6.502	19.977
$th_{max} = 360$	95%	0.930	7.114	20.719
ARED "byte	80%	0.756	6.085	19.602
mode" $th_{min} = 12$	90%	0.807	6.827	20.359
$th_{max} = 36$	95%	0.879	7.267	20.964
ARED "byte	80%	0.750	5.982	19.449
mode" $th_{min} = 120$	90%	0.818	6.499	19.976
$th_{max} = 360$	95%	0.910	6.934	20.533
ARED "new	80%	0.745	5.972	19.442
gentle" $th_{min} = 12$	90%	0.799	6.611	20.081
$th_{max} = 36$	95%	0.911	7.263	20.723
ARED "new	80%	0.750	5.985	19.450
gentle" $th_{min} = 120$	90%	0.835	6.531	19.997
$th_{max} = 360$	95%	0.953	6.936	20.511
LQD	80%	0.739	5.939	19.408
$q_{ref} = 24$	90%	0.767	6.426	19.871
	95%	0.777	6.599	20.095
LQD/ECN	80%	0.739	5.938	19.408
$q_{ref} = 24$	90%	0.752	6.313	19.726
	95%	0.765	6.503	19.935
LQD	80%	0.740	5.939	19.408
$q_{ref} = 240$	90%	0.811	6.426	19.854
	95%	0.814	6.564	20.041

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
LQD/ECN $q_{ref} = 240$	80%	0.740	5.941	19.408
	90%	0.770	6.316	19.725
	95%	0.806	6.504	19.958
DCN $q_{ref} = 24$	80%	0.733	5.948	19.3815
	90%	0.735	6.174	19.509
	95%	0.736	6.183	19.514
DCN/ECN $q_{ref} = 24$	80%	0.734	5.940	19.396
	90%	0.745	6.255	19.619
	95%	0.747	6.266	19.629
DCN $q_{ref} = 240$	80%	0.739	5.967	19.435
	90%	0.746	6.233	19.609
	95%	0.750	6.253	19.622
DCN/ECN $q_{ref} = 240$	80%	0.741	5.960	19.436
	90%	0.756	6.250	19.642
	95%	0.765	6.292	19.682

Chapter 7

Investigating the Effects of Link-Level Buffering

Experimental results in Chapters 4, 5, and 6 showed the effects of AQM algorithms on the performance of Web and general TCP applications. These results were obtained by performing experiments in a laboratory network and having two PC routers running implementations of AQM algorithms. The AQM algorithms were implemented in FreeBSD kernel of the PC routers by using the ALTQ framework [Cho98]. This framework extends the structure of the IP output queue in the TCP/IP stack and allows a modular implementation of traffic management algorithms such as AQM or packet scheduling in the IP stack of a PC router.

Figure 7.1 depicts the principal architecture of ALTQ where packets leaving the IP output queue can be redirected to the implementation of different AQM algorithms. Packets that are forwarded, i.e., not dropped, by these AQM algorithms are passed to *if_start* (a macro in FreeBSD that is used to link to various software drivers for network adapters) and then to the software drivers of various network adapters (before ALTQ was introduced, packets flew directly from the IP output queue to *if_start*). Control software running in user-space specifies at run time which implementation of AQM algorithms should be active and is to handle packets leaving the IP output queue.

The ALTQ framework provides AQM implementors with an abstract environment and frees them from having to deal directly with idiosyncratic implementation details of network adapters. Because of this abstraction, however, AQM algorithms only control the IP output queue and are not aware of additional buffering that takes place within device drivers and hardware interfaces. This "link-level" buffer inside the device drivers can cause two potential problems. First, the "link-level" buffer increases end-to-end latency and can destroy one of the main purposes of AQM algorithms (which is to reduce queuing delay and improve the performance of interactive applications). Second, because AQM algorithms is not aware of the link-level buffer, this buffer can cause inaccuracy in the operations of AQM algorithms.

For example, an empty IP output queue is interpreted as an abatement (or absence) of congestion by most AQM algorithms. However, this interpretation is incorrect if packets are buffered in software drivers at the link layer.

In order to investigate the effects of the link-level buffering, implementations for PI, REM, ARED, LQD, and DCN were modified to control both the IP output queue and the link-level buffer. Experiments with general TCP traffic under the same conditions as in 6 were repeated with the modified implementations of AQM algorithms. The modification of the implementation for AQM algorithms partially circumvented the purpose of the ALTQ framework (which was to provide an abstract environment for implementation of traffic management algorithms). However, it was necessary to achieve accurate control of packet queue(s) for drop-tail and AQM algorithms. For example, experimental results presented in Chapters 4, 5, and 6 were obtained when drop-tail and AQM algorithms ran on top of another packet queue inside the software driver for 3Com network adapters but were unaware of this queue. Further, this packet queue can store a maximum of 254 packets, a considerable amount of packet buffering.

In the new implementations of ALTQ and AQM algorithms, the packet queue inside the software driver for 3Com network adapters was limited to a maximum of 4 packets. Further, this packet queue and the IP output queue were taken into account (and controlled) by drop-tail and AQM algorithms.

As in Chapters 5 and 6, experimental results were obtained for PI, REM, ARED, LQD, and DCN. For comparison purposes, experiments were also performed for drop-tail and the uncongested network. The rest of the Chapter is organized as follows. Section 7.1 shows experimental results for drop-tail that serve as base results in evaluating AQM algorithms. Sections 7.2 and 7.3 present results for PI, REM, ARED, LQD, and DCN when they were used with and without ECN. Section 7.4 compares results of all AQM algorithms and section 7.5 concludes the Chapter.

7.1 Results for Drop-Tail

Figures 7.2, 7.3, and 7.4 show experimental results for drop-tail with general TCP traffic at 80%, 90%, and 95% offered loads. These results were obtained with a drop-tail queue of 24 and 240 packets.

At 80% load, drop-tail with a queue length of 240 packets delivered performance that was identical to the performance of the uncongested network. However, drop-tail with a queue length of 24 packets gave significantly worse performance. This result demonstrate the fact that application performance can be deteriorated significantly by an underprovisioned buffer. This result stands in stark contrast with result in Chapter 6 where, under identical conditions, a drop-tail queue of 24 packets and a link-level buffer of 254 packets gave very

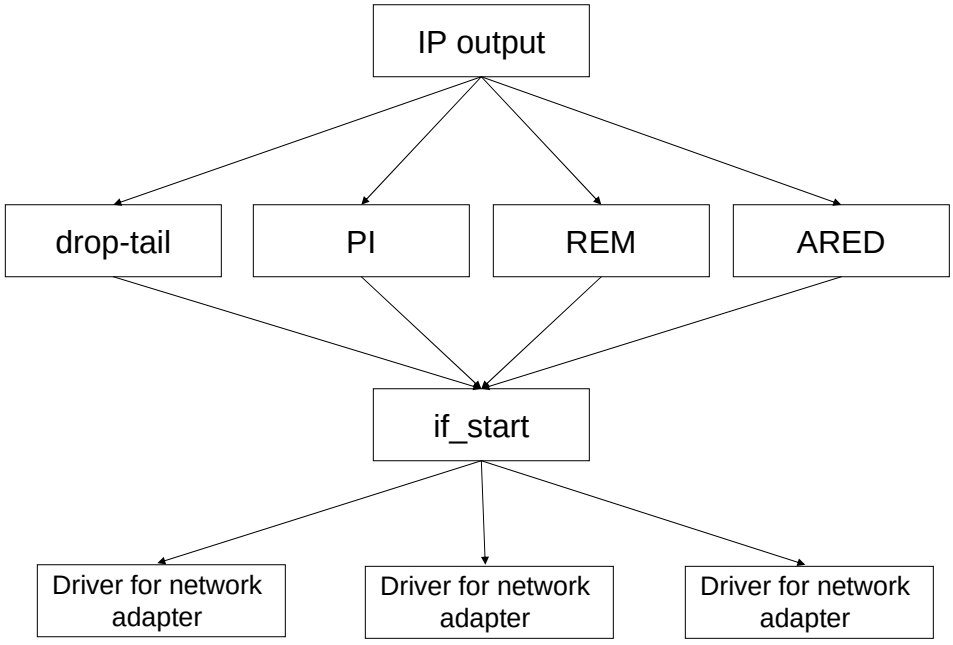

Figure 7.1: ALTQ principal architecture

good results. These good results were achieved because the link-level buffer masked the underprovisioned drop-tail queue.

At 90%, drop-tail with a queue length of 240 packets suffered a small performance degradation. However, drop-tail with a queue length of 240 packets still delivered good performance and closely approximated the performance of the uncongested network at this load. Drop-tail with a queue length of 24 packets continued to be an underprovisioned buffer. It gave poor performance and significantly underperformed drop-tail with a queue length of 240 packets.

As the offered load increased to 95%, the performance for drop-tail with a queue length of 240 packets degraded noticeably. However, drop-tail with a queue length of 240 packets still gave reasonably good performance at this load. Drop-tail with a queue length of 24 packets once again was underprovisioned and gave significantly worse performance than drop-tail with a queue length of 240 packets.

7.2 Results for ARED, PI, LQD, and REM

Experiments from 6 were repeated with the new implementation of ALTQ and AQM algorithms for PI, REM, ARED, LQD, and DCN at 80%, 90%, and 95% loads. Experiments for PI, REM, LQD, and DCN were performed with a queue reference of 24 and 240 packets.

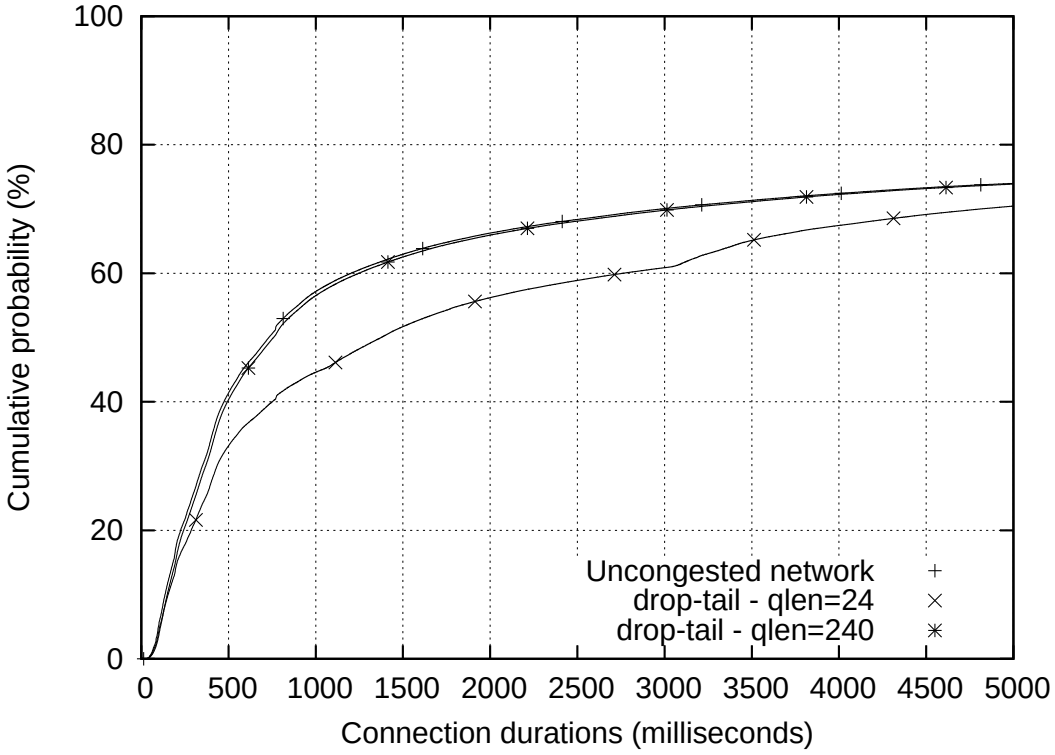

Figure 7.2: Drop-tail performance at 80% load

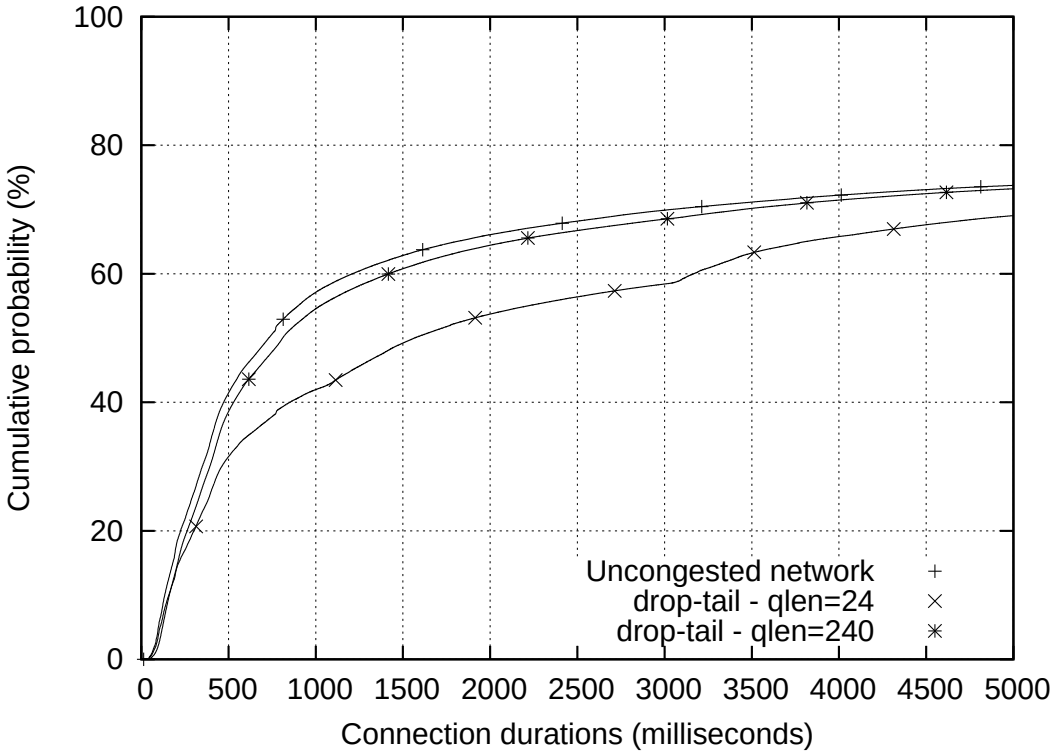

Figure 7.3: Drop-tail performance at 90% load

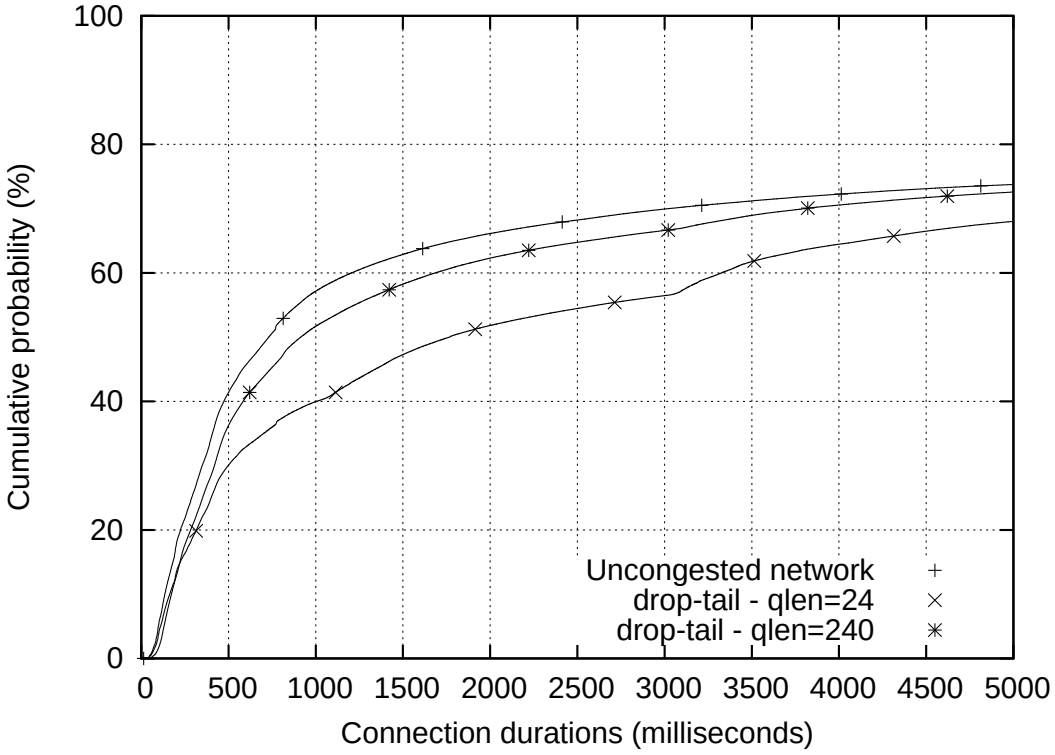

Figure 7.4: Drop-tail performance at 95% load

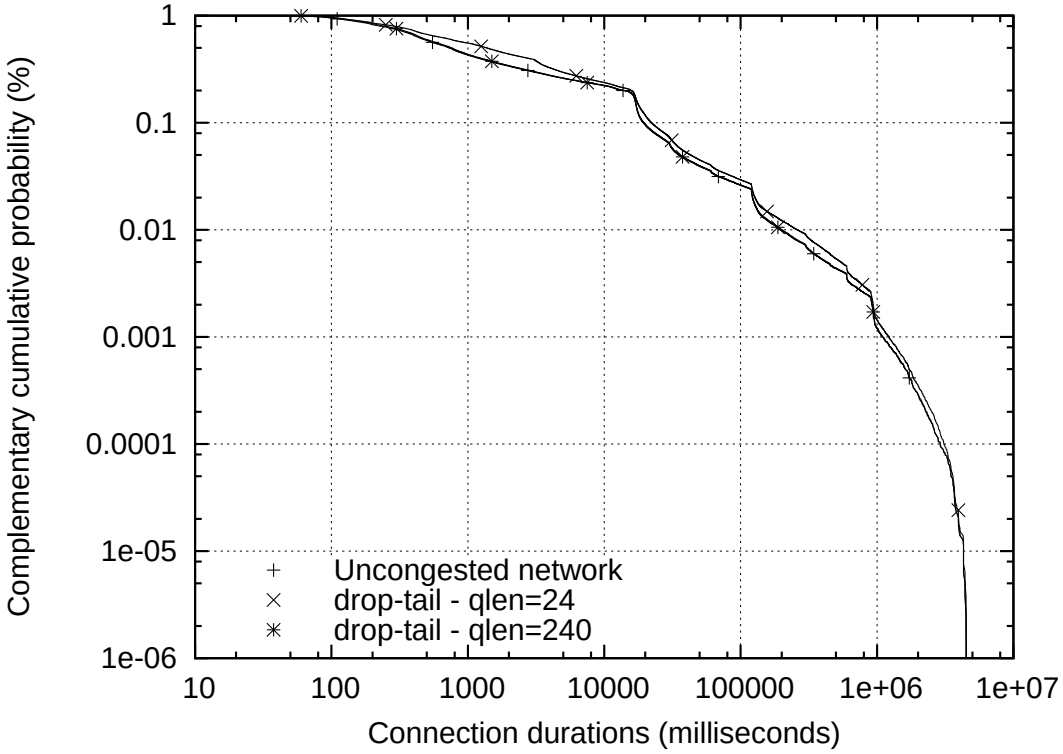

Figure 7.5: Drop-tail performance at 80% load (CCDF)

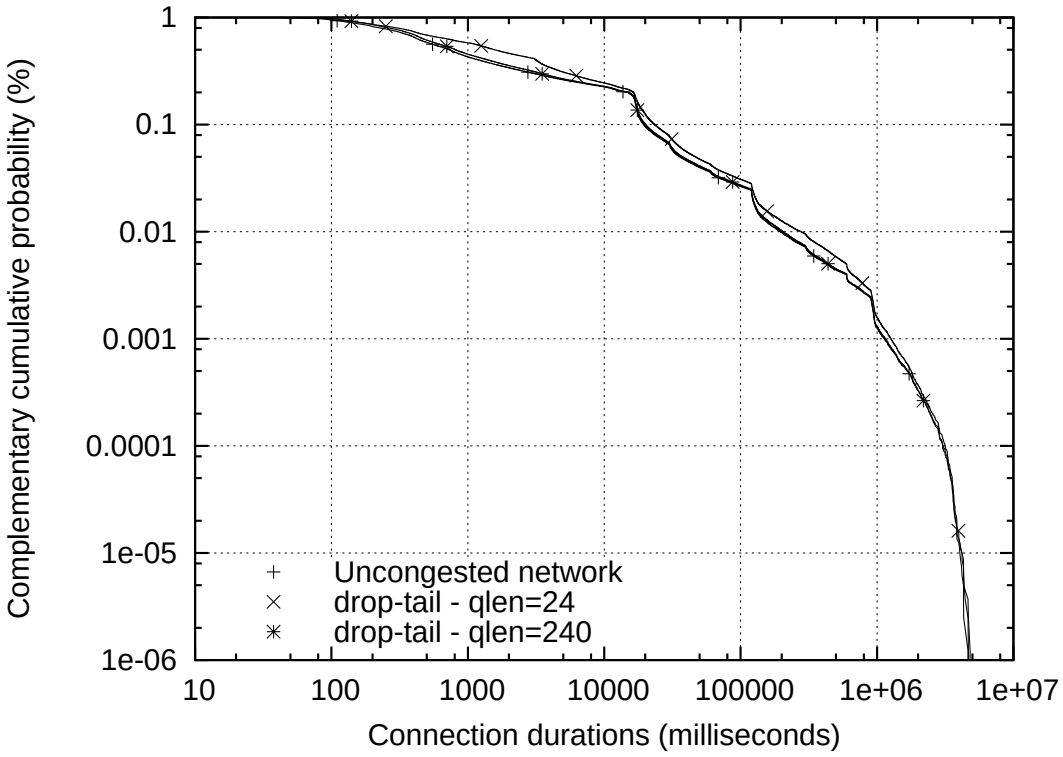

Figure 7.6: Drop-tail performance at 90% load (CCDF)

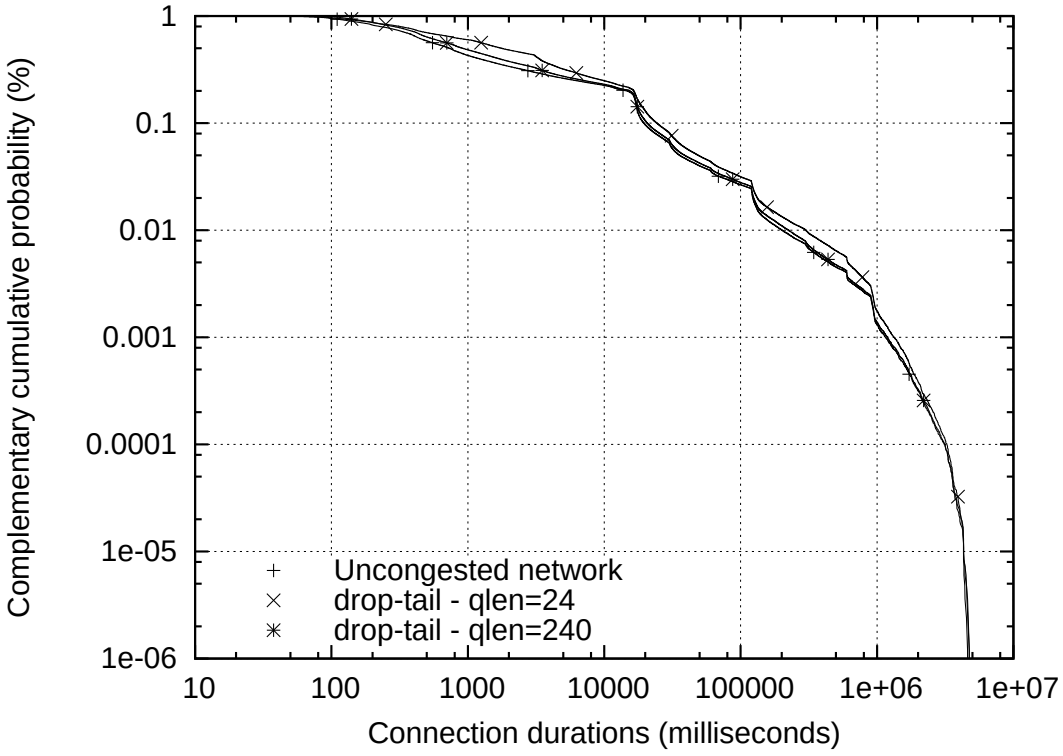

Figure 7.7: Drop-tail performance at 95% load (CCDF)

Experiments for ARED were run in both "packet mode" and "byte mode" with parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets) and ($th_{min} = 120$ packets, $th_{max} = 360$ packets). These parameter settings for ARED achieved a target queue reference of 24 and 240 packets respectively. The results of AQM algorithms were compared with the results of drop-tail and of the uncongested network to investigate the effects of AQM algorithms on general TCP applications when link-level buffering is limited (contrary to results in Chapter 6 where results were obtained with a considerable amount of link-level buffering).

7.2.1 Results for PI

Figures 7.8, 7.9, and 7.10 show experimental results for PI with general TCP applications at 80%, 90%, and 95% loads. These results were obtained without link-level buffering and with a queue reference of 24 and 240 packets for PI.

At 80% offered load, PI obtained equally good performance with both queue references of 24 and 240 packets. The performance for PI was identical to the performance of the uncongested network and that of drop-tail with a queue length of 240 packets at this load.

At 90% offered load, PI delivered similar performance with both queue references of 24 and 240 packets. The performance for PI was comparable to that of drop-tail with a queue length of 240 packets and came close to the performance of the uncongested network.

As the offered load increased to 95%, the performance for PI with both queue references degraded slightly. PI with a queue reference of 24 packets gave slightly better performance than with a queue reference of 240 packets for 60% of flows. These were flows that completed within 1,500 milliseconds or less. For the other 40% of flows, PI delivered similar performance with both queue references. Further, PI obtained slightly better performance than drop-tail with both queue references at this load.

7.2.2 Results for REM

Figures 6.15, 6.16, and 6.17 show experimental results for REM with general TCP applications at 80%, 90%, and 95% when REM was used with packet drops and without link-level buffering. These results were obtained for REM with a queue reference of 24 and 240 packets.

At 80% offered load, REM gave identical performance with queue references of 24 and 240 packets. Further, the performance for REM was indistinguishable from that of the uncongested network and of drop-tail with a queue length of 240 packets at this load.

At 90% offered load, REM delivered very similar performance with queue references of 24 and 240 packets. The performance for REM was slightly better than that of drop-tail with a queue length of 240 packets and came close to the performance of the uncongested network.

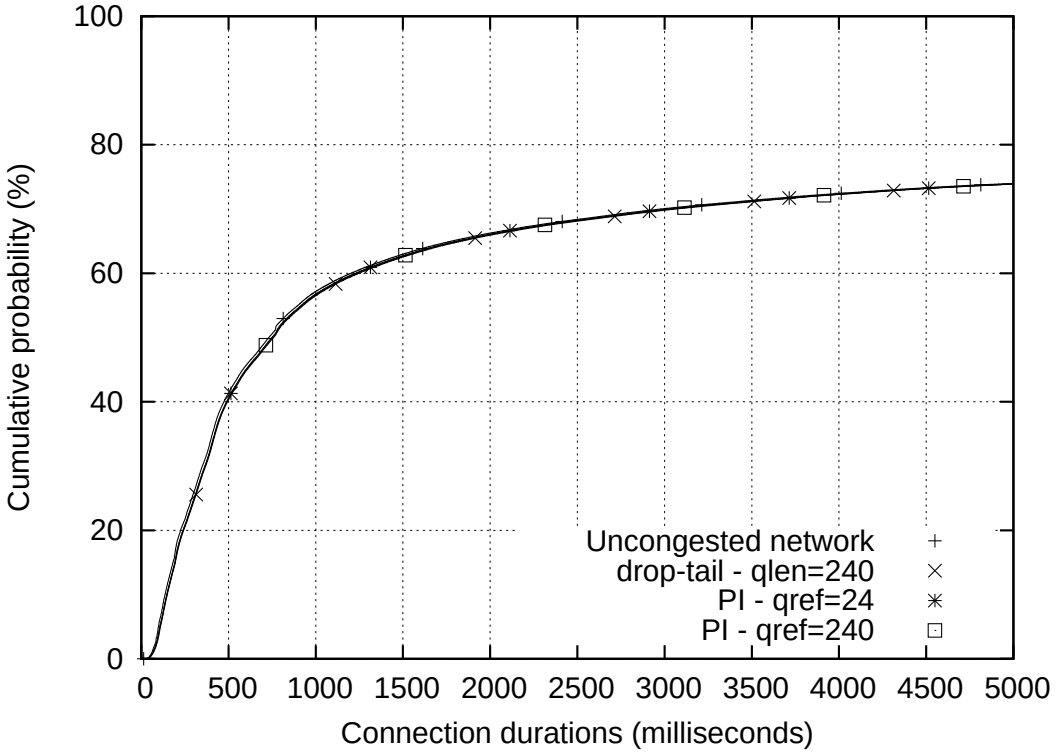

Figure 7.8: PI performance at 80% load

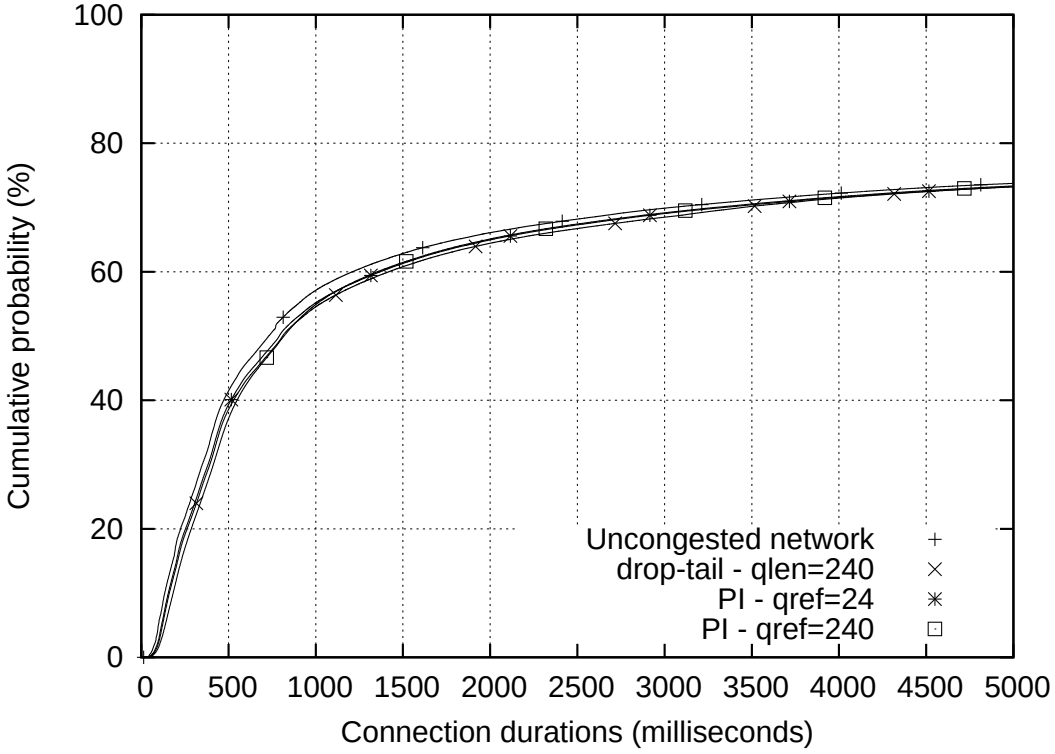

Figure 7.9: PI performance at 90% load

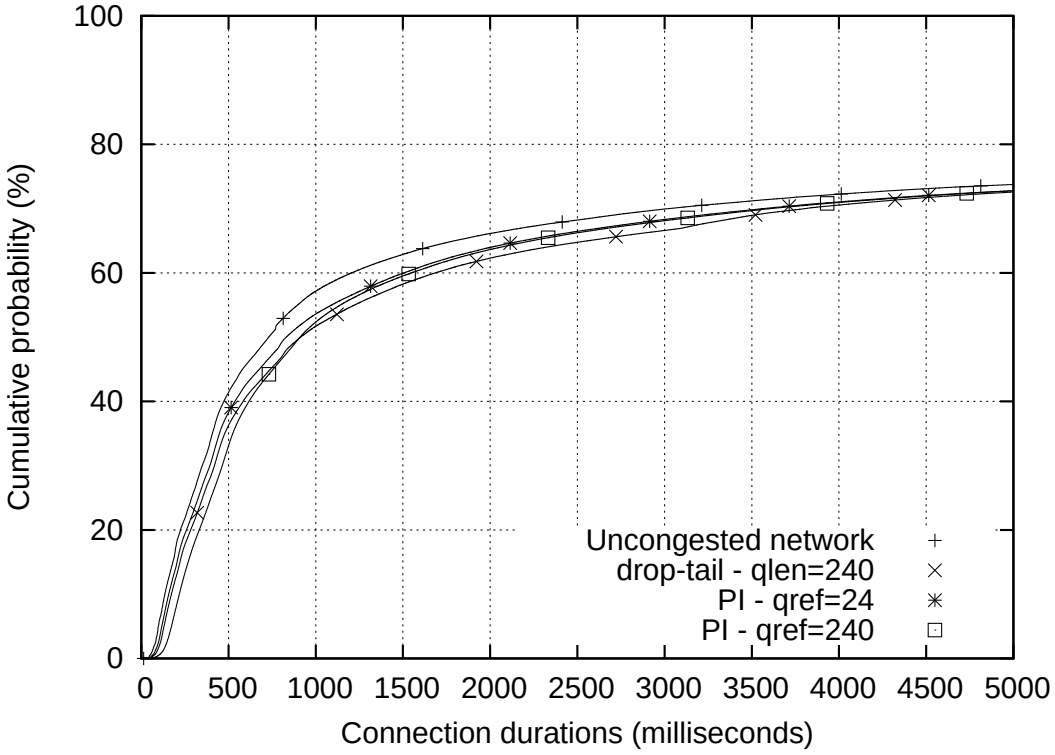

Figure 7.10: PI performance at 95% load

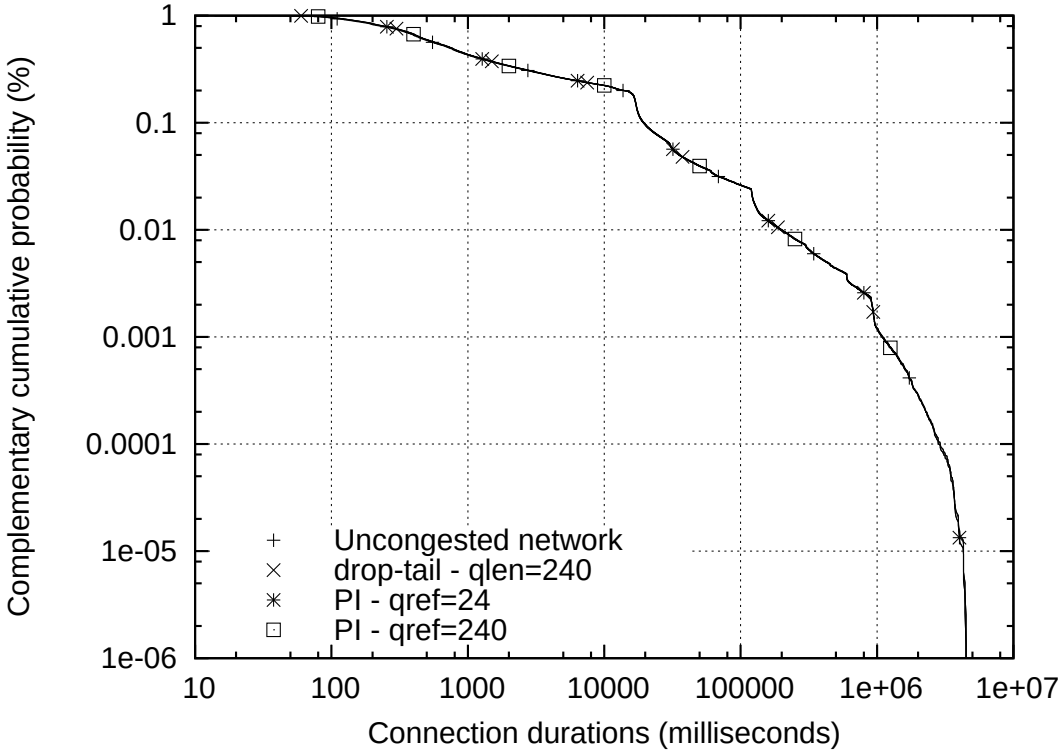

Figure 7.11: PI performance at 80% load (CCDF)

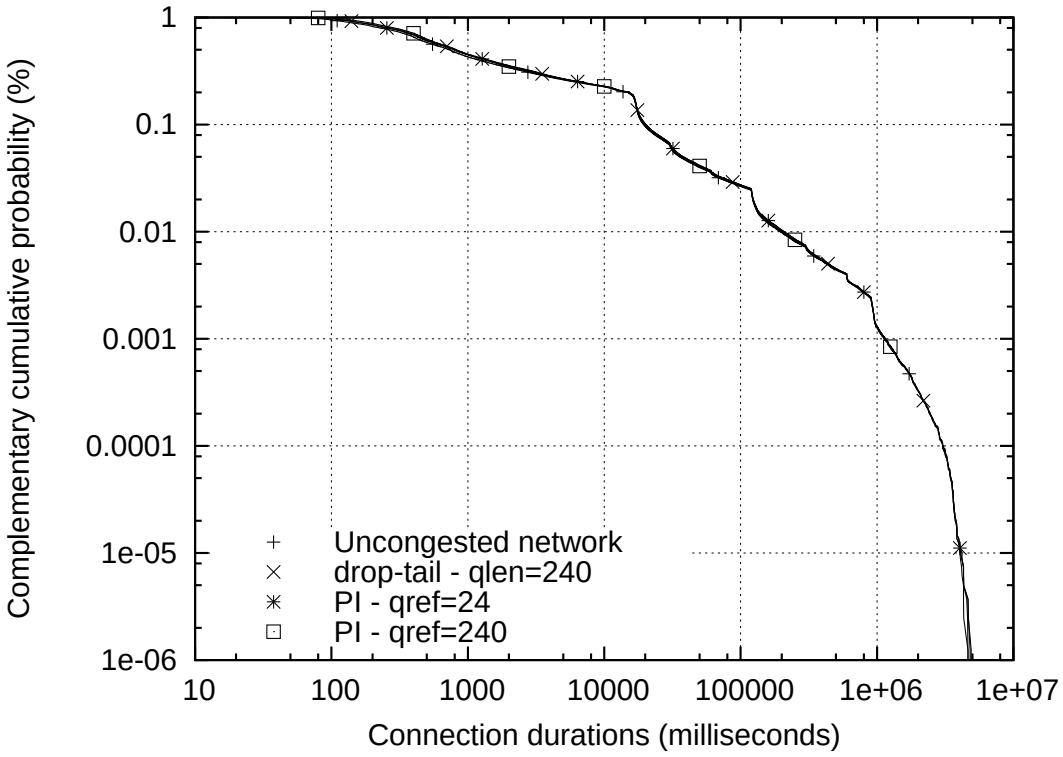

Figure 7.12: PI performance at 90% load (CCDF)

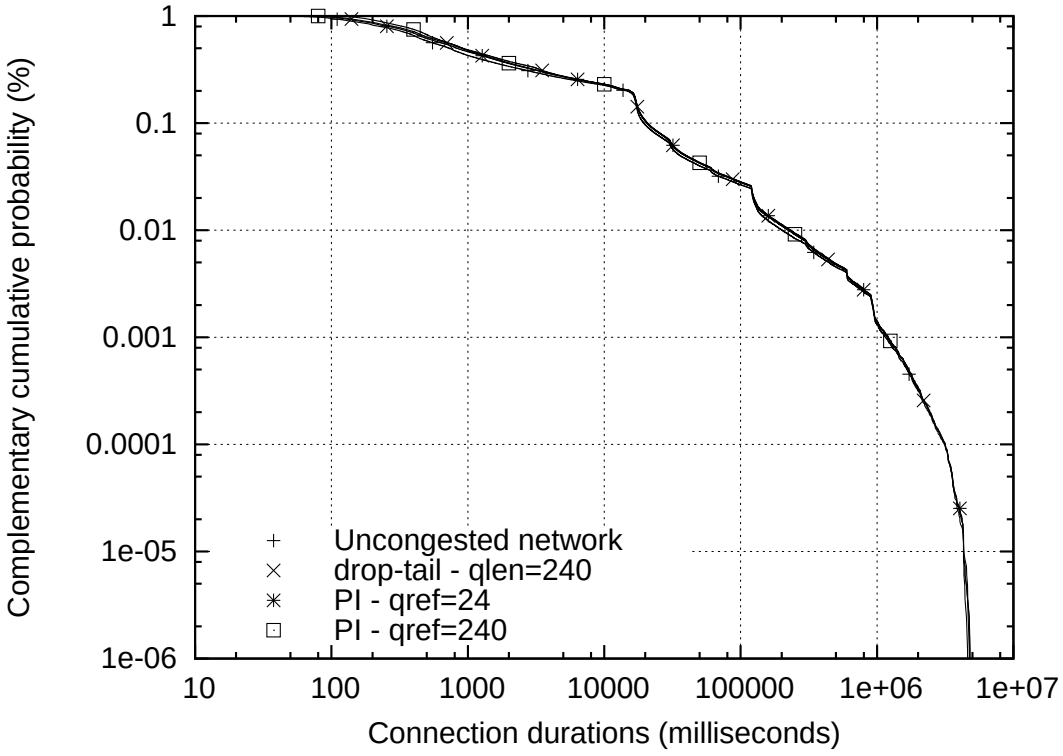

Figure 7.13: PI performance at 95% load (CCDF)

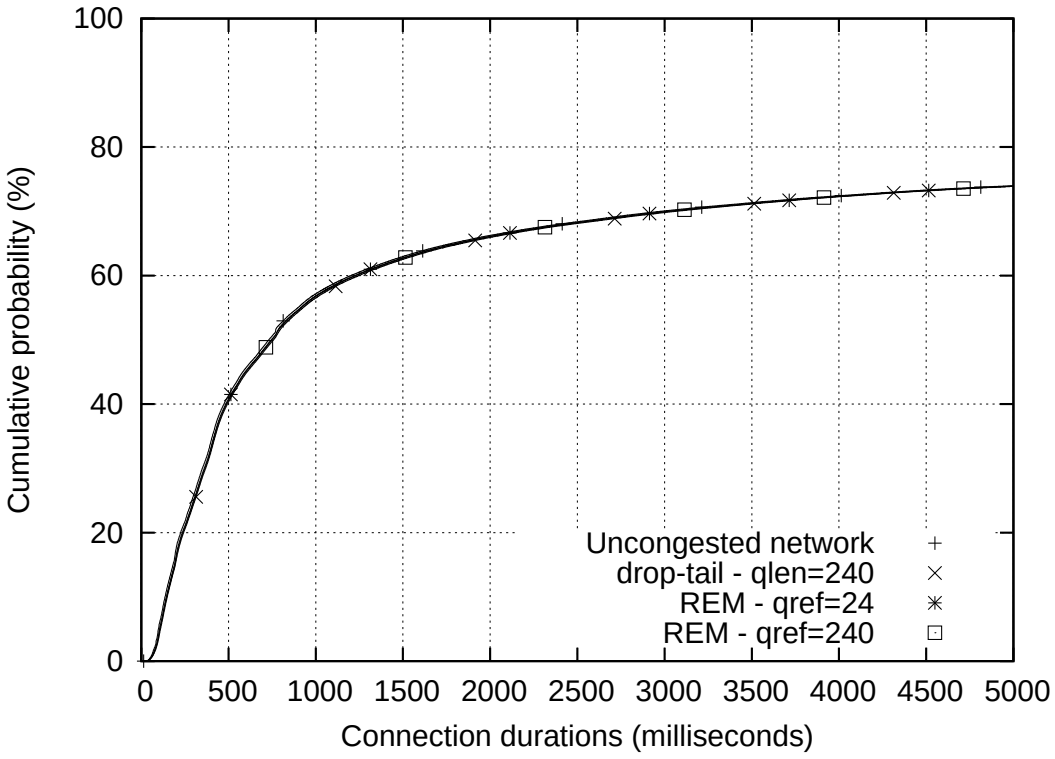

Figure 7.14: REM performance at 80% load

As the offered load increased to 95%, the performance for REM with both queue references degraded slightly. REM obtained slightly performance with a queue reference of 24 packets than with a queue reference of 240 packets for approximately 55% of flows that completed within 1 second or less. Overall, REM gave better performance than drop-tail with both queue references at this load.

7.2.3 Results for ARED

Figures 7.20, 7.21, and 7.22 show experimental results for ARED "packet mode" and "byte mode" when experiments were performed with packet drops and without link-level buffering. ARED "packet mode" and "byte mode" were operated with parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets) and ($th_{min} = 120$ packets, $th_{max} = 360$ packets). These parameter settings gave an implicit target queue reference of 24 and 240 packets for ARED.

At 80% load, ARED "packet mode" obtained identical performance with both parameter settings. The performance for ARED "packet mode" at this load was undistinguishable from that of drop-tail and of the uncongested network.

At 90% offered load, the performance for ARED "packet mode" with parameter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets) degraded slightly and was comparable to

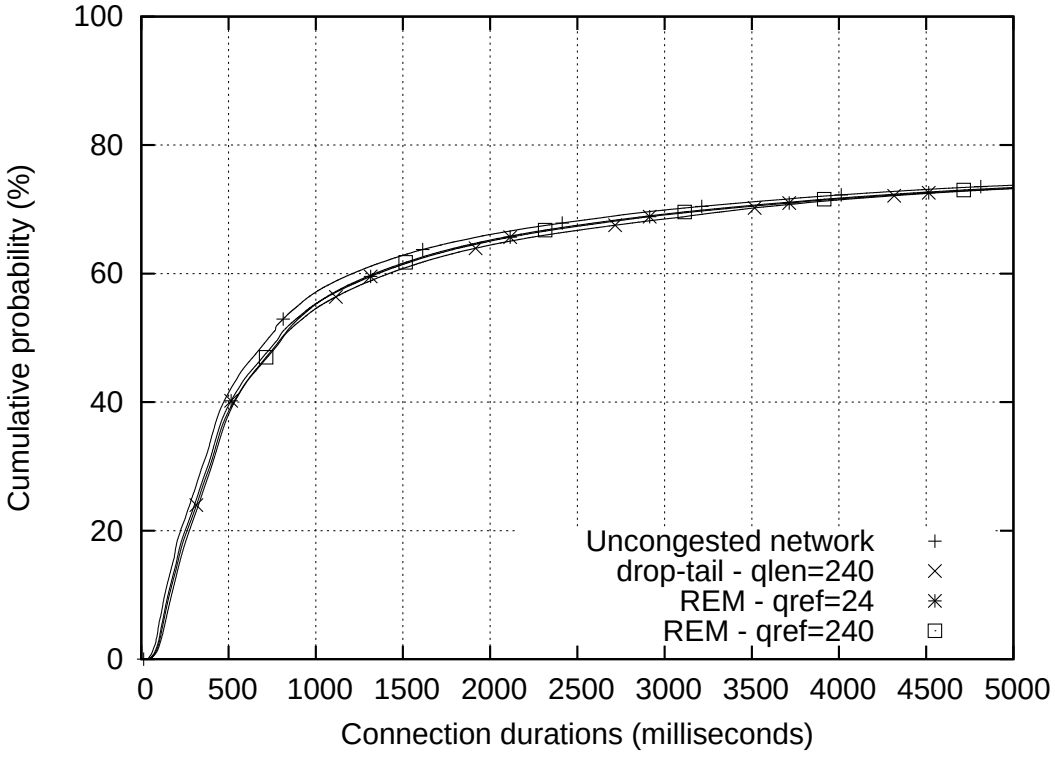

Figure 7.15: REM performance at 90% load

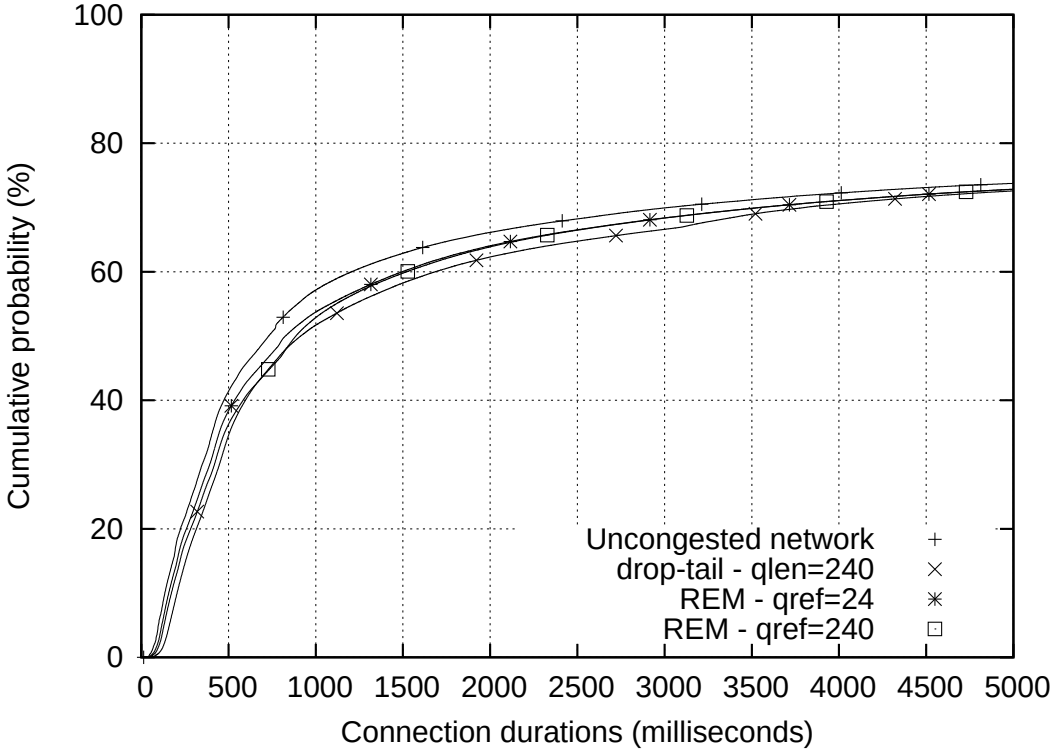

Figure 7.16: REM performance at 95% load

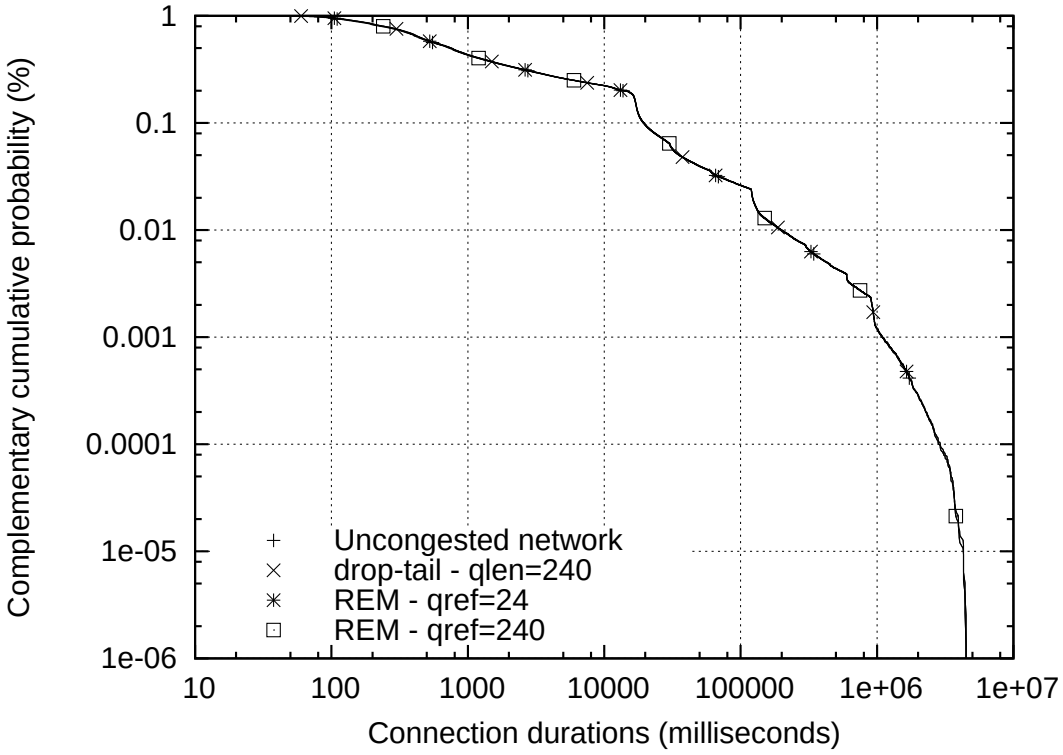

Figure 7.17: REM performance at 80% load (CCDF)

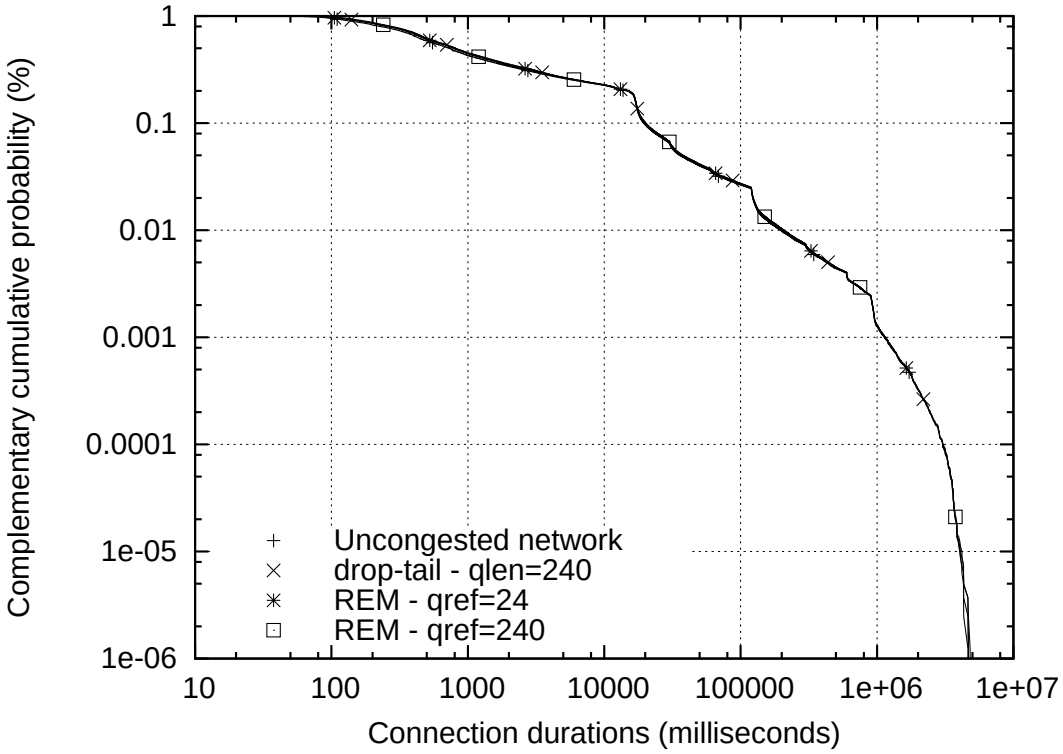

Figure 7.18: REM performance at 90% load (CCDF)

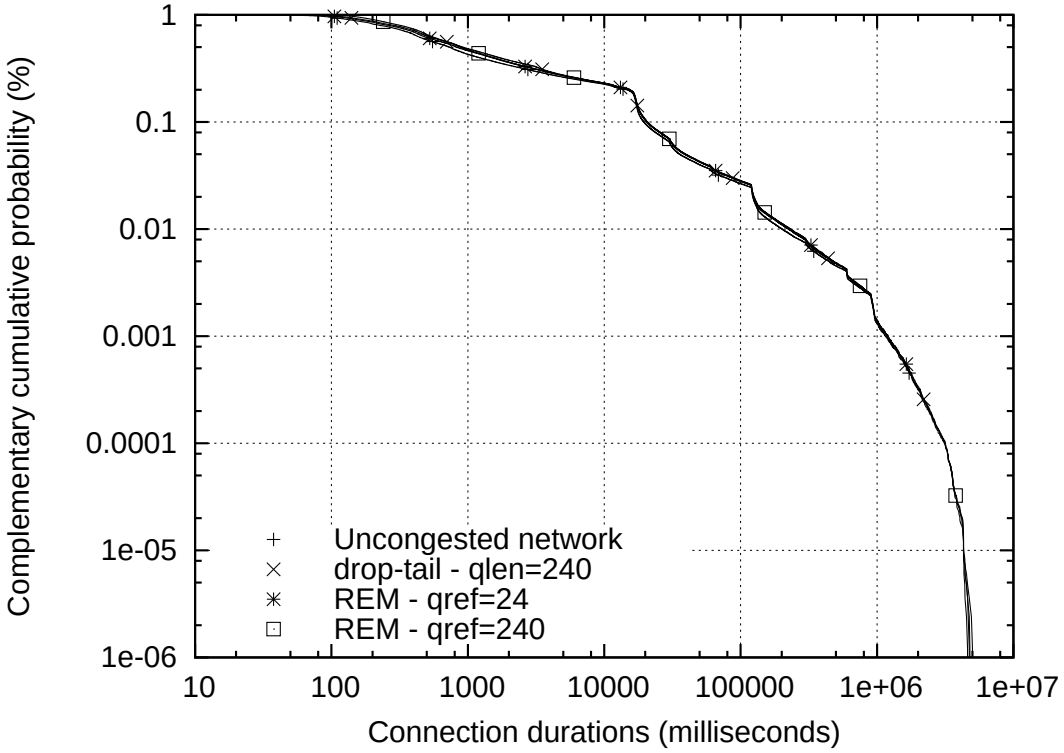

Figure 7.19: REM performance at 95% load (CCDF)

the performance of drop-tail with a queue length of 240 packets. ARED "packet mode" with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) suffered considerable performance degradation and uderperformed drop-tail at this load.

As the offered increased to 95%, ARED "packet mode" with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) suffered noticeable performance degradation but obtained about the same performance as drop-tail with a queue length of 240 packets. The performance for ARED "packet mode" with parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets) degraded even more and was considerably lower than the performance of drop-tail at this load.

When compared with ARED "packet mode", ARED "byte mode" did not give any performance improvement at 80% offered load. With both parameter settings ($th_{min} = 12$ packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets), ARED "packet mode" and "byte mode" gave identical performance as drop-tail and the uncongested network.

At 90% offered load, ARED "byte mode" performed slightly better with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) than with (th_{min} = 12 packets, $th_{max} = 36$ packets). With parameter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets), ARED "packet mode" and "byte mode" delivered essentially the same performance that was

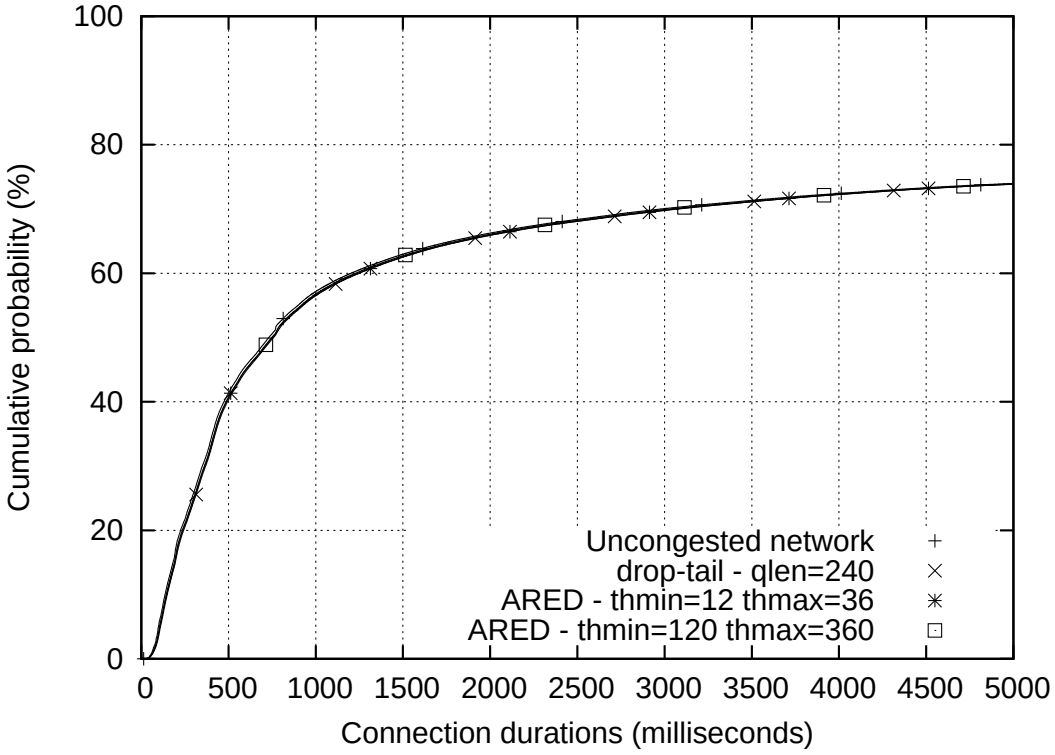

Figure 7.20: ARED performance at 80% load

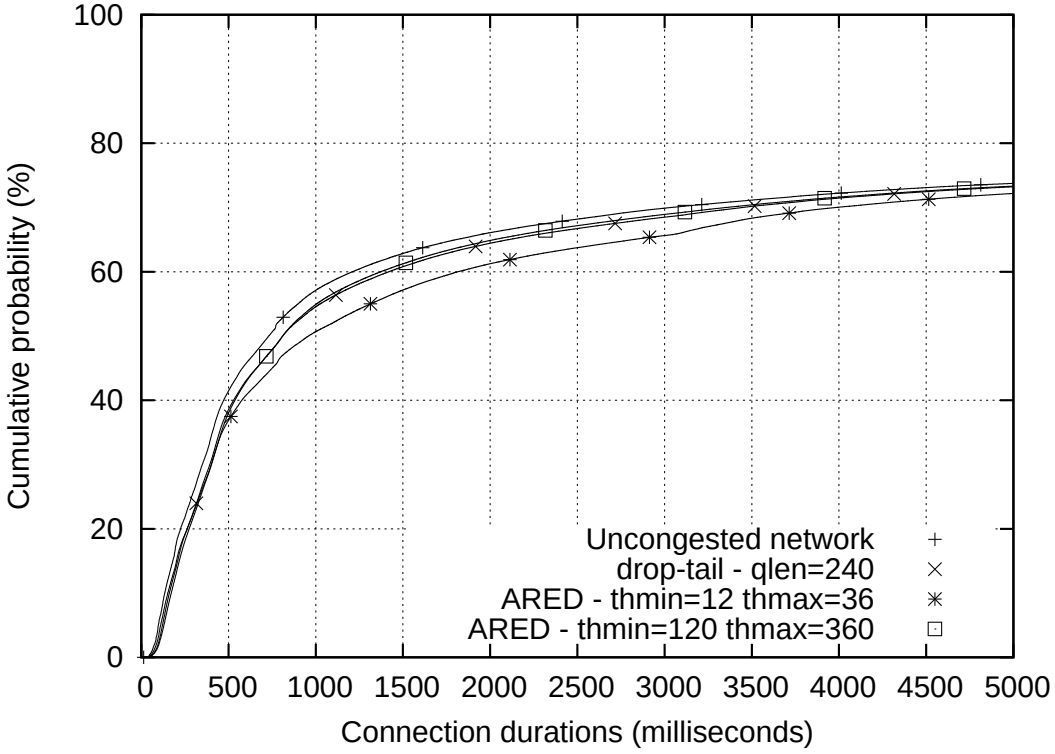

Figure 7.21: ARED performance at 90% load

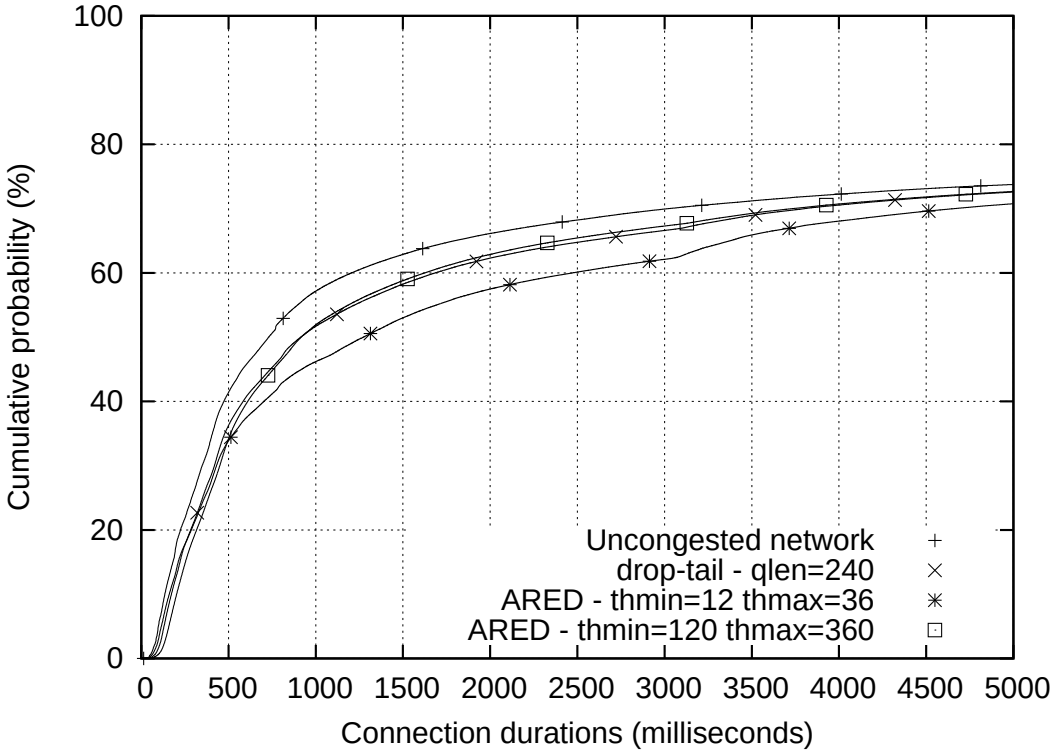

Figure 7.22: ARED performance at 95% load

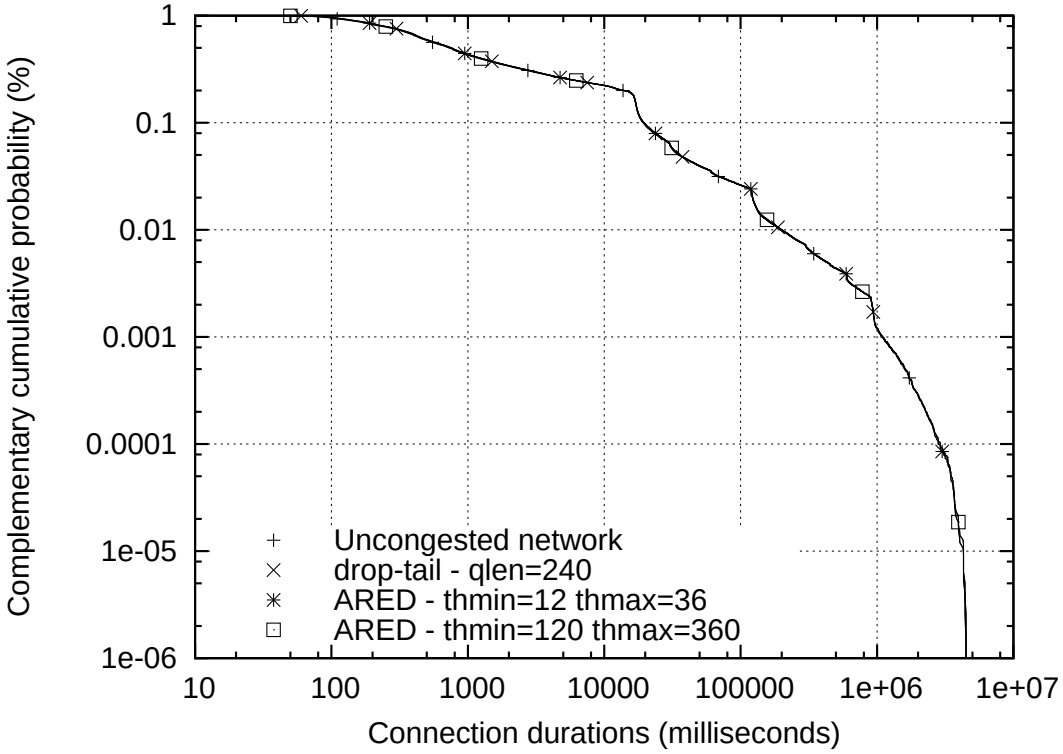

Figure 7.23: ARED performance at 80% load (CCDF)

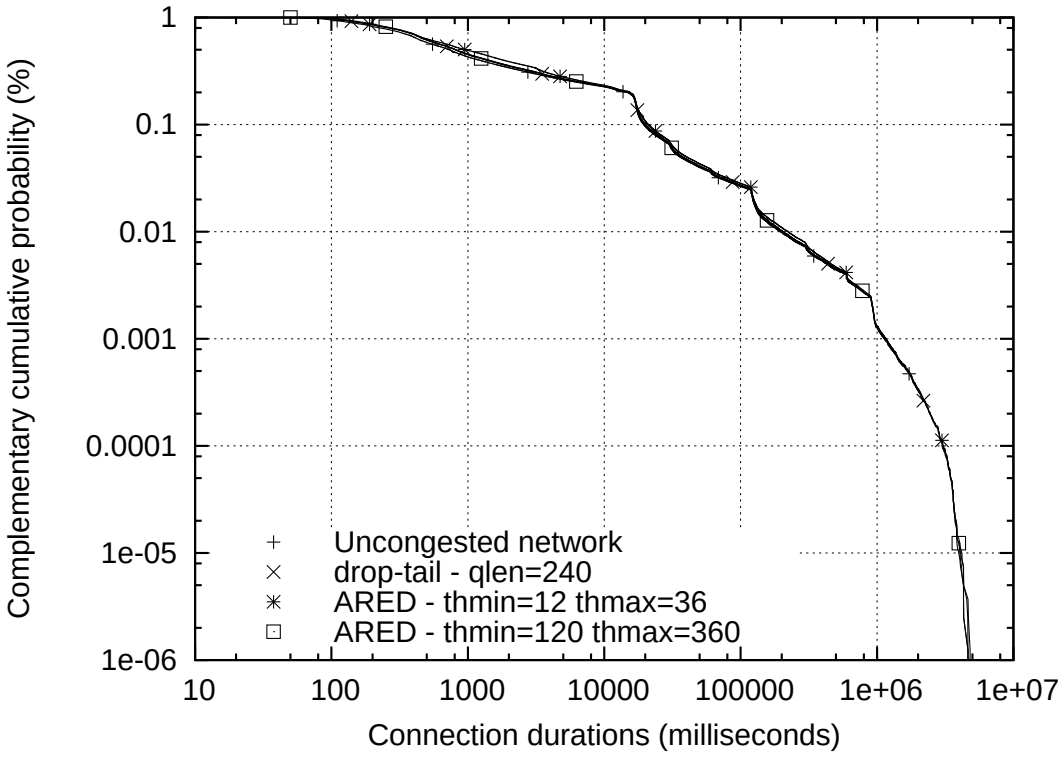

Figure 7.24: ARED performance at 90% load (CCDF)

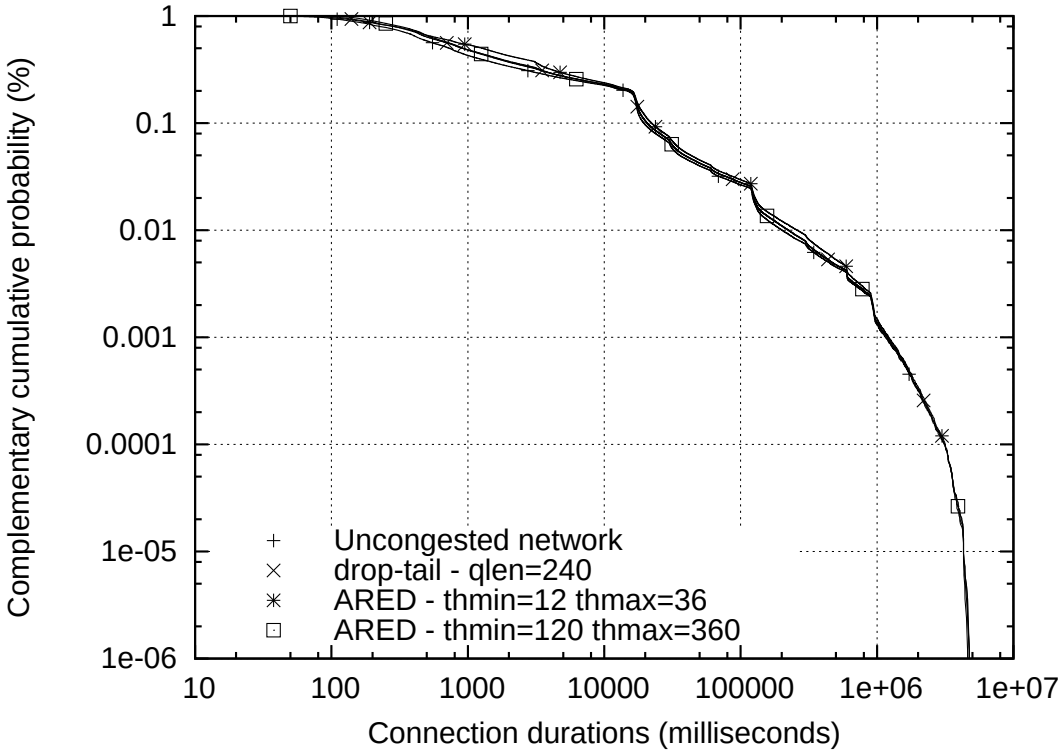

Figure 7.25: ARED performance at 95% load (CCDF)

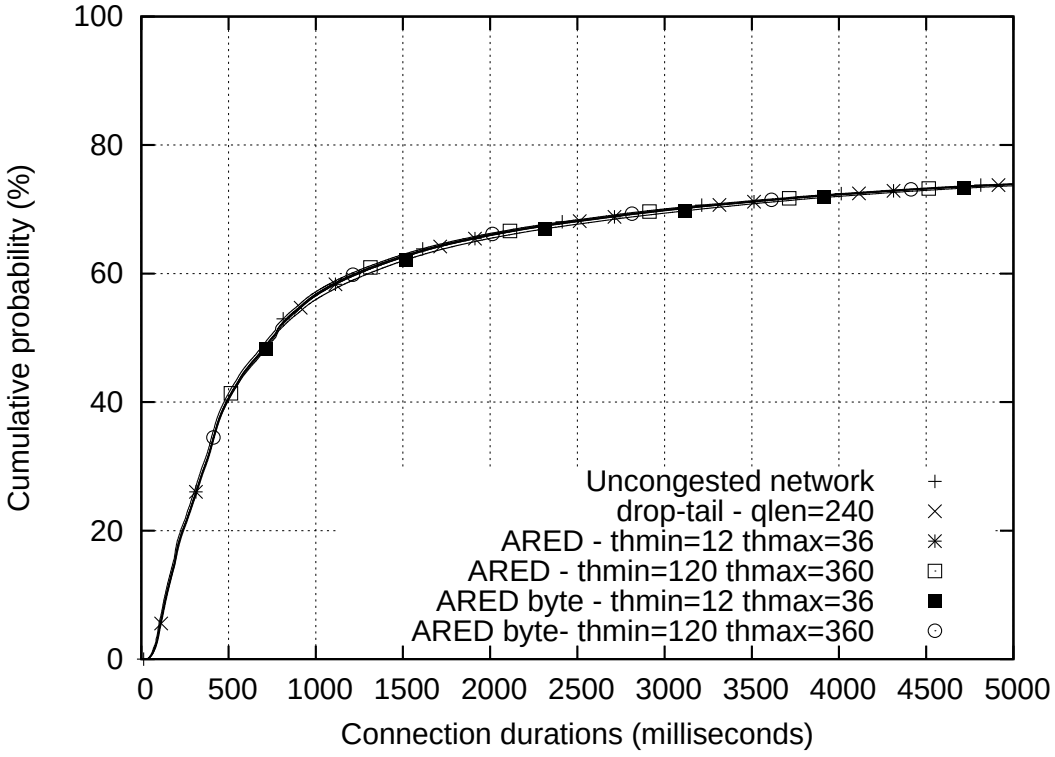

Figure 7.26: ARED byte mode performance at 80% load

also comparable with drop-tail. When used with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets), ARED "byte mode" slightly underperformed drop-tail but obtained slightly better performance than ARED "packet mode".

As the offered load increased to 95%, ARED "byte mode" obtained similar performance with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). Further, the performance for ARED "byte mode" delivered approximately the same performance as drop-tail and ARED "packet mode" with (th_{min} = 120 packets, th_{max} = 360 packets). ARED "packet mode" with (th_{min} = 12 packets, th_{max} = 36 packets) gave considerably worse performance at this load.

7.2.4 Results for LQD

Figures 7.32, 7.33, and 7.34 show experimental results for LQD without link-level buffering and with general TCP applications. These resutls were obtained when LQD operated with packet drops and with a queue reference of 24 and 240 packets.

At 80% offered load, LQD obtained the same performance with both queue references. The performance for LQD at this load was indistinguishable from that of drop-tail and of the uncongested network.

At 90% offered load, LQD delivered very similar performance with both queue references.

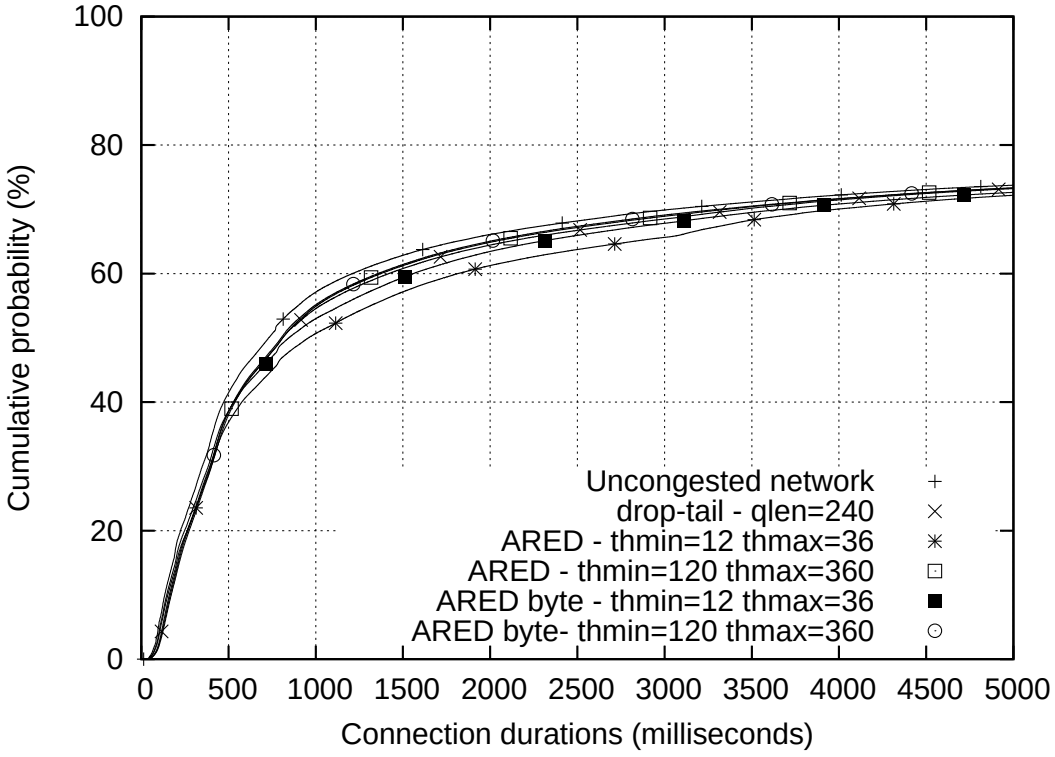

Figure 7.27: ARED byte mode performance at 90% load

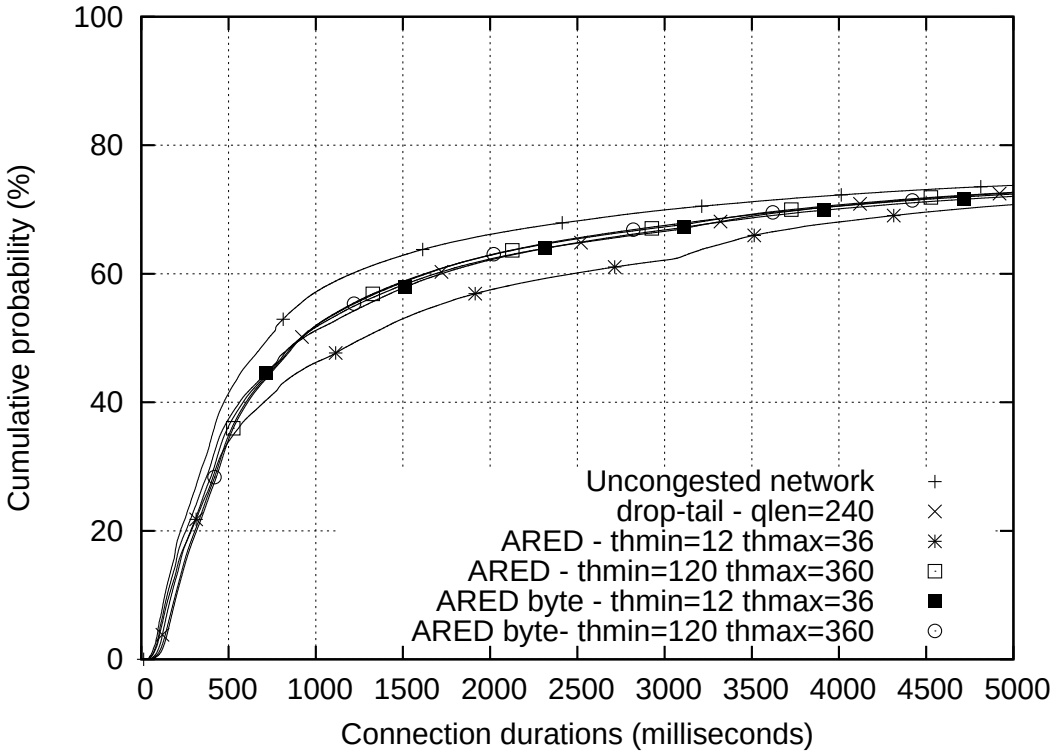

Figure 7.28: ARED byte mode performance at 95% load

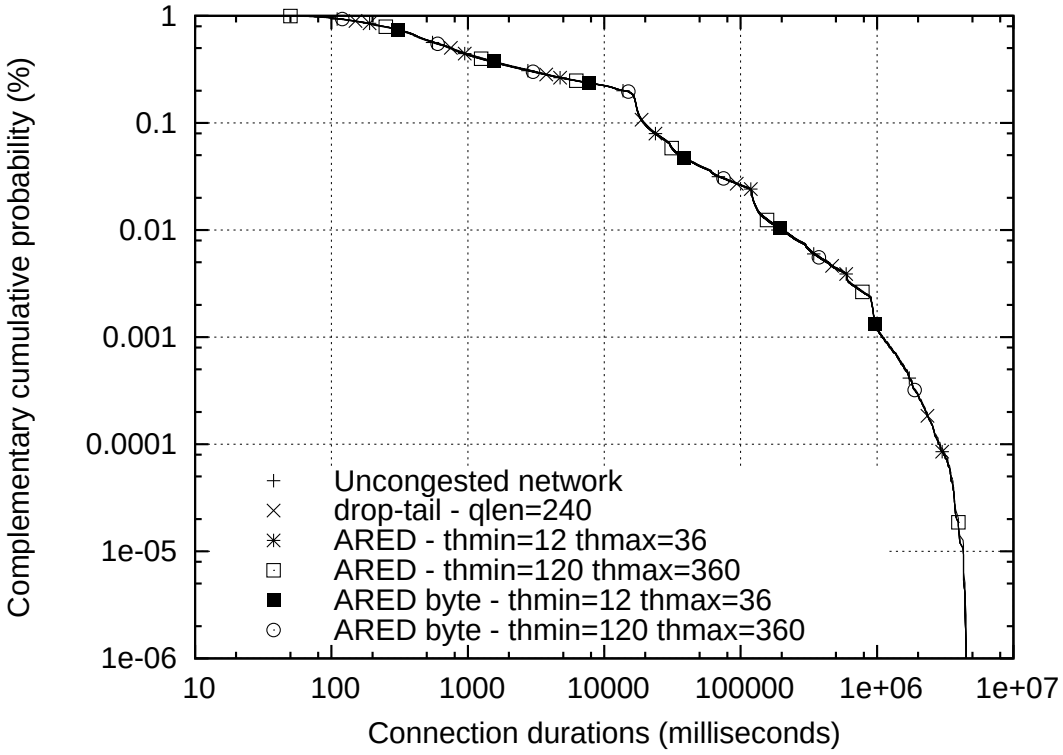

Figure 7.29: ARED byte mode performance at 80% load (CCDF)

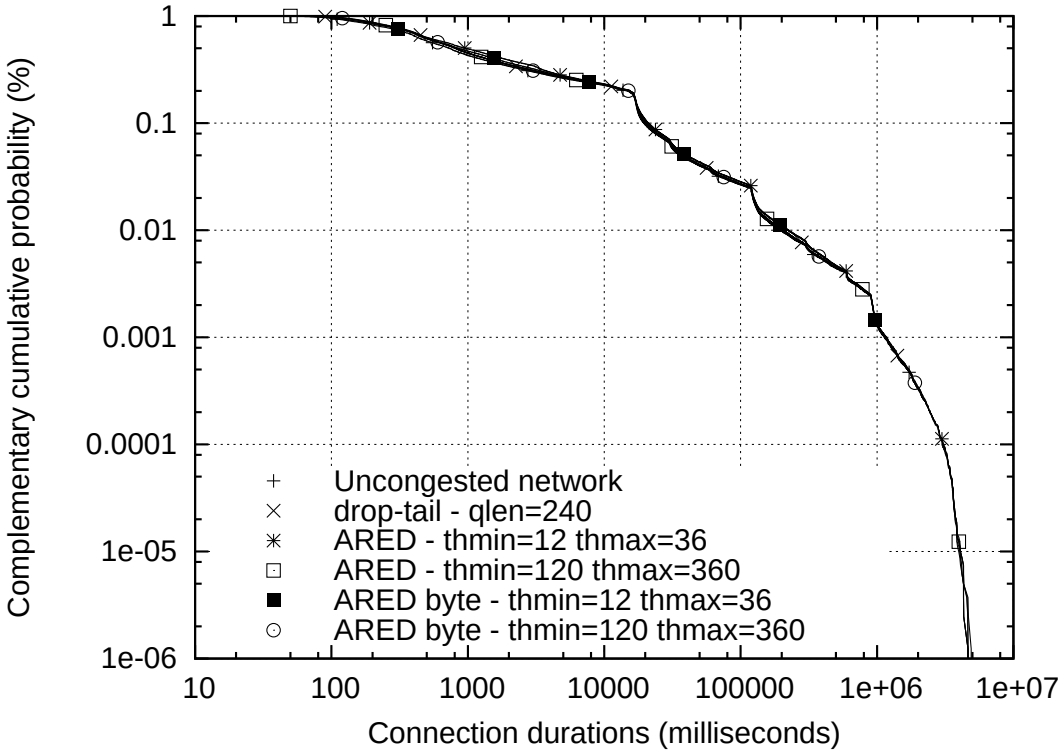

Figure 7.30: ARED byte mode performance at 90% load (CCDF)

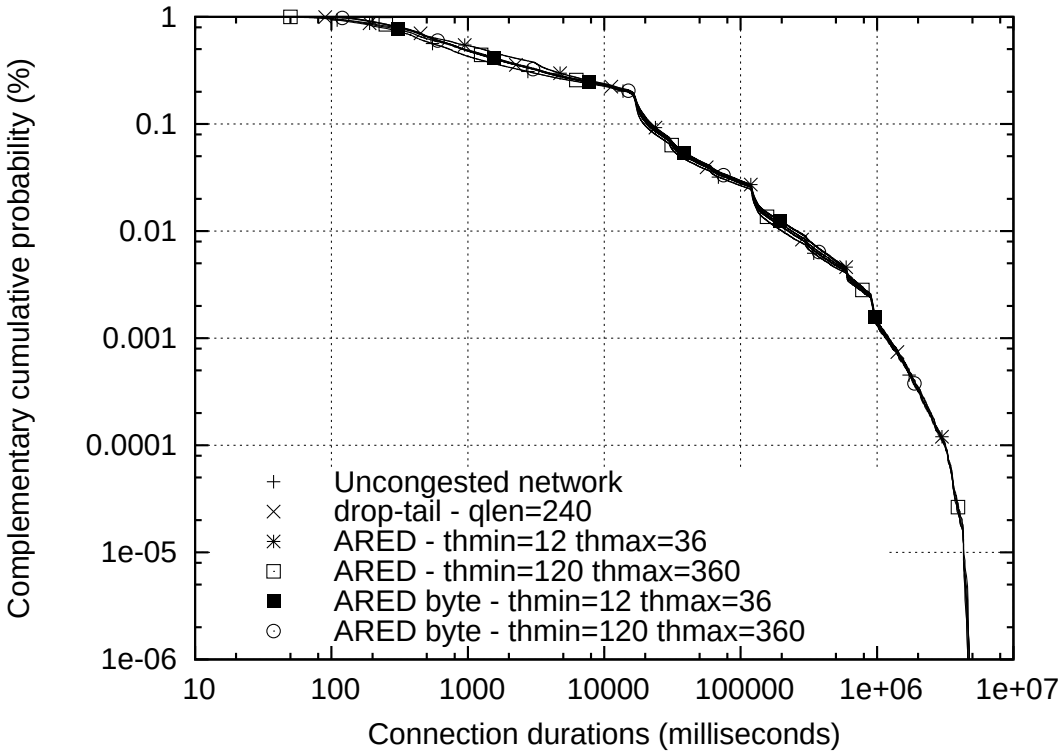

Figure 7.31: ARED byte mode performance at 95% load (CCDF)

The performance for LQD at this load was slightly better than that of drop-tail and closely approximated the performance of the uncongested network.

As the offered load increased to 95%, LQD gave slightly better performance with a queue reference of 24 packets than with a queue reference of 240 packets for approximately 55% of flows. The other 45% of flows experienced the same performance with both queue references for LQD. LQD outperformed drop-tail and came close to the performance of the uncongested network at this load.

7.2.5 Results for DCN

Figures 7.38, 7.39, and 7.40 show experimental results for DCN with general TCP applications and without link-level buffering. These results were obtained when DCN was used with packet drops and with a queue reference of 24 and 240 packets.

At 80% offered load, DCN obtained the same performance with both queue references of 24 and 240 packets. The performance of DCN at this load was identical to that of drop-tail and of the uncongested network.

At 90% offered load, DCN delivered the same performance with both queue references. The performance for DCN at this load was undistinguishable from that of the uncongested network and slightly better than the performance of drop-tail with a queue length of 240

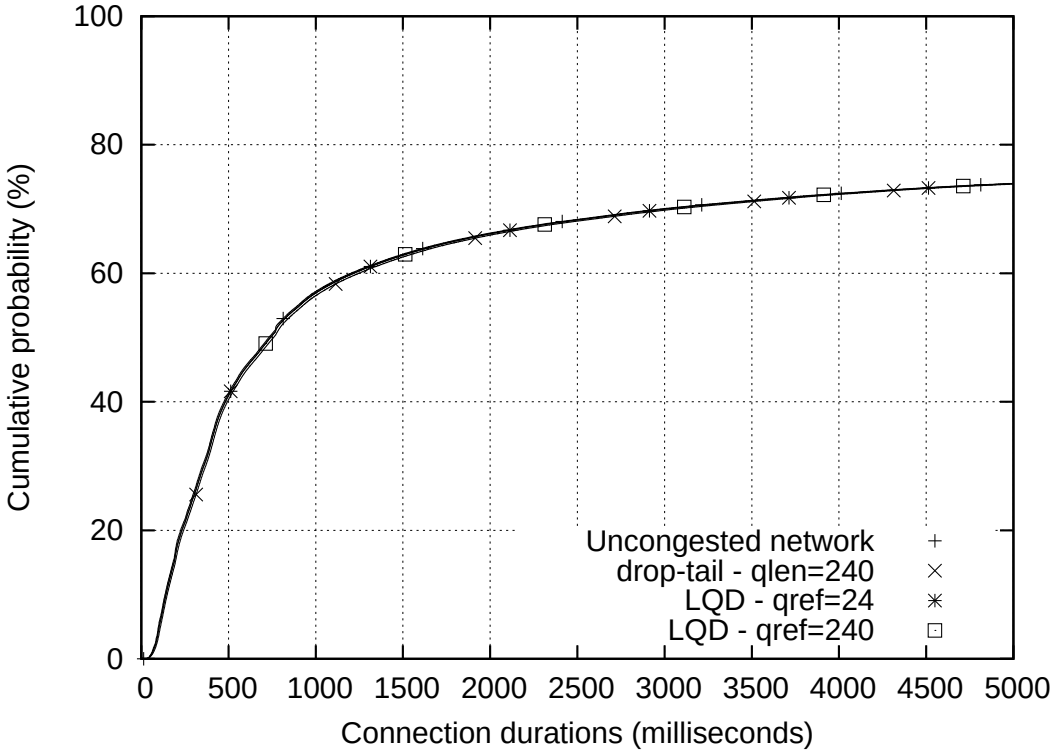

Figure 7.32: LQD performance at 80% load

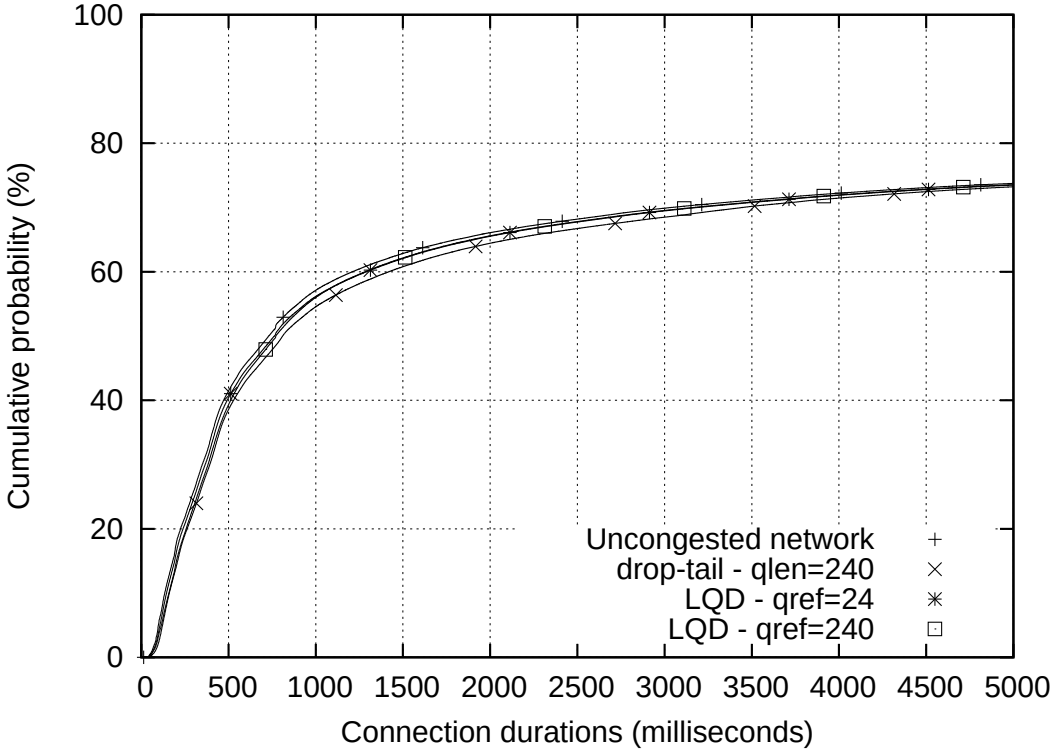

Figure 7.33: LQD performance at 90% load

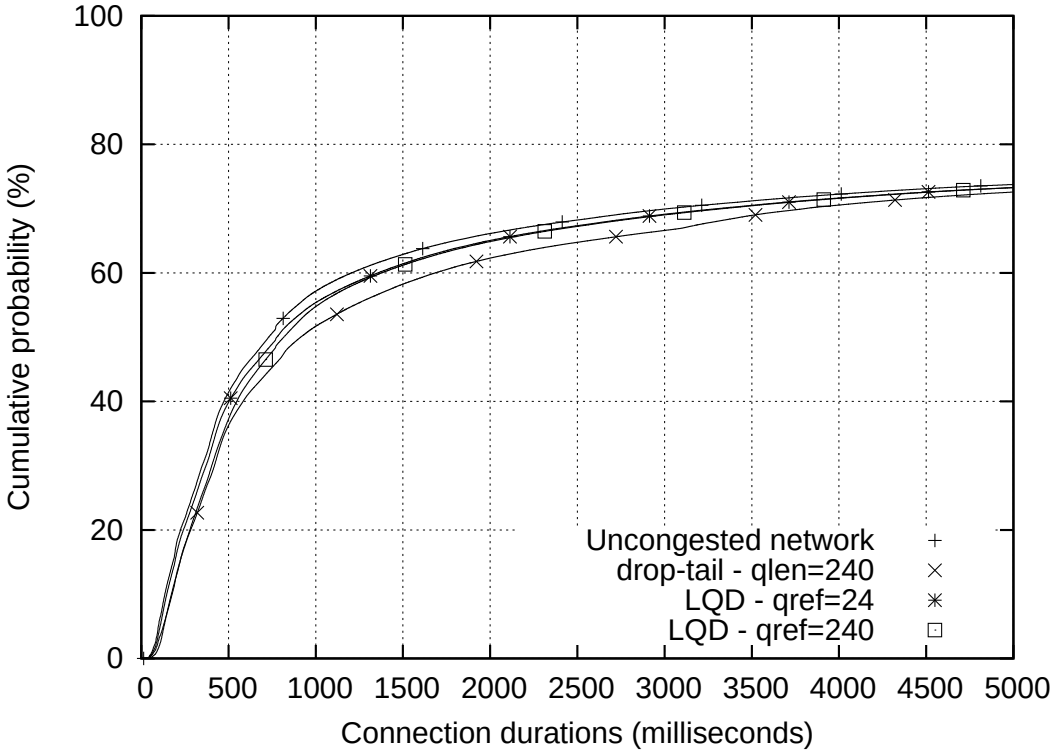

Figure 7.34: LQD performance at 95% load

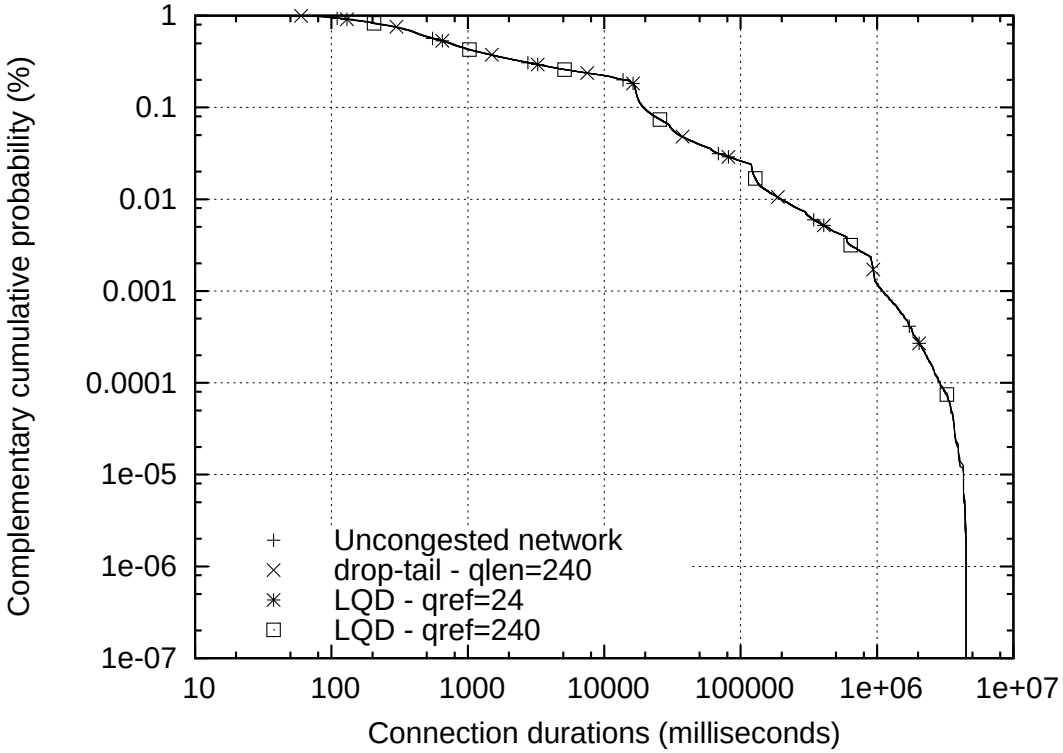

Figure 7.35: LQD performance at 80% load (CCDF)

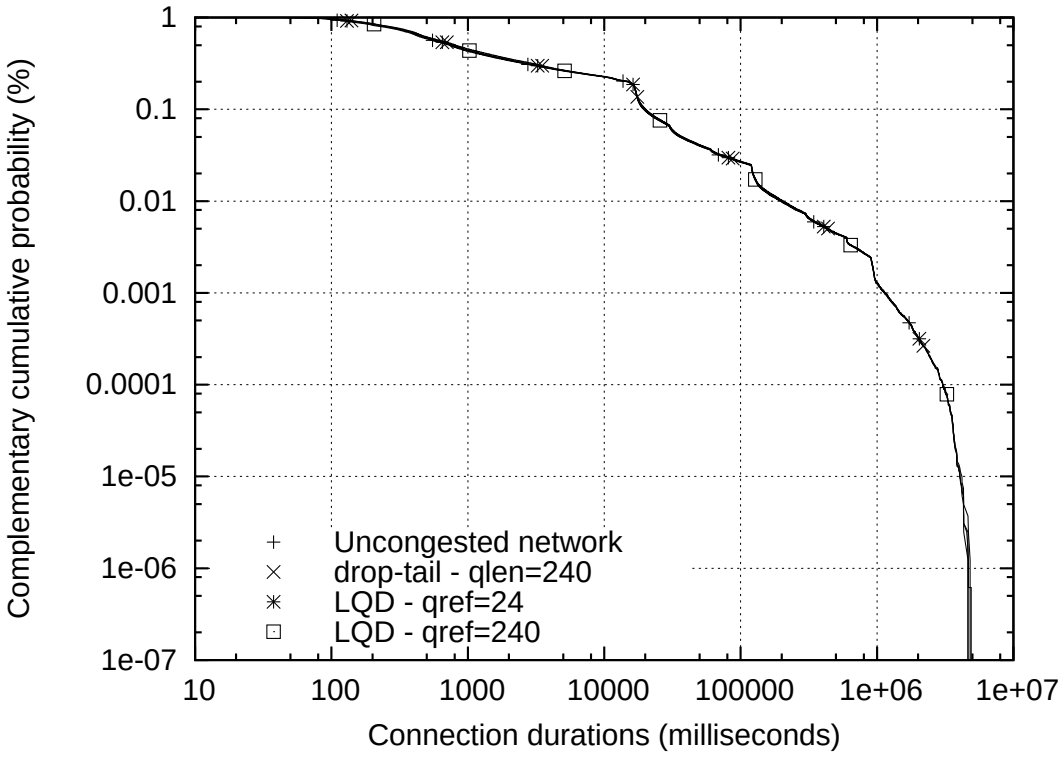

Figure 7.36: LQD performance at 90% load (CCDF)

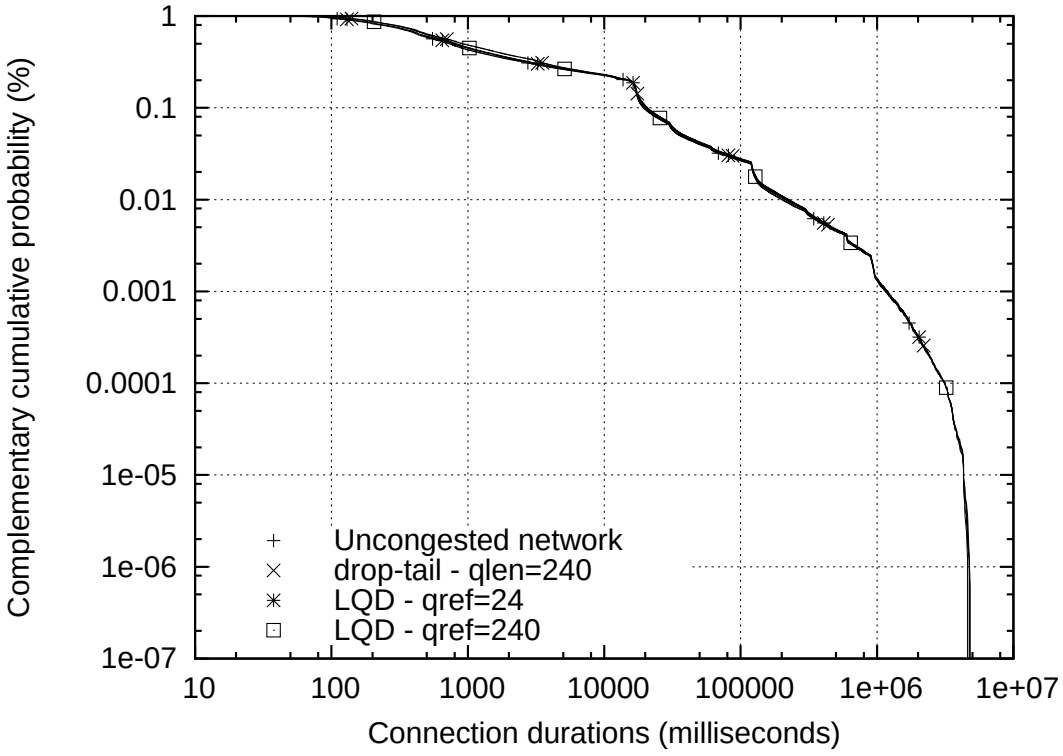

Figure 7.37: LQD performance at 95% load (CCDF)

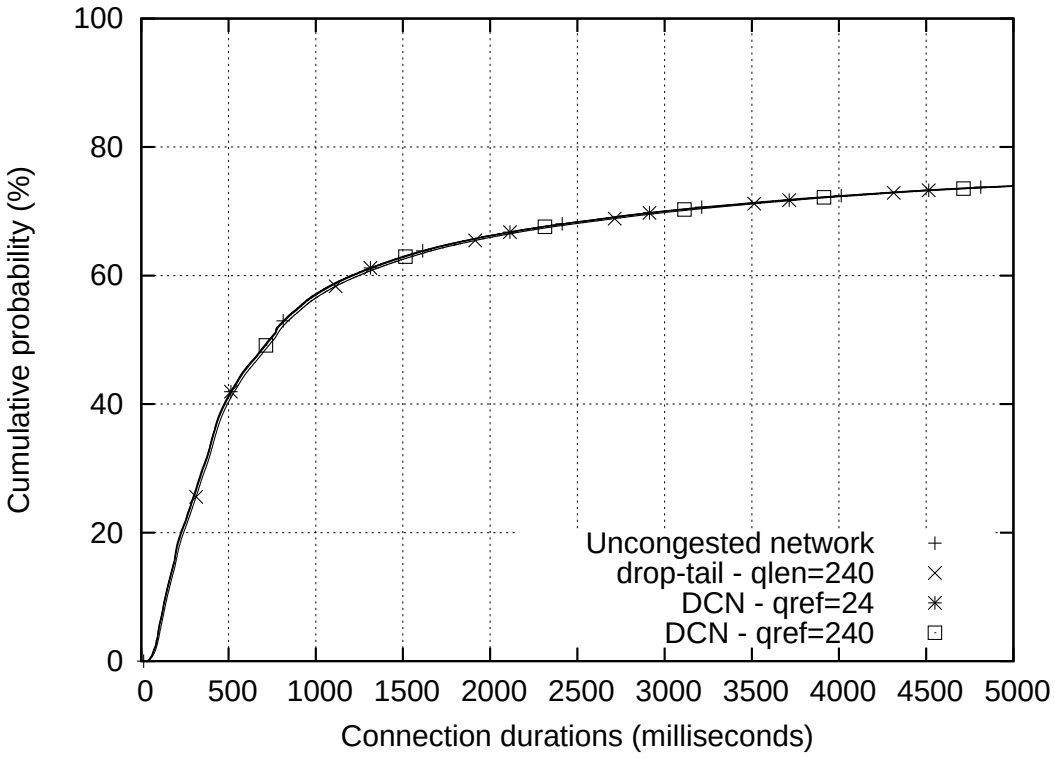

Figure 7.38: DCN performance at 80% load

packets.

As the offered load increased to 95%, DCN continued to give the same performance as the uncongested network with both queue references. The performance of DCN was considerable better than that of drop-tail at this load. The performance for DCN once again demonstrated the advantage of differential treatment of flows in improving application performance.

7.3 Results for ARED, PI, LQD, and REM with ECN

Experimental results for various AQM algorithms presented in section 7.2 were obtained when they were used with packet drops. In order to quantify the effects of the ECN signaling protocol, experiments from section 7.2 were repeated but the ECN protocol was now used and routers were allowed to mark instead of dropping packets.

7.3.1 Results for PI/ECN

Figures 7.44, 7.45, and 7.46 show experimental results for PI with and without ECN at 80%, 90%, and 95% offered loads. These results were obtained with general TCP applications when PI was used without link-level buffering and with a queue reference of 24 and

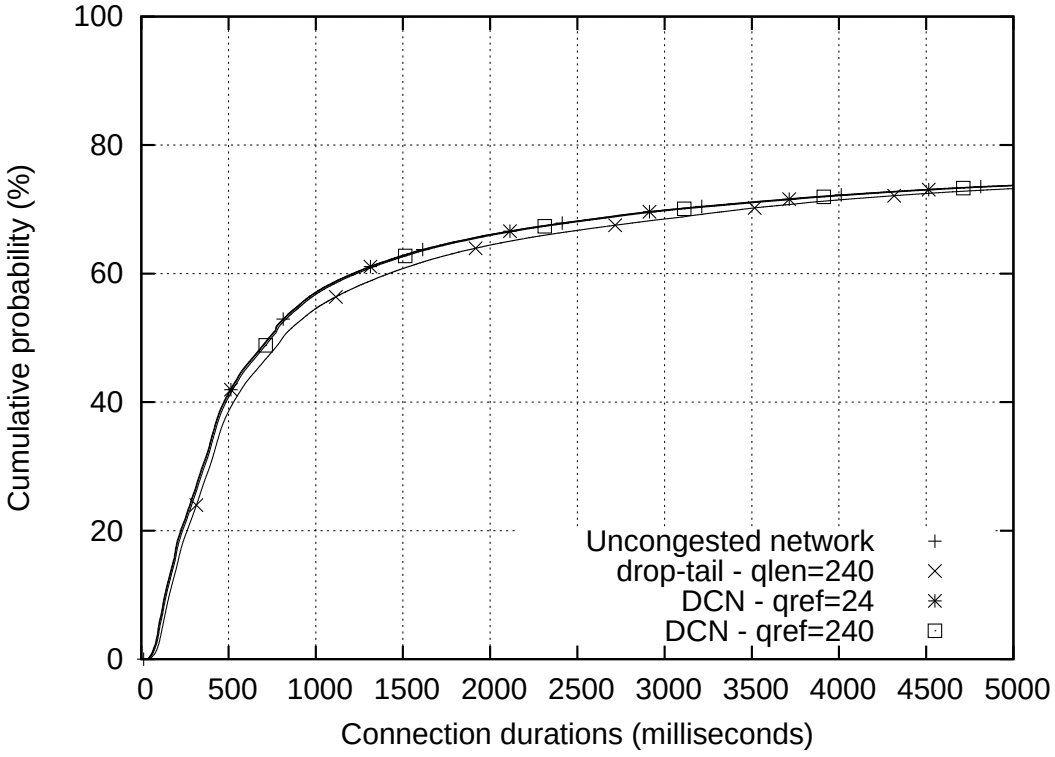

Figure 7.39: DCN performance at 90% load

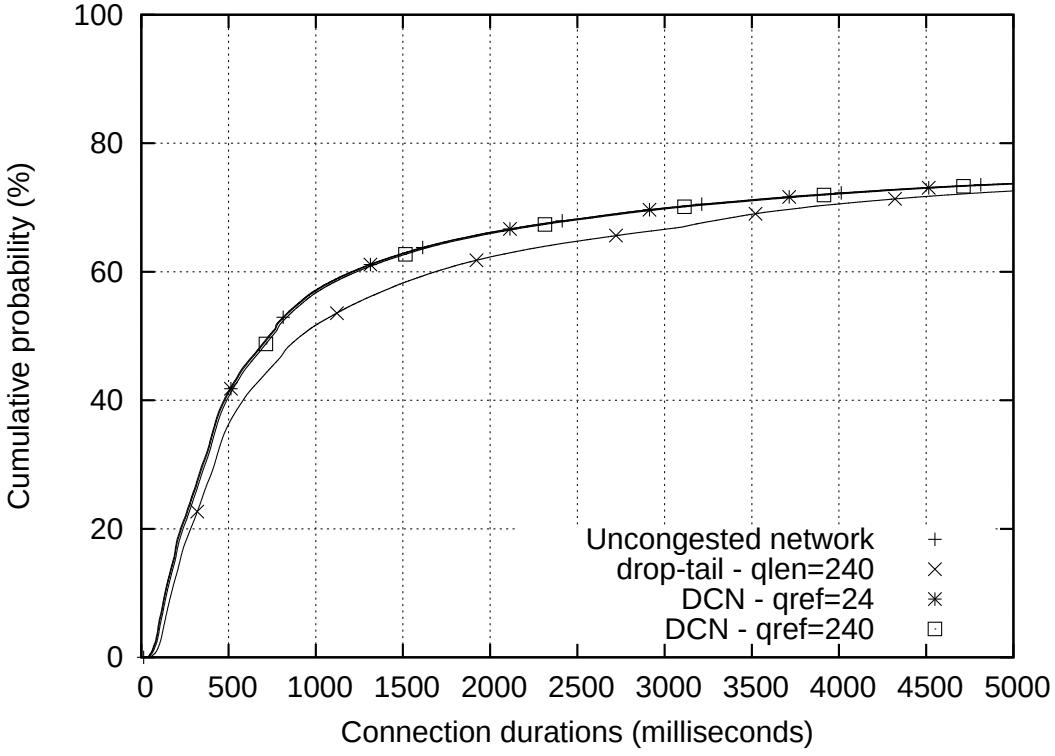

Figure 7.40: DCN performance at 95% load

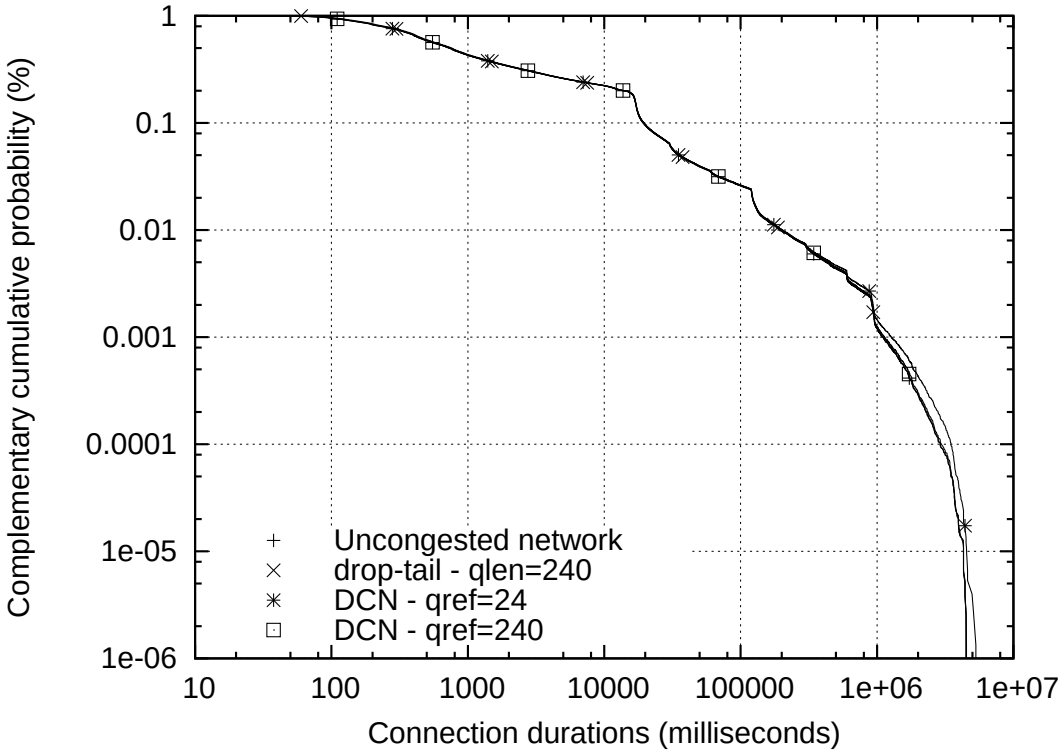

Figure 7.41: DCN performance at 80% load (CCDF)

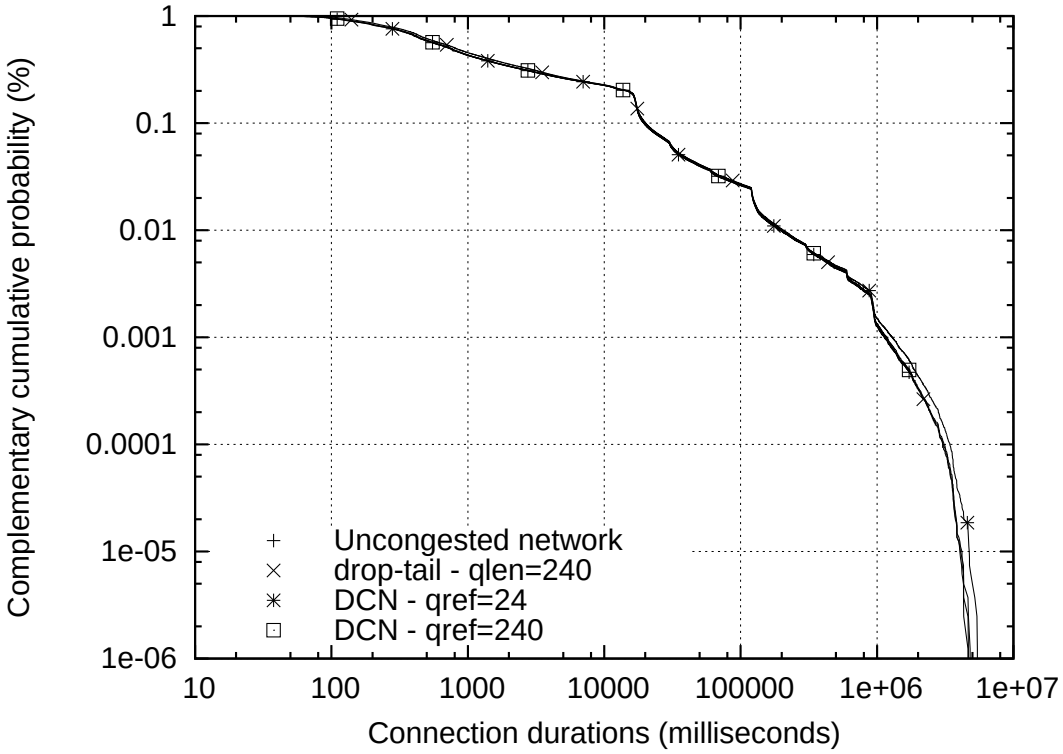

Figure 7.42: DCN performance at 90% load (CCDF)

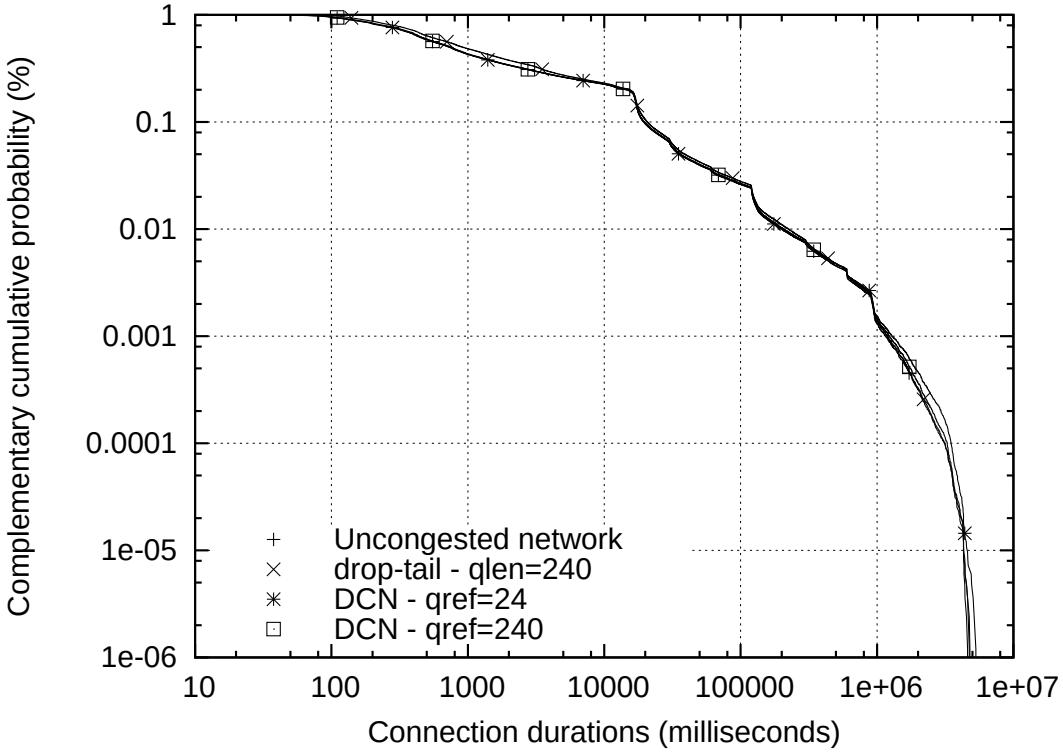

Figure 7.43: DCN performance at 95% load (CCDF)

240 packets.

At 80% load, PI did not gain any performance improvement with ECN over packet drops. With or without ECN, PI with both queue references obtained the same performance the uncongested network and drop-tail with a queue reference of 240 packets.

At 90% offered load, ECN did not affect the performance for PI when PI was used with a queue reference of 240 packets. With or without ECN, PI delivered the same performance when it was used with a queue reference of 240 packets. The performance for PI with a queue reference of 240 packets (with or without ECN) was comparable to that of drop-tail and closely approximated the performance of the uncongested network. When PI was used with a queue reference of 24 packets, the addition of ECN sightly decreased the performance of PI for approximately 60% flows that completed within 1,500 milliseconds.

As the offered load increased to 95%, ECN improved the performance for PI slightly when PI was used with a queue referece of 24 packets. With or without ECN, PI with a queue reference of 24 packets gave slightly better performance than drop-tail with a queue length of 240 packets. It is interesting that PI/ECN with a queue reference of 24 packets was the worst performing combination for PI at 90% load but gave the best performance for PI at 95% load. When PI was used with a queue reference of 240 packets, ECN neither increased or decreased the performance for PI. With or without ECN, PI with a queue reference of

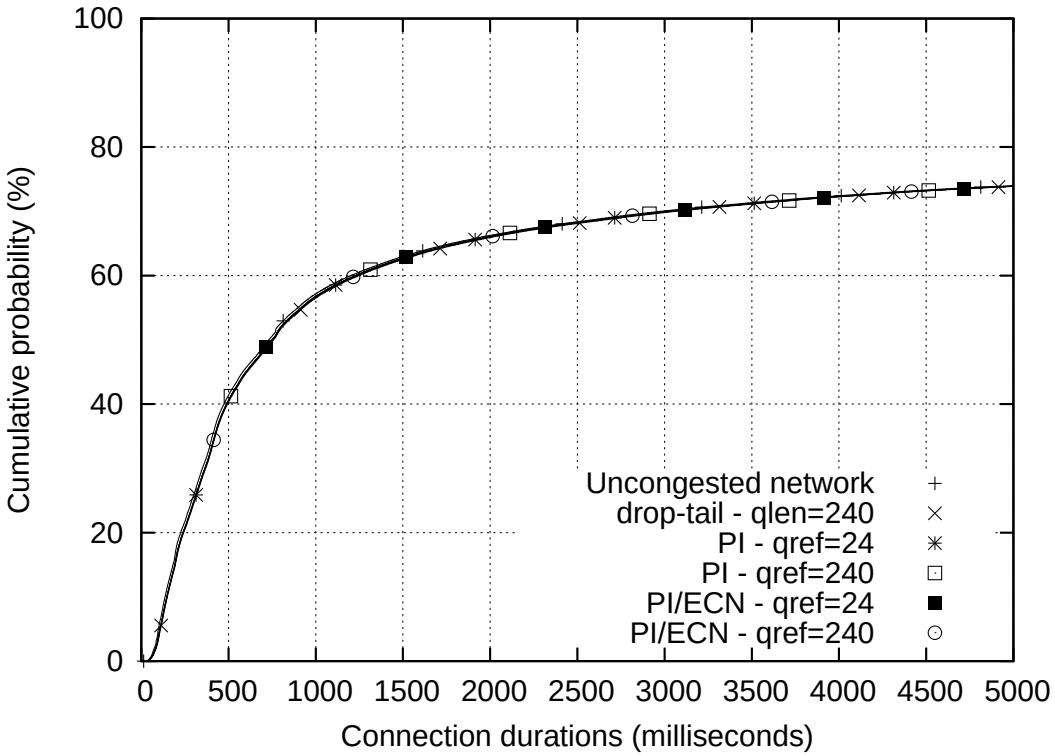

Figure 7.44: PI/ECN performance at 80% load

240 packets underperformed drop-tail for approximately 50% of flows that completed within 1 second but gave better performance than drop-tail for the other 50% of flows.

7.3.2 Results for REM/ECN

Figures 7.50, 7.51, and 7.52 show experimental results for REM with and without ECN at 80%, 90%, and 95% loads. These results were obtained with general TCP applications and without link-level buffering.

At 80% offered load, REM delivered the same performance with both queue references when it was used with and without ECN. The performance for REM at this load was identical to that of drop-tail and of the uncongested network.

At 90% offered load, the addition of ECN did not change the performance for REM with both queue references. With or without ECN, REM with both queue references gave about the same performance as drop-tail with a queue reference of 24 packets and closely approximated the performance of the uncongested network.

As the offered load increased to 95%, the addition of ECN marking protocol improved the performance for REM with both queue references slightly. With or without ECN, the performance for REM came relatively close to that of the uncongested network.

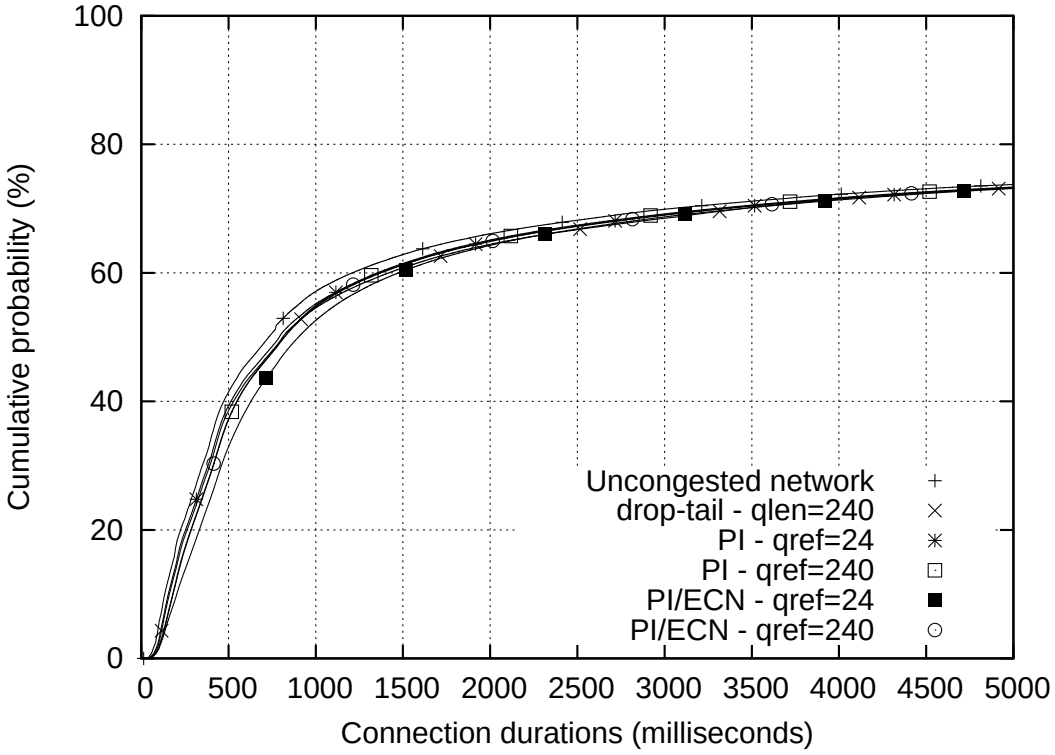

Figure 7.45: PI/ECN performance at 90% load

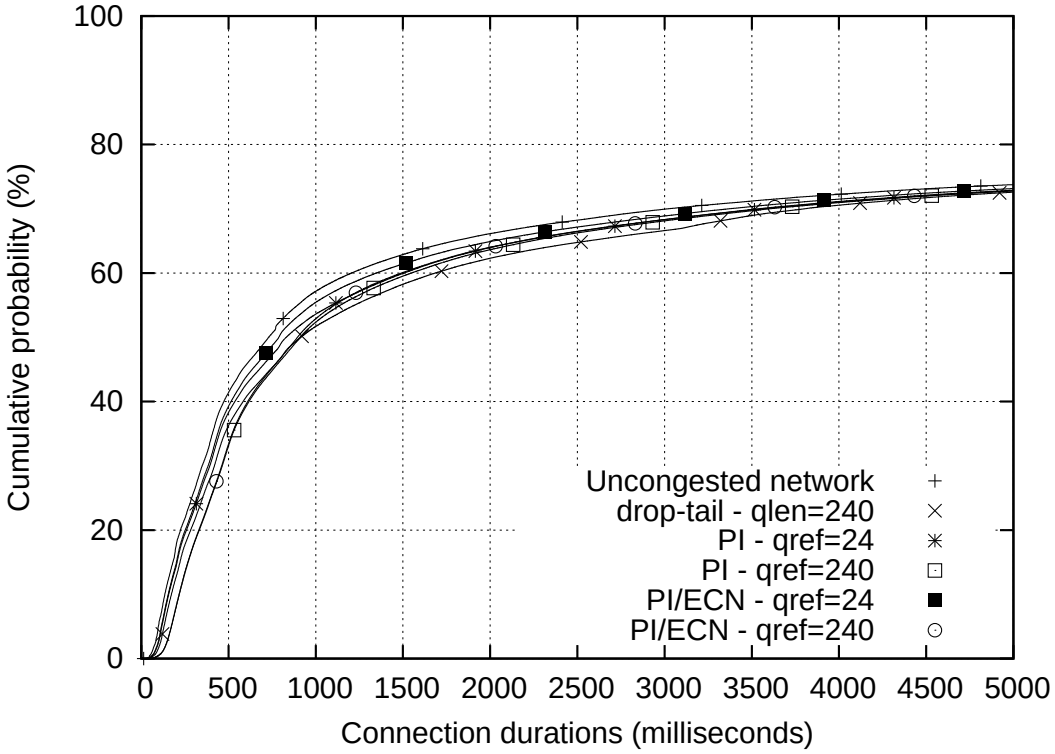

Figure 7.46: PI/ECN performance at 95% load

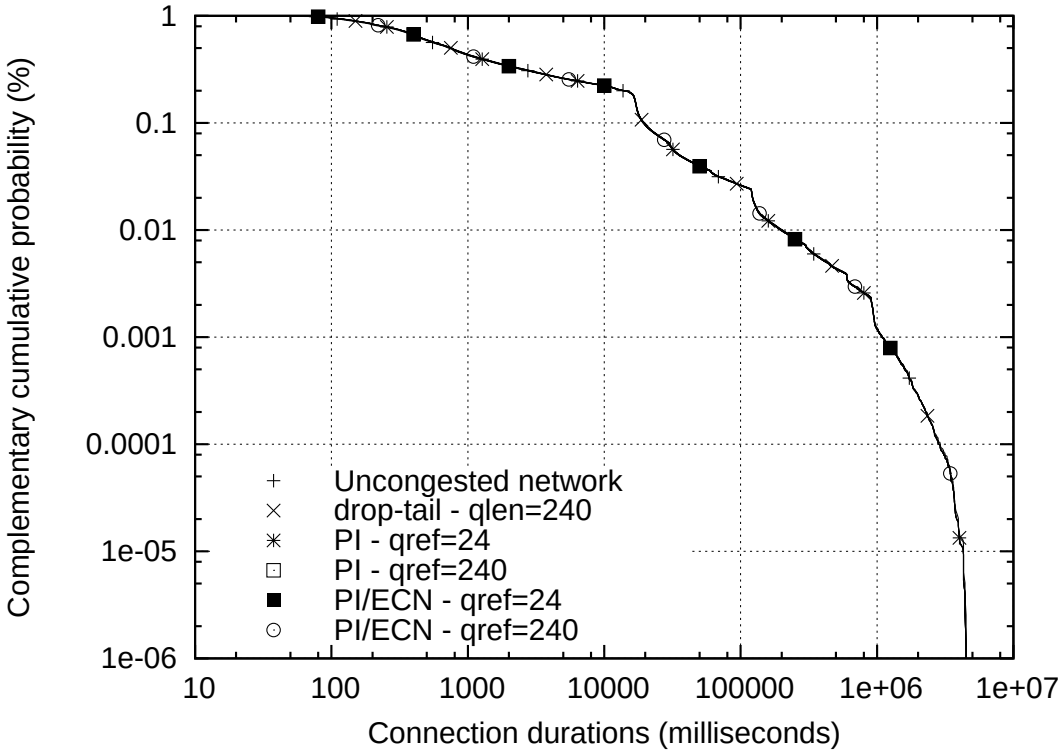

Figure 7.47: PI/ECN performance at 80% load (CCDF)

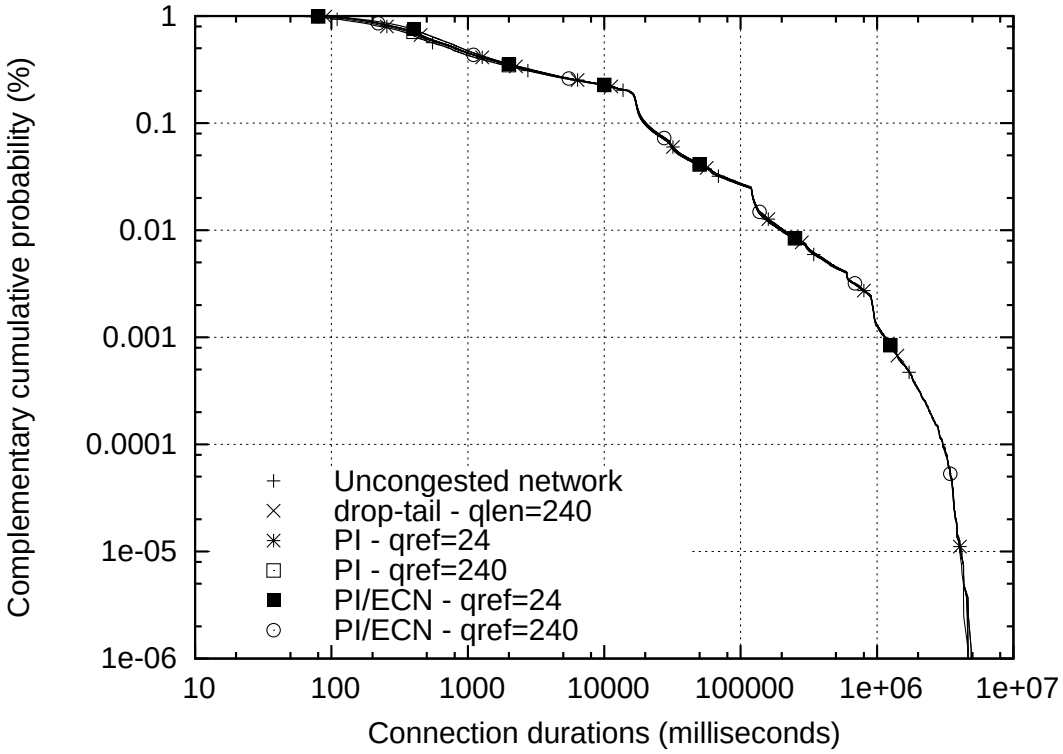

Figure 7.48: PI/ECN performance at 90% load (CCDF)

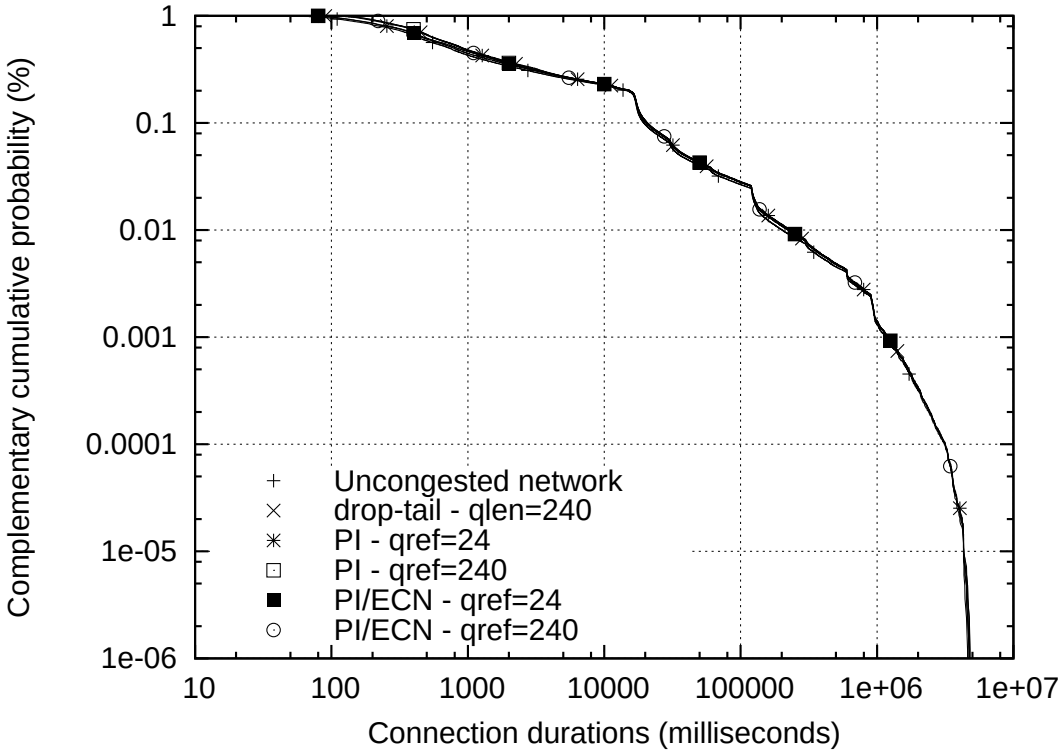

Figure 7.49: PI/ECN performance at 95% load (CCDF)

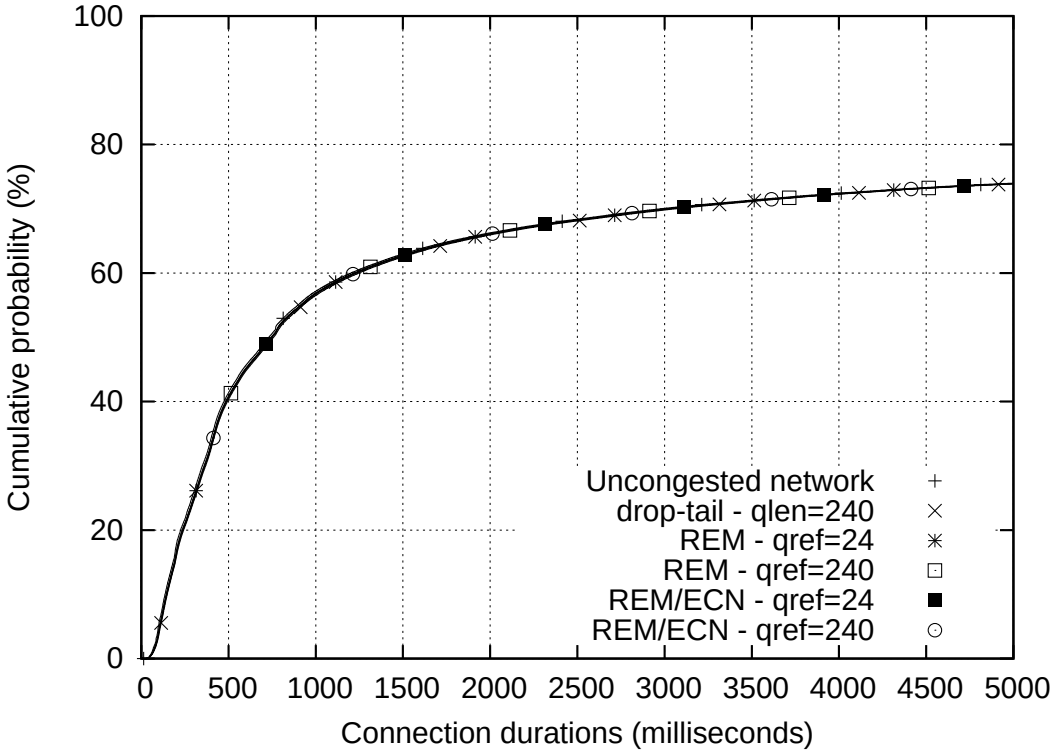

Figure 7.50: REM/ECN performance at 80% load

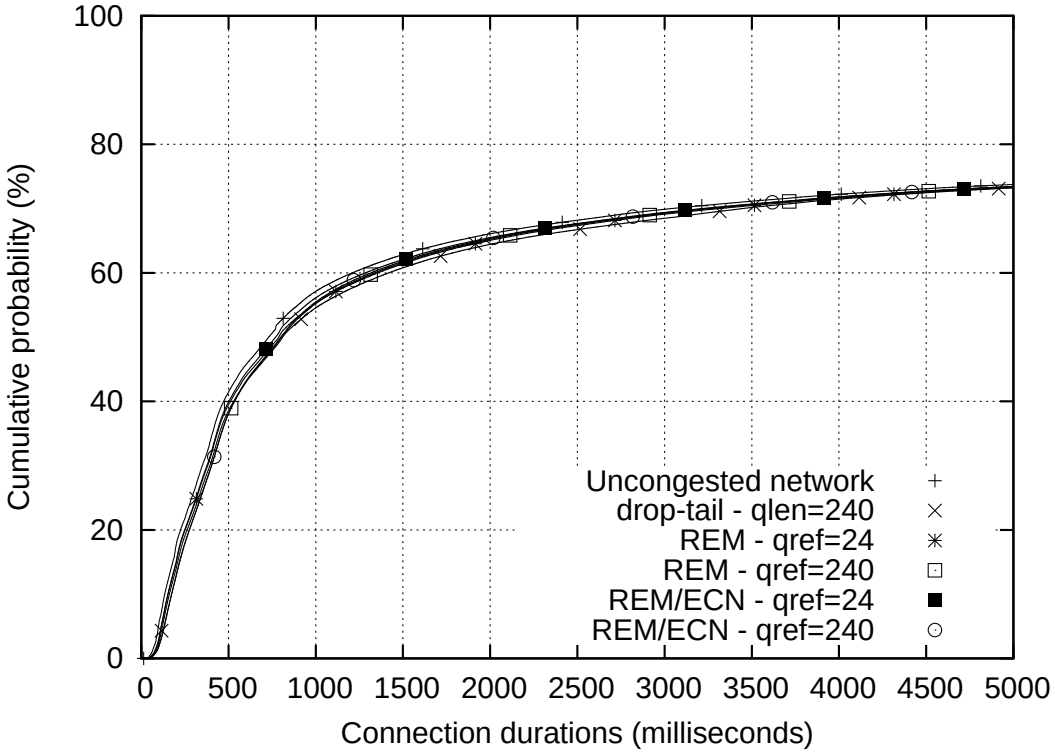

Figure 7.51: REM/ECN performance at 90% load

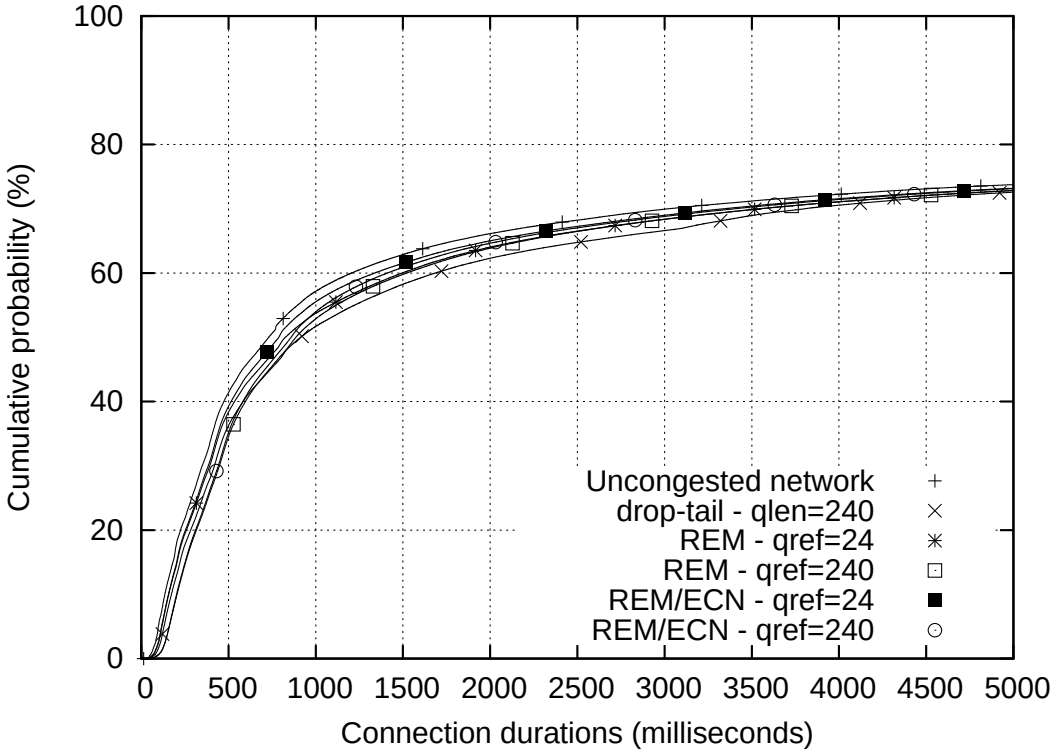

Figure 7.52: REM/ECN performance at 95% load

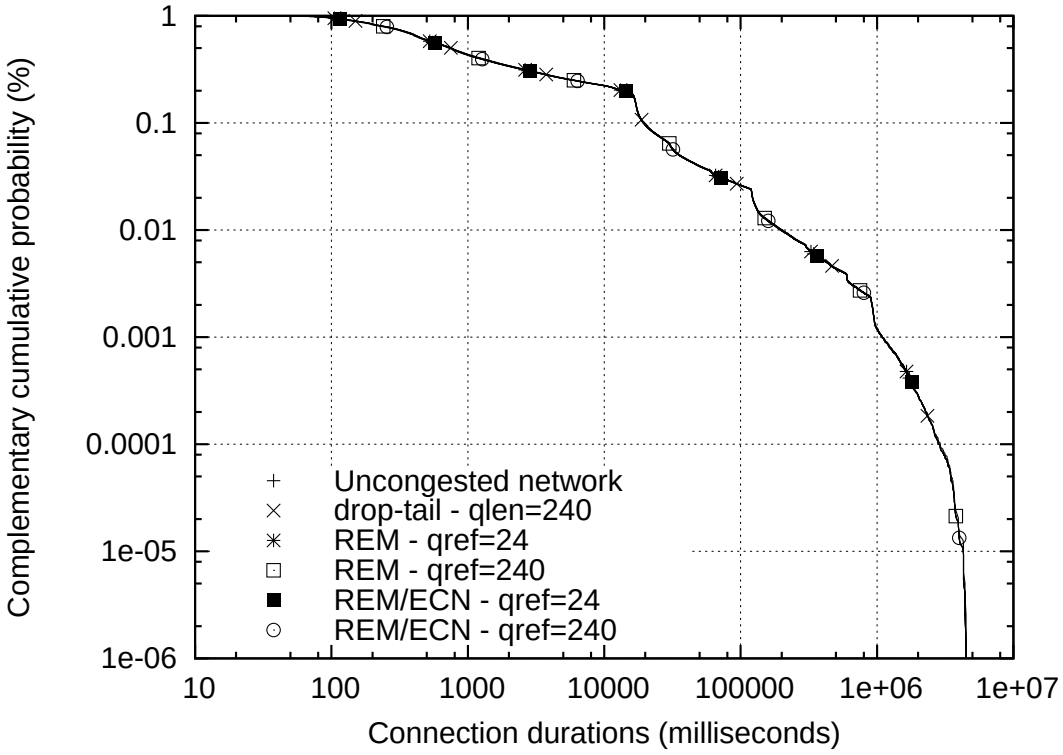

Figure 7.53: REM/ECN performance at 80% load (CCDF)

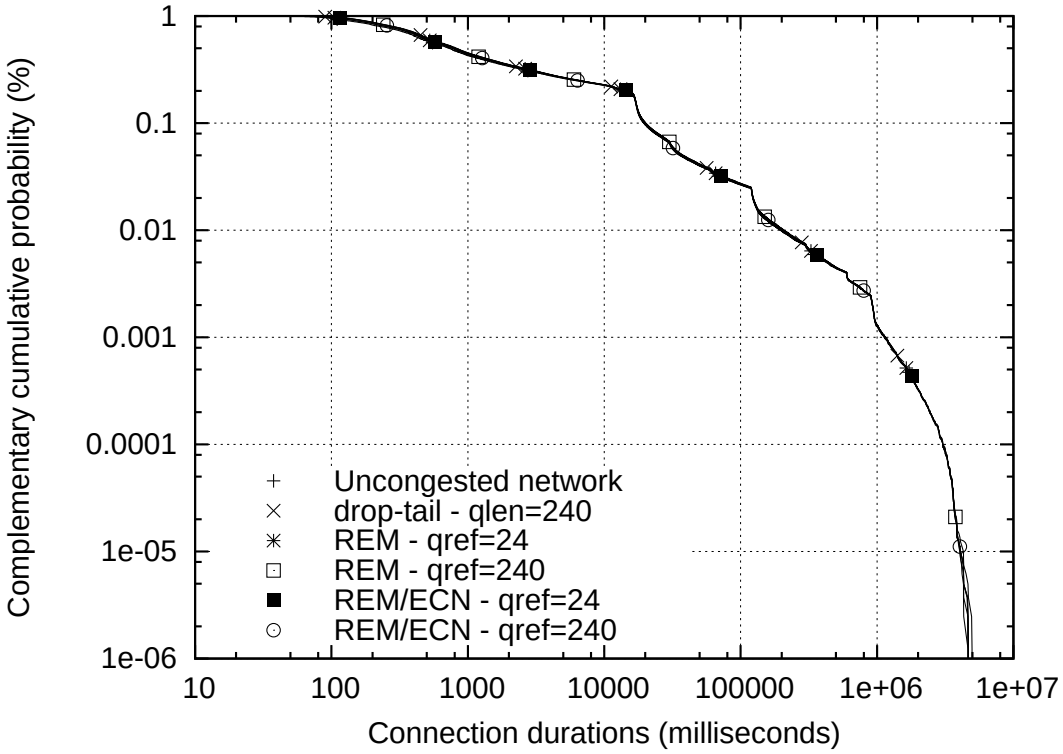

Figure 7.54: REM/ECN performance at 90% load (CCDF)

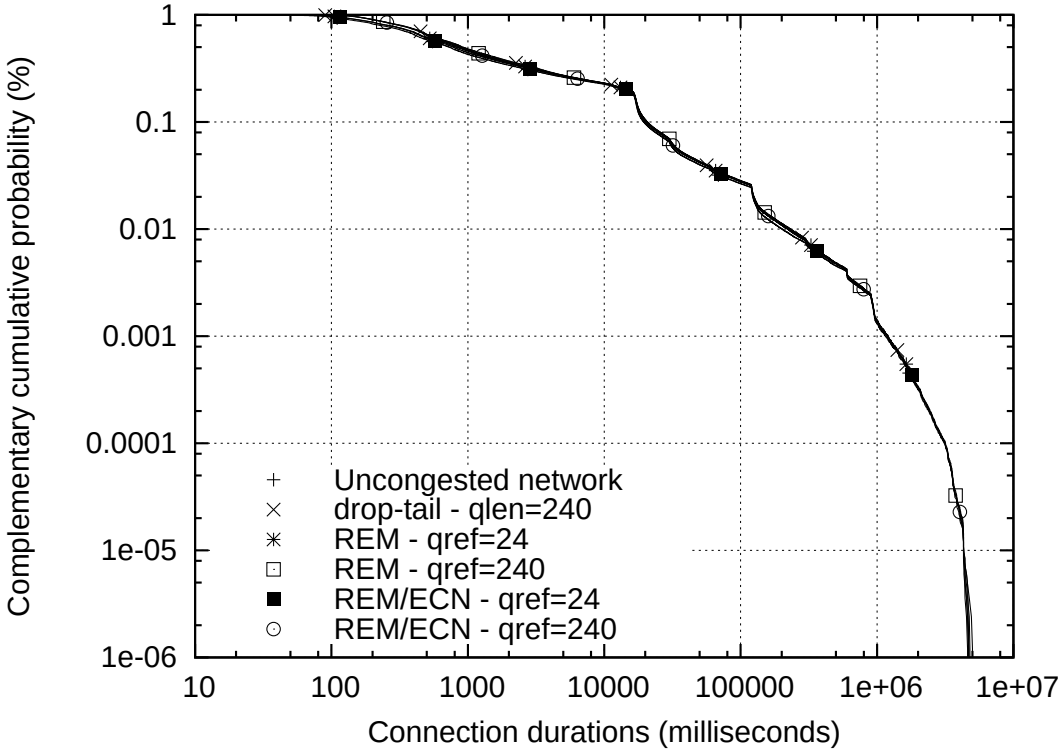

Figure 7.55: REM/ECN performance at 95% load (CCDF)

7.3.3 Results for ARED/ECN

Figures 7.56, 7.57, and 7.58 show experimental results for ARED with general TCP applications when link-level buffering was eliminated. These results were obtained with and without ECN when ARED operated with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets). Further, figures 7.62, 7.63, and 7.64 show experimental results for ARED when it was used with ECN in the "new gentle" mode (ARED/ECN "new gentle") to demonstrate the effects of dropping packets in ECN mode of ARED on general TCP applications.

At 80% offered load, the addition of ECN did not improve the performance for ARED. This was because ARED without ECN already gave performance that was undistinguishable from the performance of the uncongested network. With or without ECN, ARED approximated the performance of the uncongested network with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets).

At 90% offered load, ARED also did not benefit from the addition of the ECN signaling protocol. When used with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets), ARED obtained the same performance with and without ECN. Further, the performance for ARED with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) was considerably

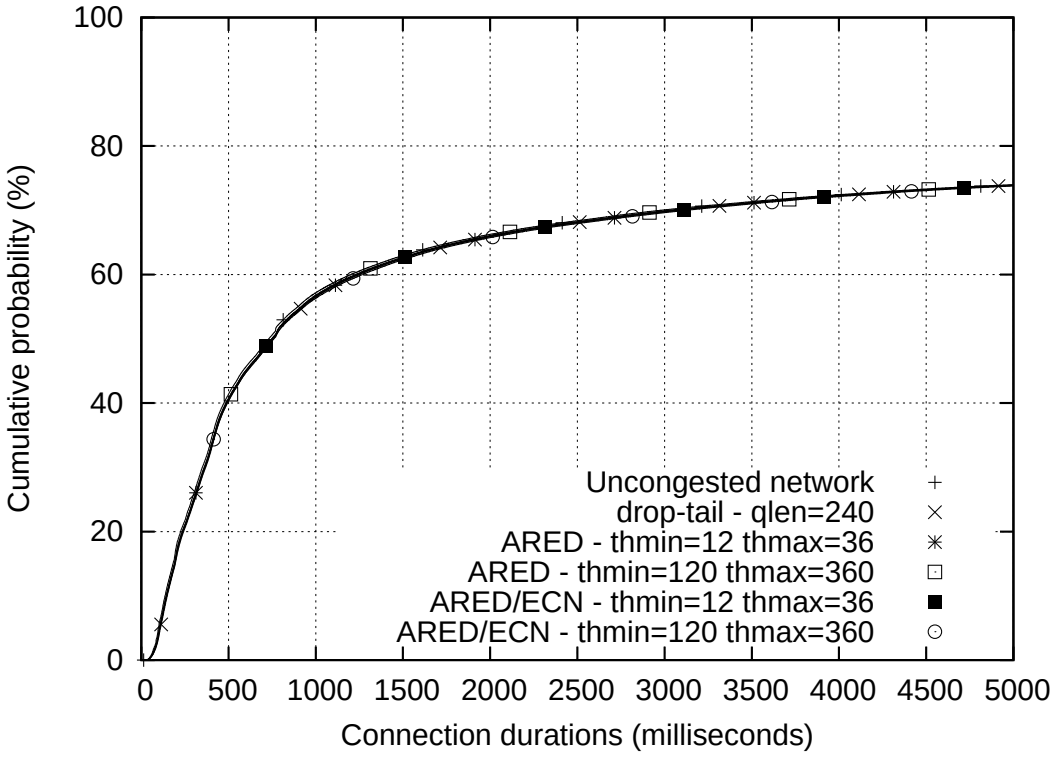

Figure 7.56: ARED/ECN performance at 80% load

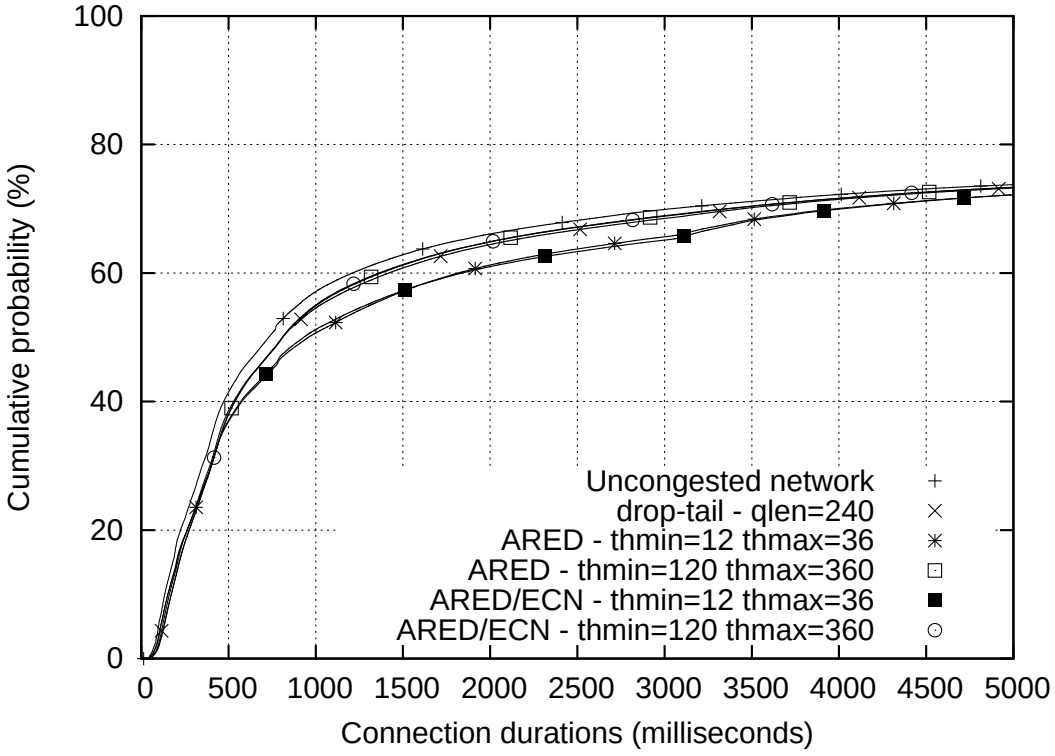

Figure 7.57: ARED/ECN performance at 90% load

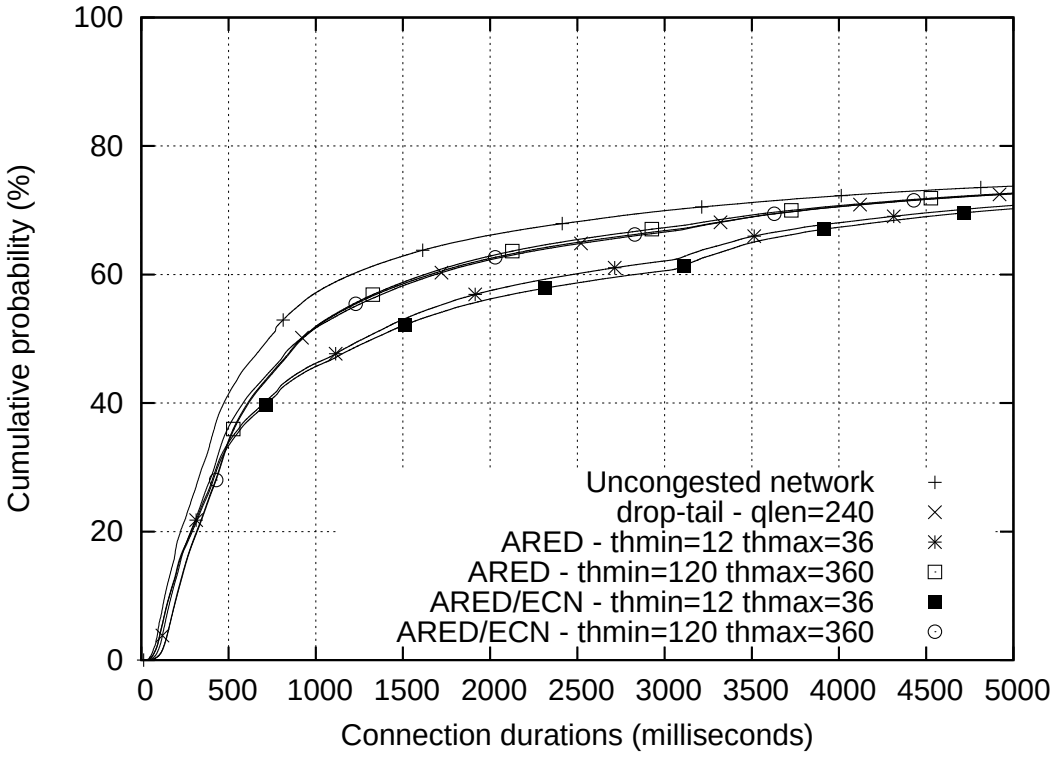

Figure 7.58: ARED/ECN performance at 95% load

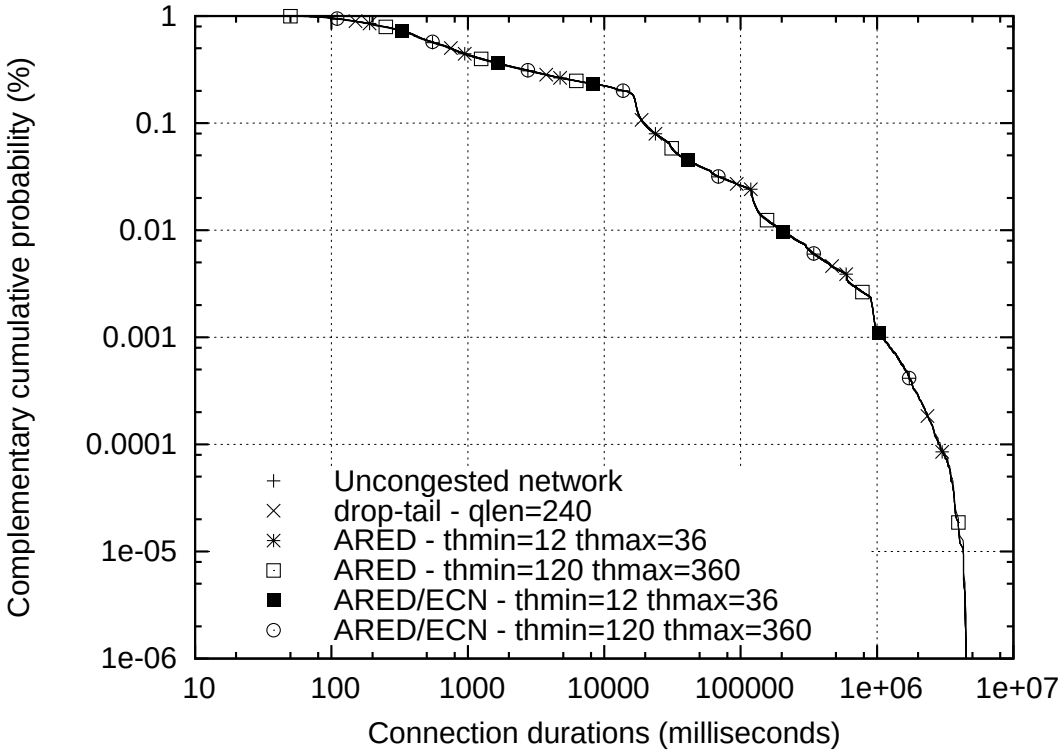

Figure 7.59: ARED/ECN performance at 80% load (CCDF)

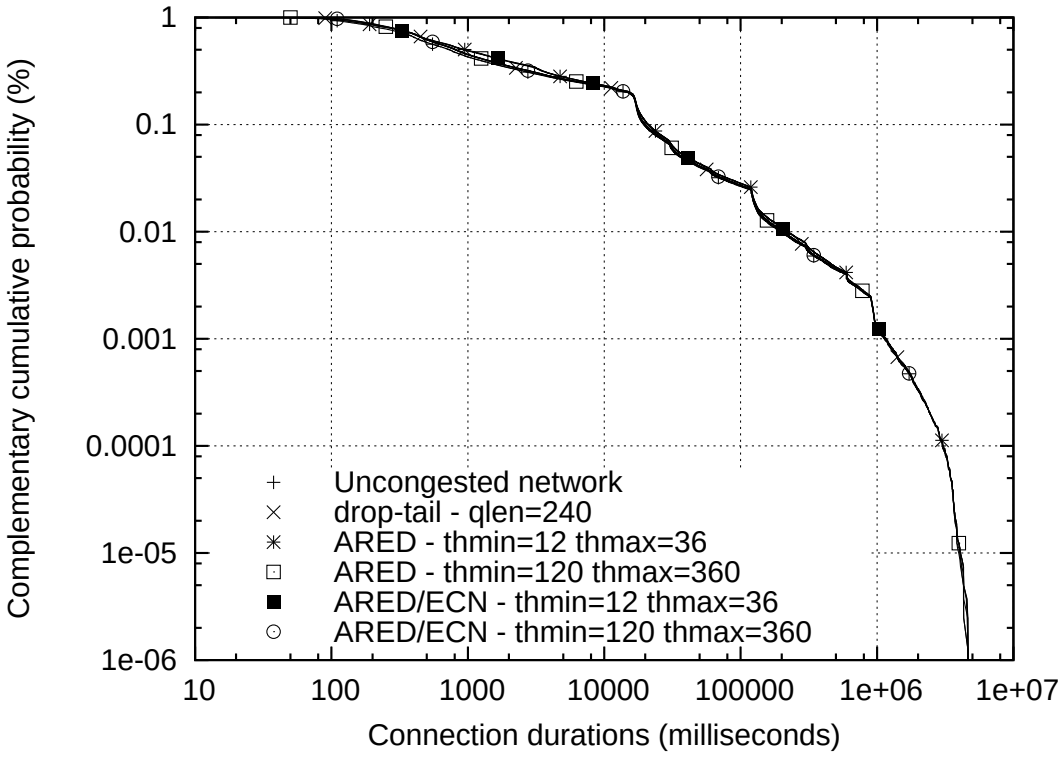

Figure 7.60: ARED/ECN performance at 90% load (CCDF)

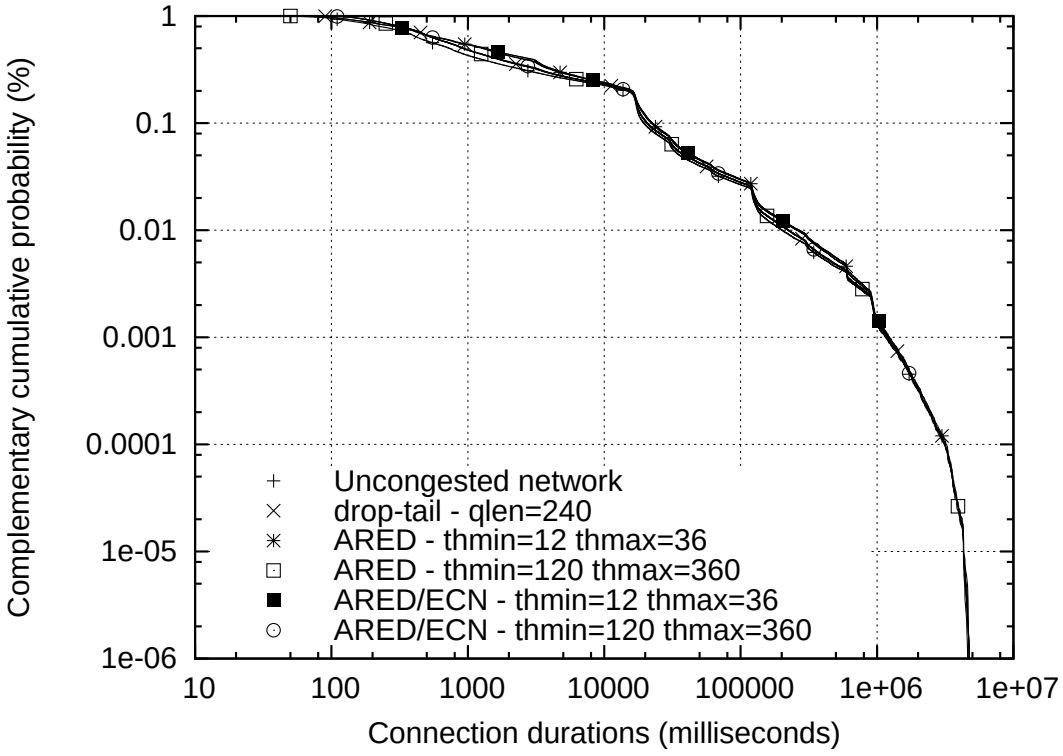

Figure 7.61: ARED/ECN performance at 95% load (CCDF)

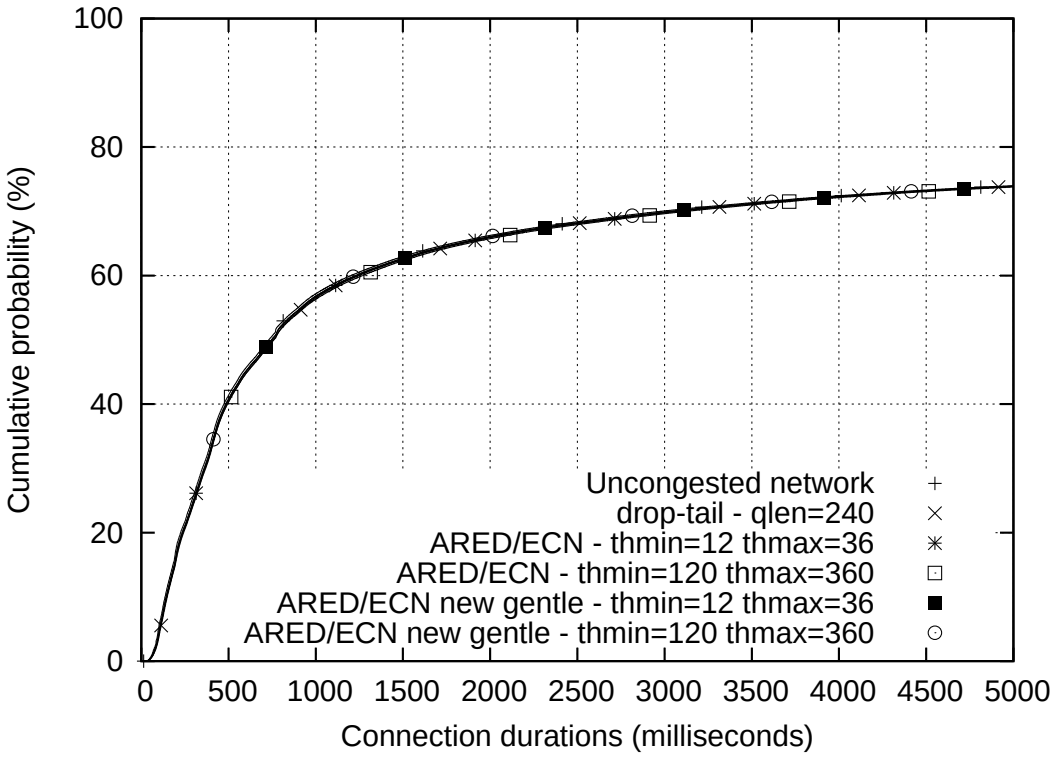

Figure 7.62: ARED/ECN new gentle performance at 80% load

lower than that of drop-tail and of the uncongested network. When ARED was used with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets), it delivered comparable performance with drop-tail and came close to the performance of the uncongested network. With or without ECN, ARED gave the same performance with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets).

At 95% offered load, the addition of ECN slightly degraded the performance for ARED with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets). With or without ECN, ARED with parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) underperformed drop-tail at this load. When ARED was used with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets), it obtained the same performance with and without ECN. The performance for ARED with parameter settings (th_{min} = 120 packets, th_{max} = 360 packets) was comparable to that of drop-tail but considerably lower than the performance of the uncongested network.

When ARED was used with ECN in "new gentle" mode, it delivered the same performance as the original ARED/ECN algorihtm with both parameter settings (th_{min} = 12 packets, th_{max} = 36 packets) and (th_{min} = 120 packets, th_{max} = 360 packets) at 80% offered load. The performance for ARED/ECN and ARED/ECN "new gentle" was undistinguishable from that of drop-tail and of the uncongested network at this load.

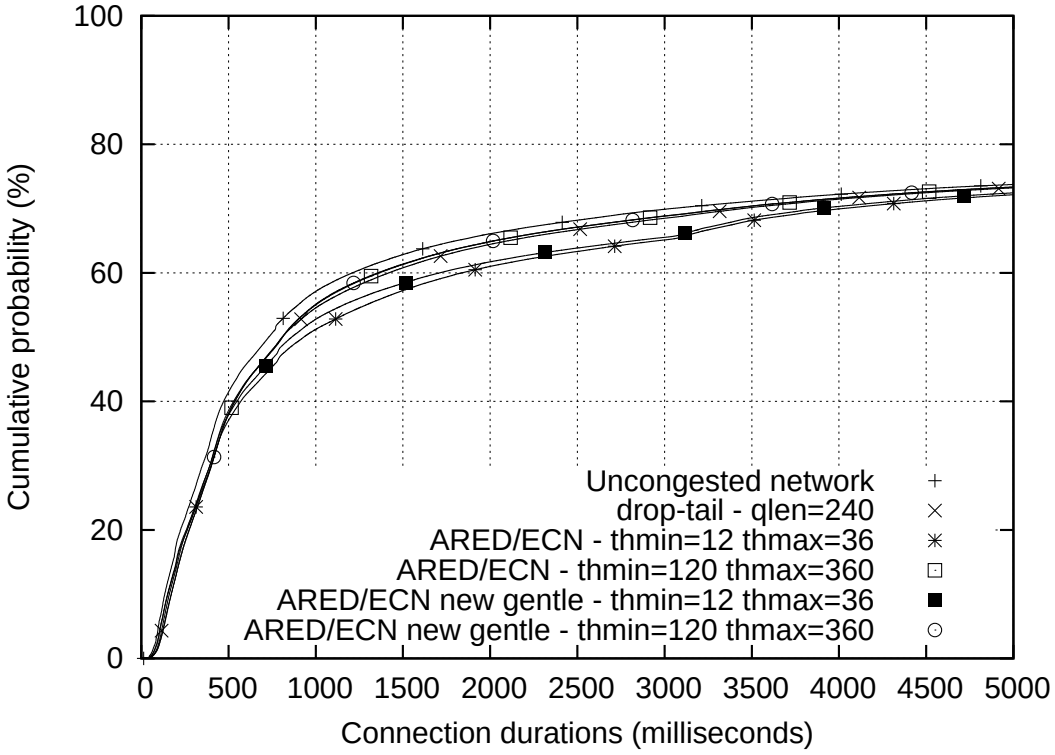

Figure 7.63: ARED/ECN new gentle performance at 90% load

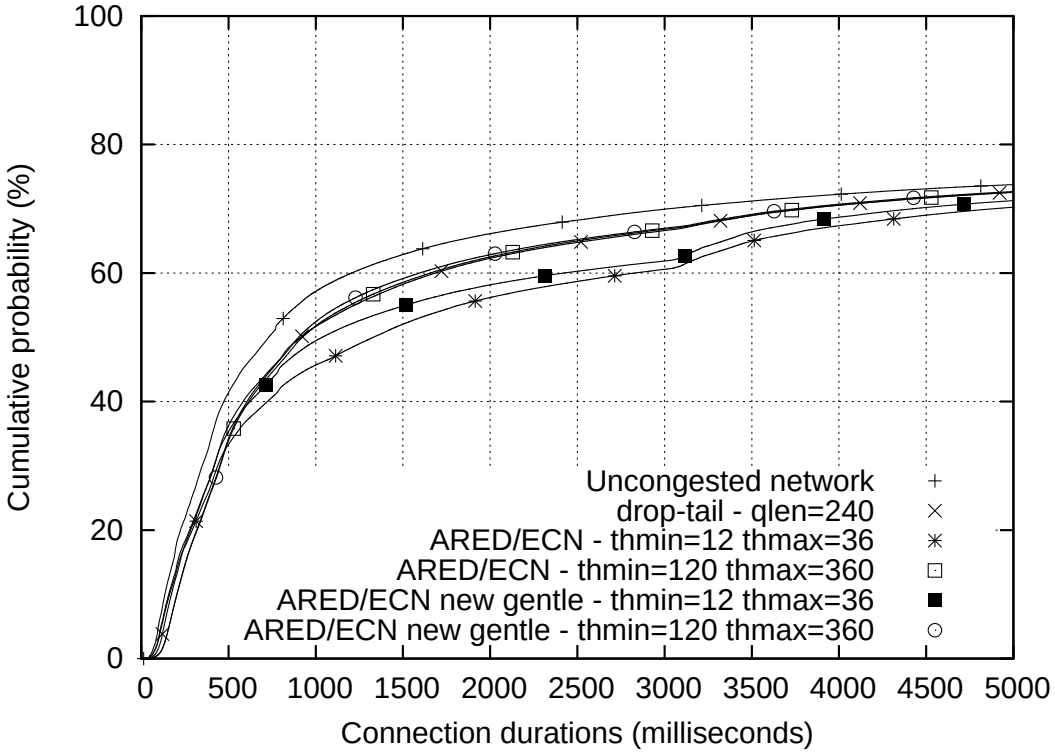

Figure 7.64: ARED/ECN new gentle performance at 95% load

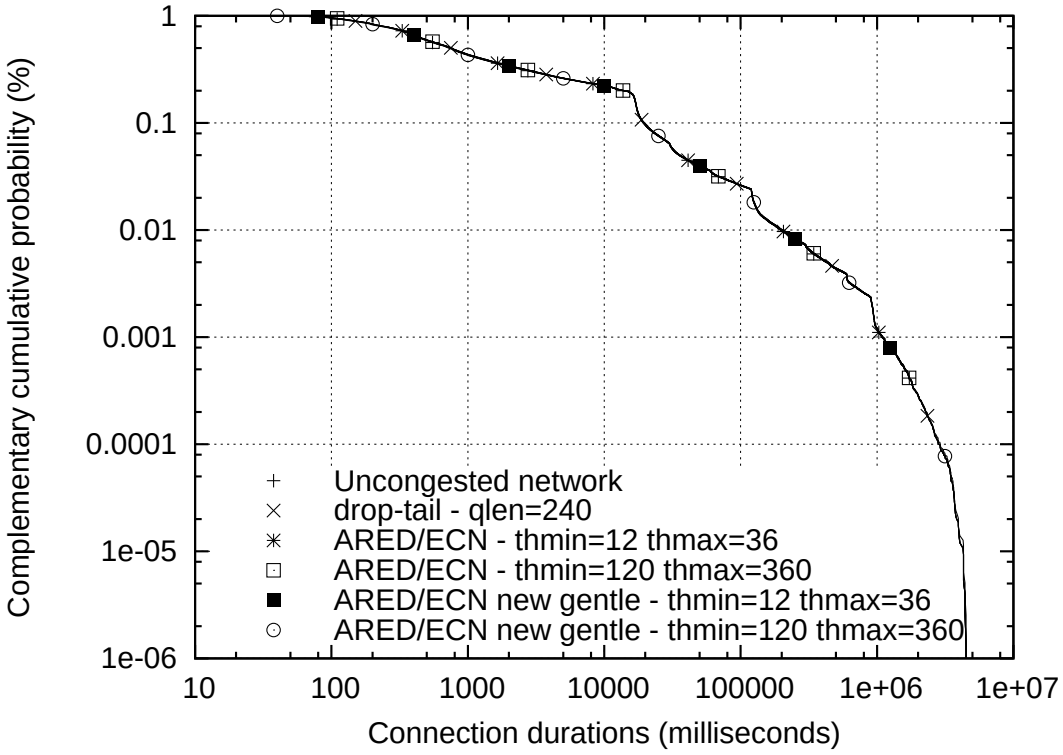

Figure 7.65: ARED/ECN new gentle performance at 80% load (CCDF)

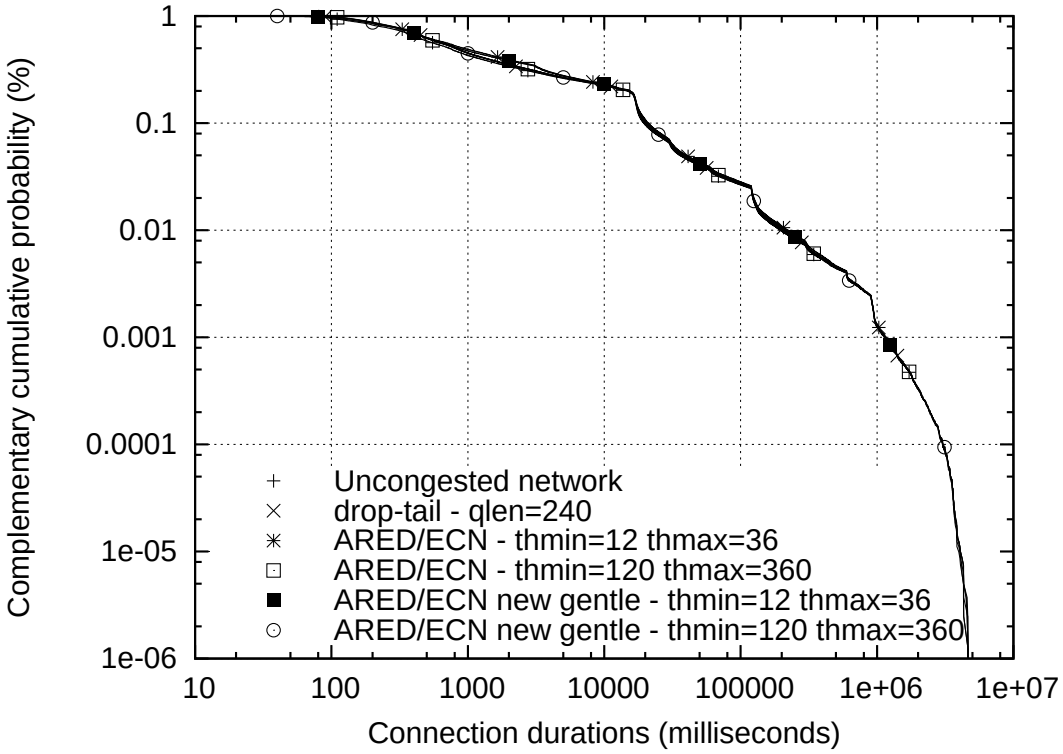

Figure 7.66: ARED/ECN new gentle performance at 90% load (CCDF)

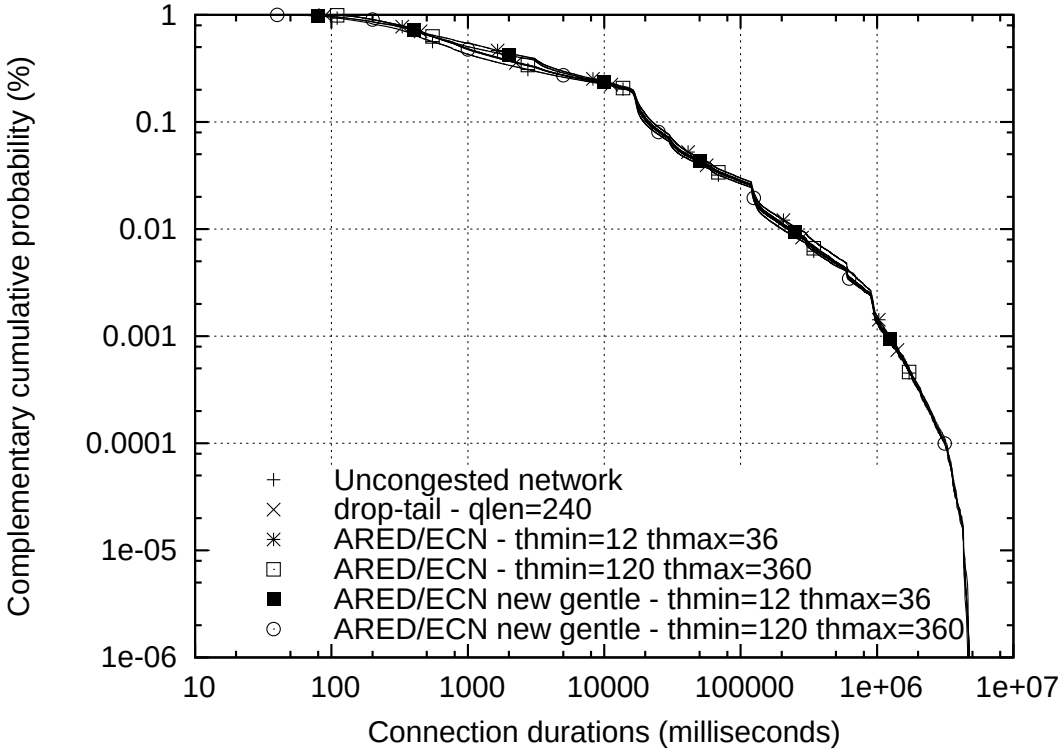

Figure 7.67: ARED/ECN new gentle performance at 95% load (CCDF)

At 90% offered load, ARED/ECN "new gentle" gave a small performance improvement over ARED/ECN when they were used with parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets). However, the performance for ARED/ECN "new gentle" with the parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets) was still lower than the performance of drop-tail with a queue length of 240 packets. When ARED was used with parameter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets), ARED/ECN and ARED/ECN "new gentle" gave essentially the same performance. The performance for ARED/ECN and ARED/ECN "new gentle" with parameter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets) was comparable with that of drop-tail and came close to the performance of the uncongested network.

At 95% offered load, ARED/ECN "new gentle" delivered considerable performance improvement over ARED/ECN when they were used with parameter settings ($th_{min} = 12$ packets, $th_{max} = 36$ packets). Nevertheless, ARED/ECN "new gentle" underperformed drop-tail for approximately 60% of flows that needed more than 600 milliseconds to complete. When ARED was used with parameter settings ($th_{min} = 120$ packets, $th_{max} = 360$ packets), both ARED/ECN and ARED/ECN "new gentle" obtained the same performance as drop-tail with a queue length of 240 packets.

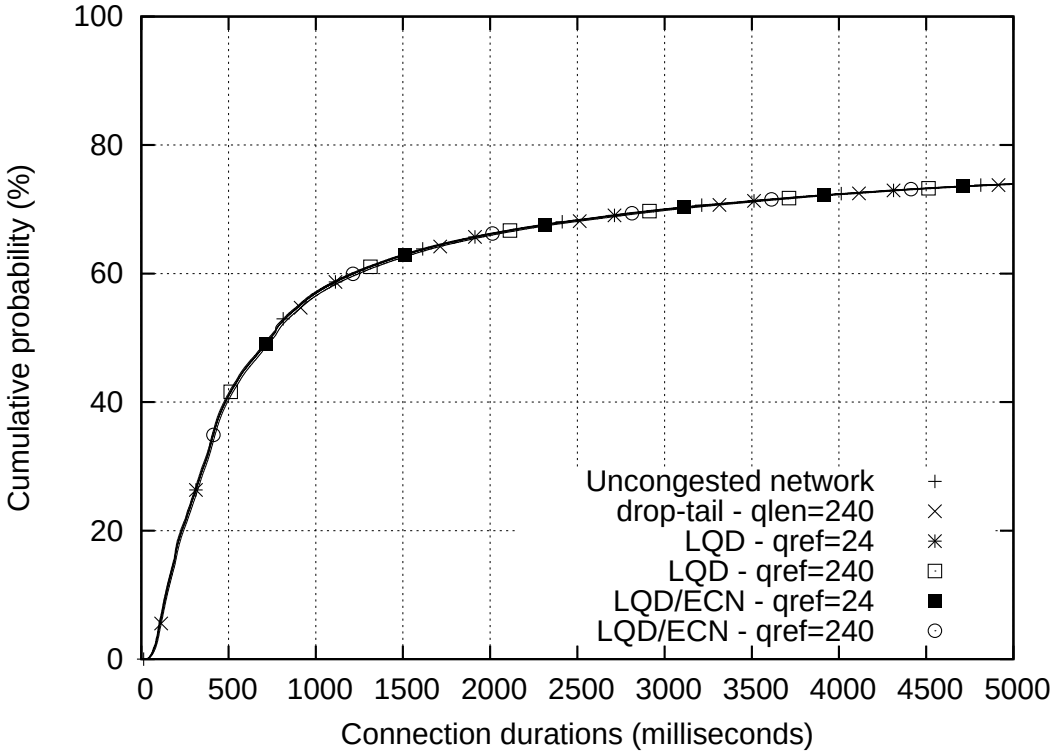

Figure 7.68: LQD/ECN performance at 80% load

7.3.4 Results for LQD/ECN

Figures 7.68, 7.69, and 7.70 show experimental results for LQD with general TCP applications when LQD was used without link-level buffering. These results were obtained for LQD with and without ECN when LQD operated with a queue reference of 24 and 240 packets.

At 80% offered load, LQD with both queue references of 24 and 240 packets delivered the same performance when it was used with and without ECN. The performance for LQD was undistinguishable from that of drop-tail and of the uncongested network at this load.

At 90% offered load, LQD did not benefit from the ECN signaling protocol because it already obtained performance very close to that of the uncongested network. With both queue references of 24 and 240 packets, LQD gave the same performance when it operated with and without ECN. The performance for LQD at this load was slightly better than drop-tail.

At 95% offered load, LQD did not gain any performance improvement with ECN when it was used with both queue references. With or without ECN, the performance for LQD was very close to that of the uncongested network and considerably better than the performance of drop-tail with a queue length of 240 packets at this load.

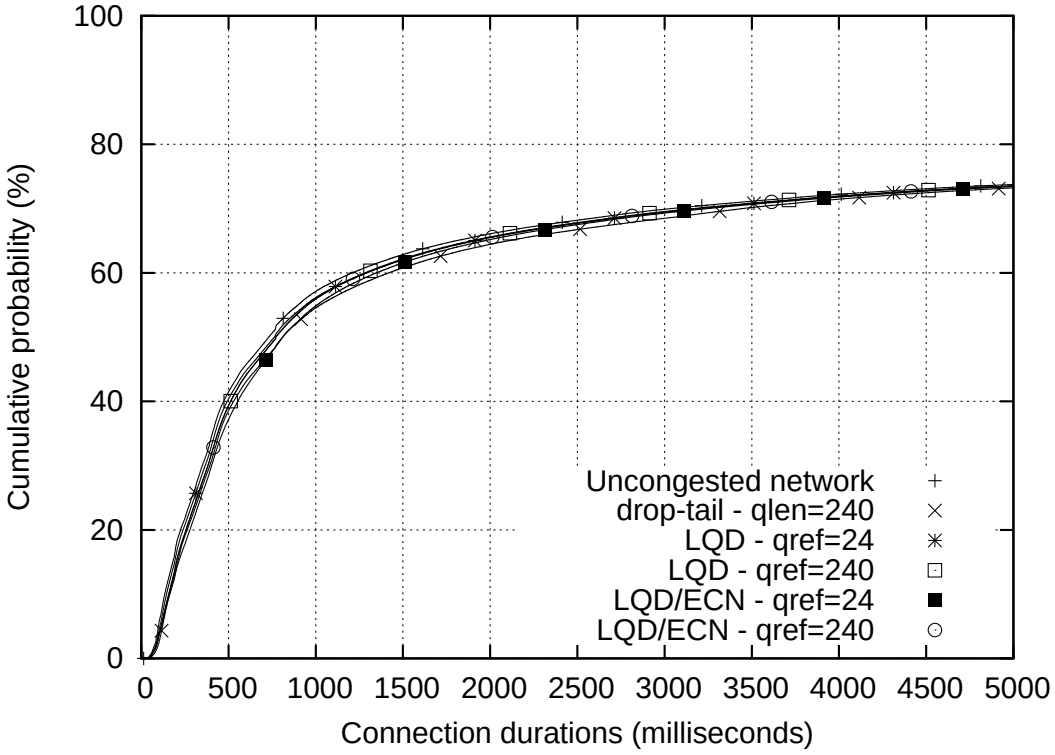

Figure 7.69: LQD/ECN performance at 90% load

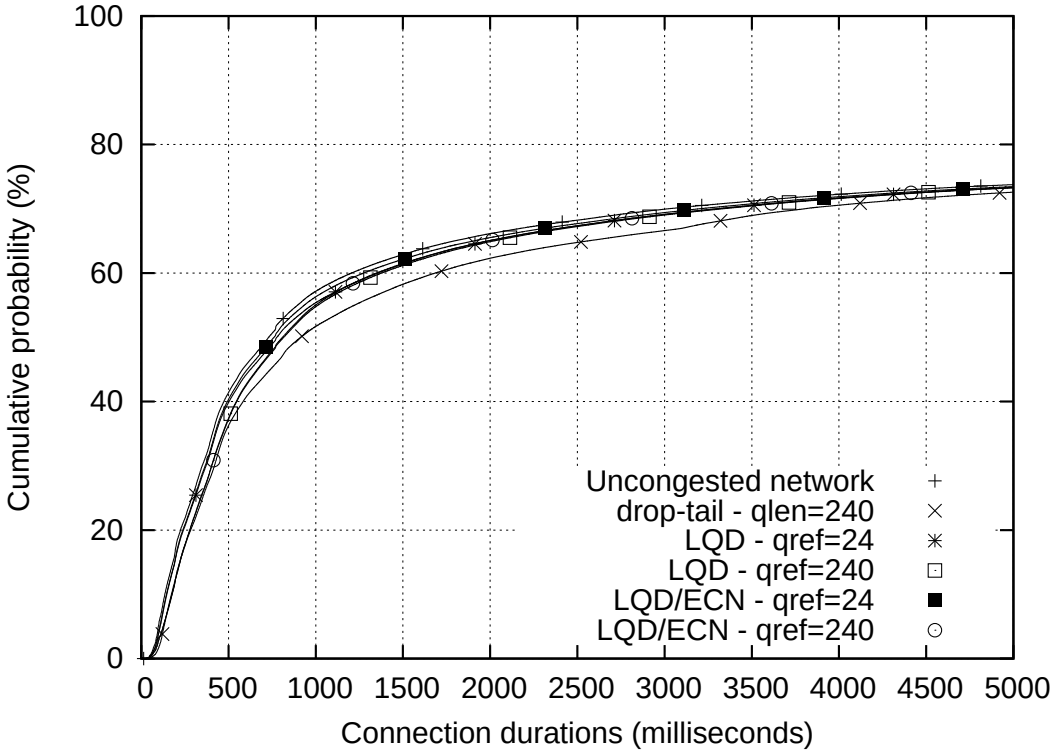

Figure 7.70: LQD/ECN performance at 95% load

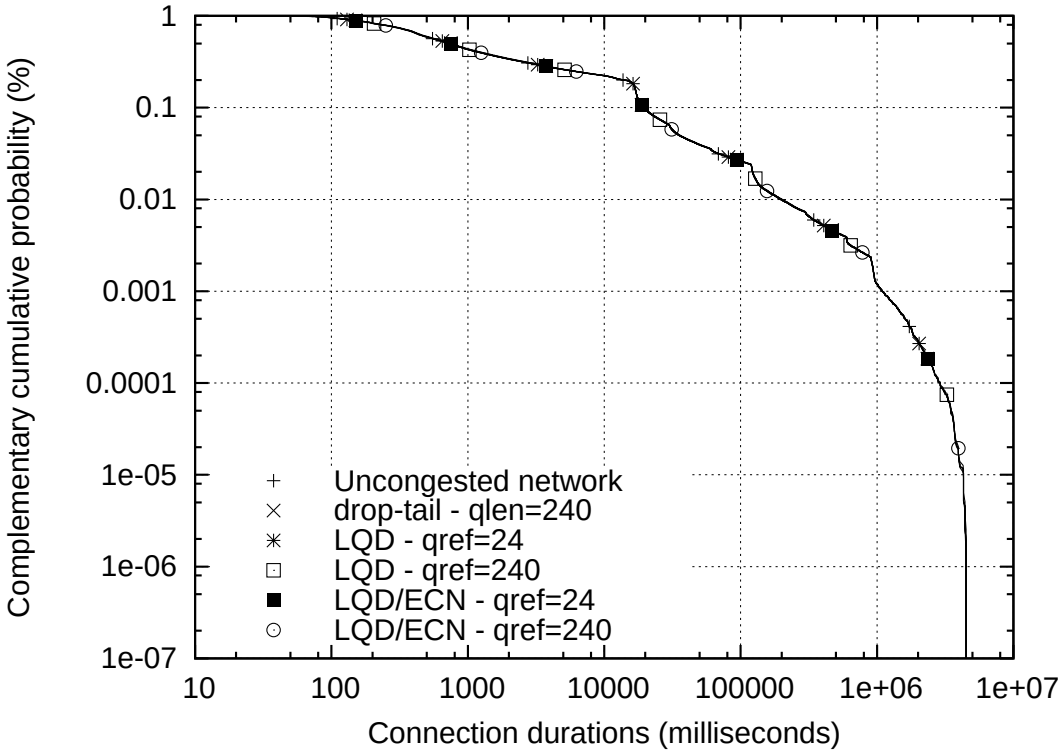

Figure 7.71: LQD/ECN performance at 80% load (CCDF)

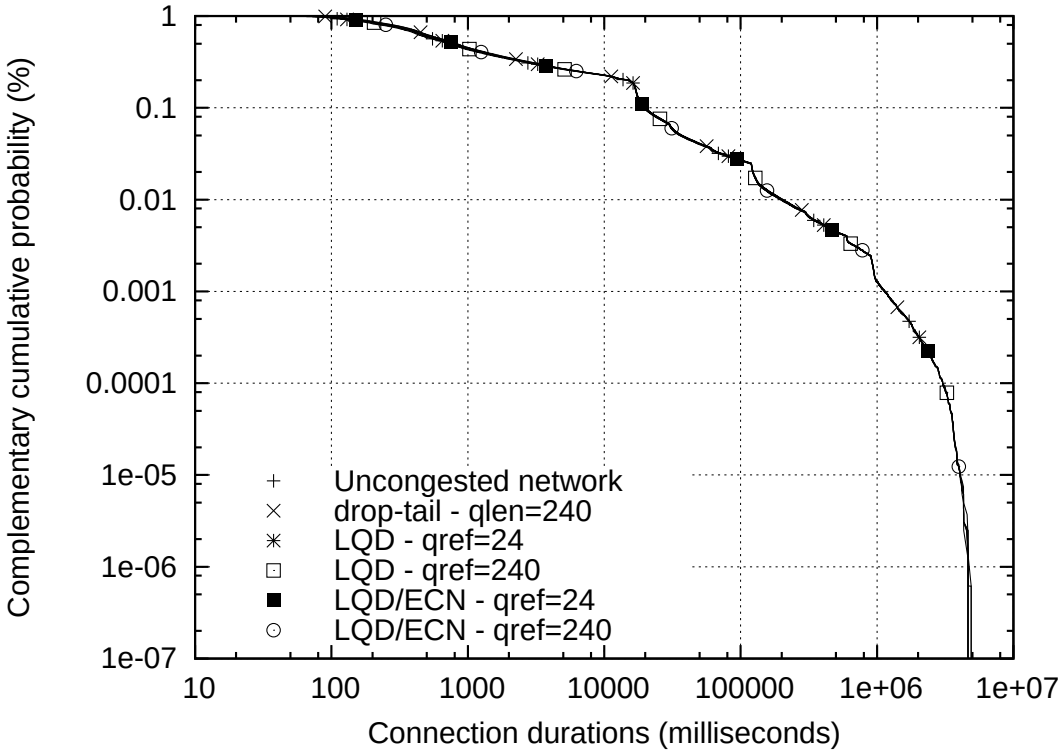

Figure 7.72: LQD/ECN performance at 90% load (CCDF)

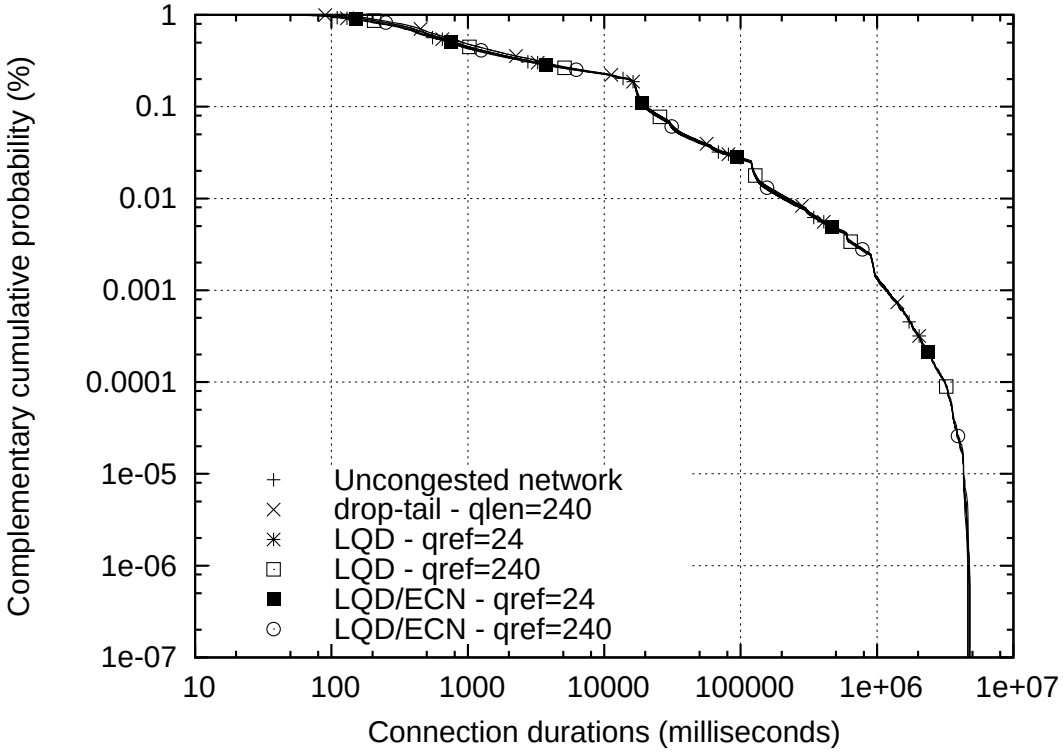

Figure 7.73: LQD/ECN performance at 95% load (CCDF)

7.3.5 Results for DCN/ECN

Figures 7.74, 7.75, and 7.76 show experimental results for DCN with general TCP applications and without link-level buffering when DCN was used with a queue reference of 24 and 240 packets. These results were obtained for DCN with and without ECN.

At 80% offered load, DCN with both queue references of 24 and 240 packets delivered the same performance with and without ECN. The performance for DCN at this load was identical to that of drop-tail and of the uncongested network.

At 90% offered load, DCN obtained the same performance with both queue references when it was used with and without ECN. Further, the performance for DCN at this load was undistinguishable from that of the uncongested network and better than the performance of drop-tail.

As the offered load increased to 95%, DCN still did not suffer any performance degradation. With or without ECN, DCN with both queue references gave performance that was identical to that of the uncongested network and considerably better than the performance of drop-tail with a queue length of 240 packets.

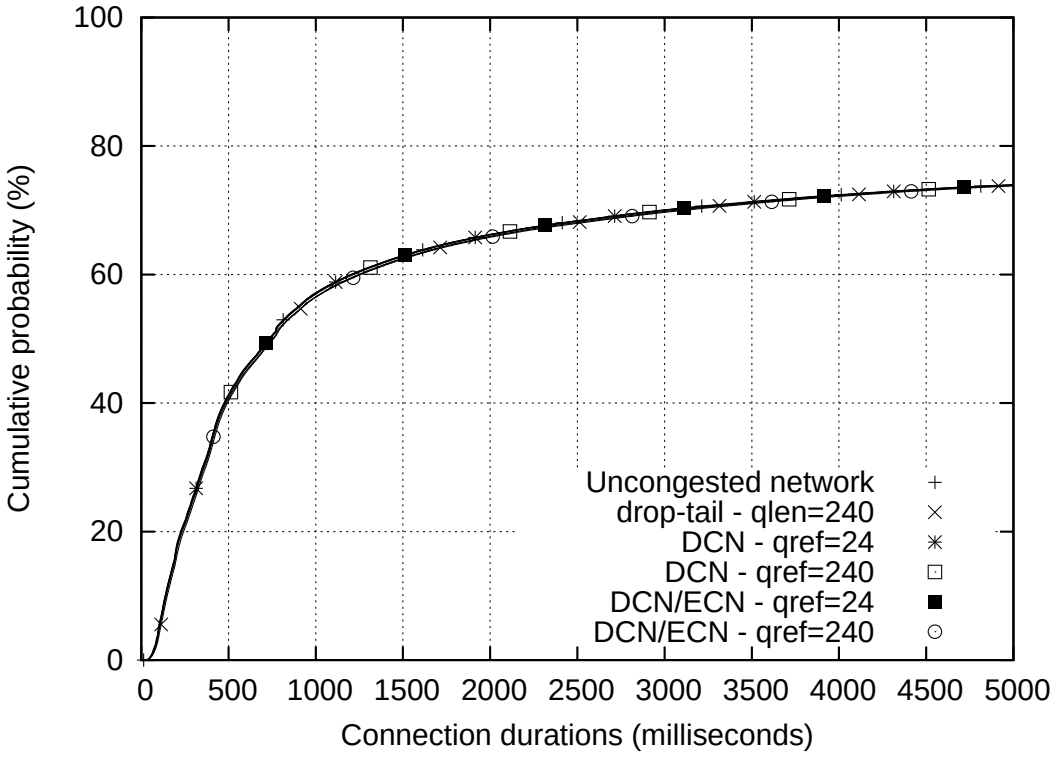

Figure 7.74: DCN/ECN performance at 80% load

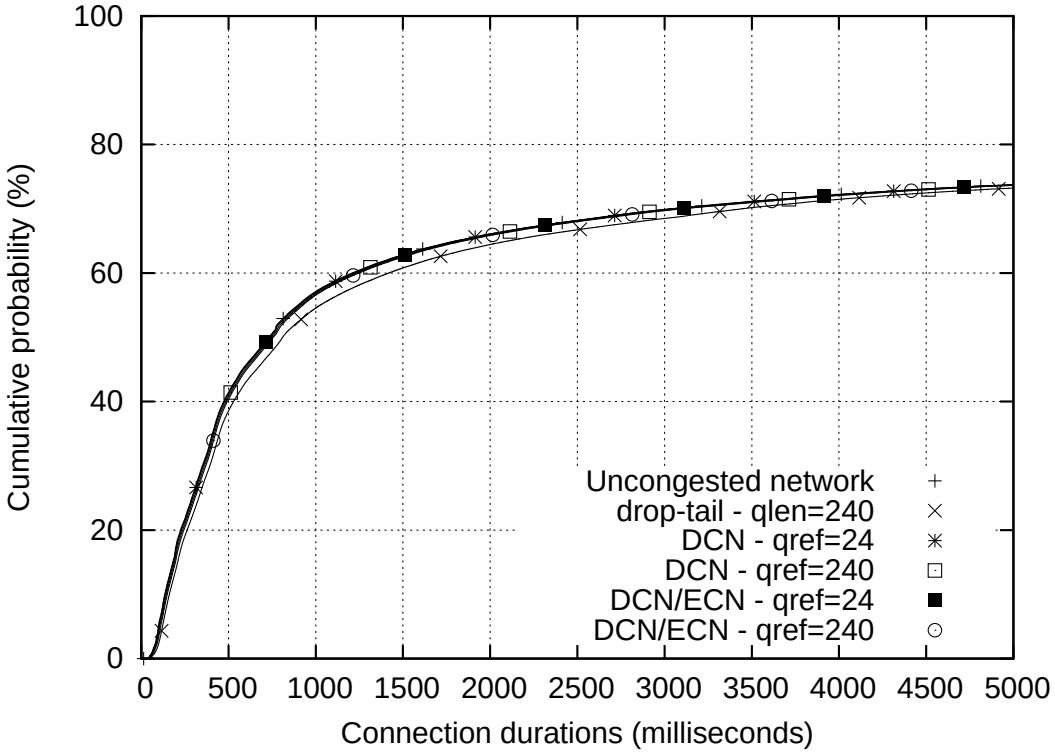

Figure 7.75: DCN/ECN performance at 90% load

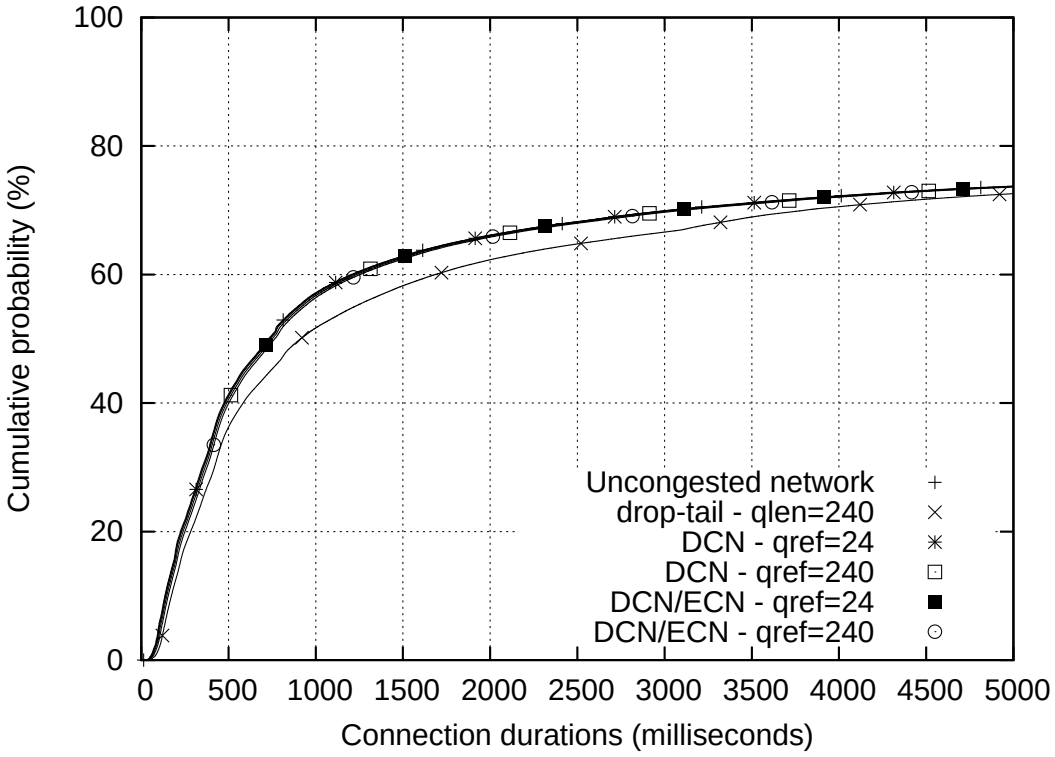

Figure 7.76: DCN/ECN performance at 95% load

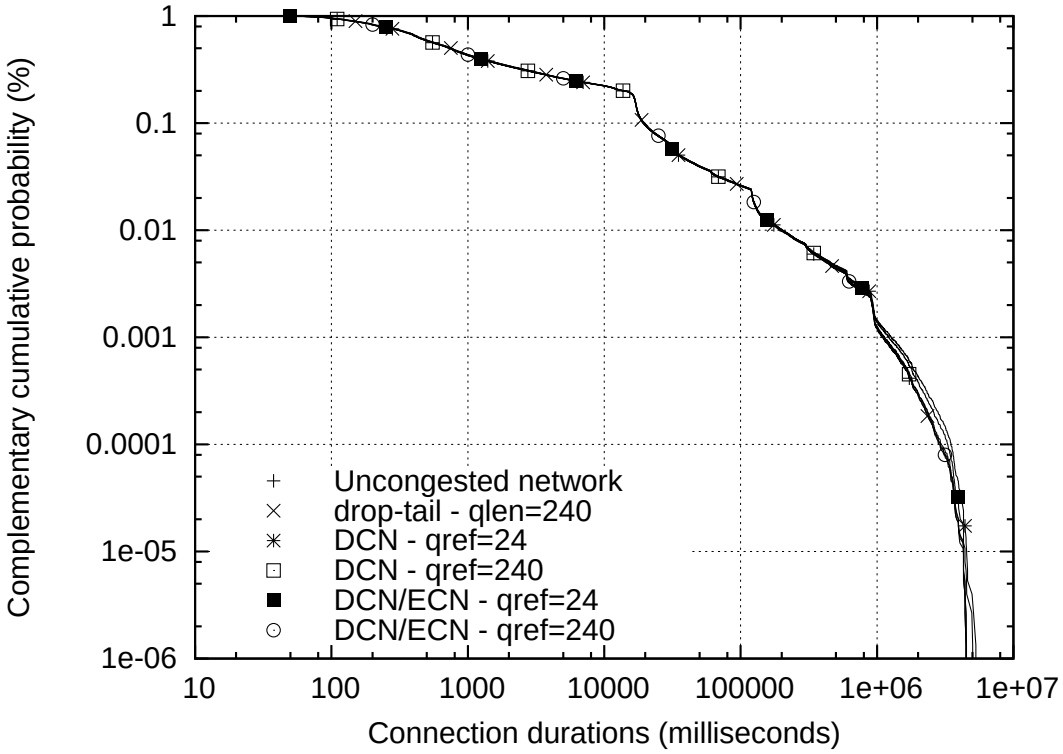

Figure 7.77: DCN/ECN performance at 80% load (CCDF)

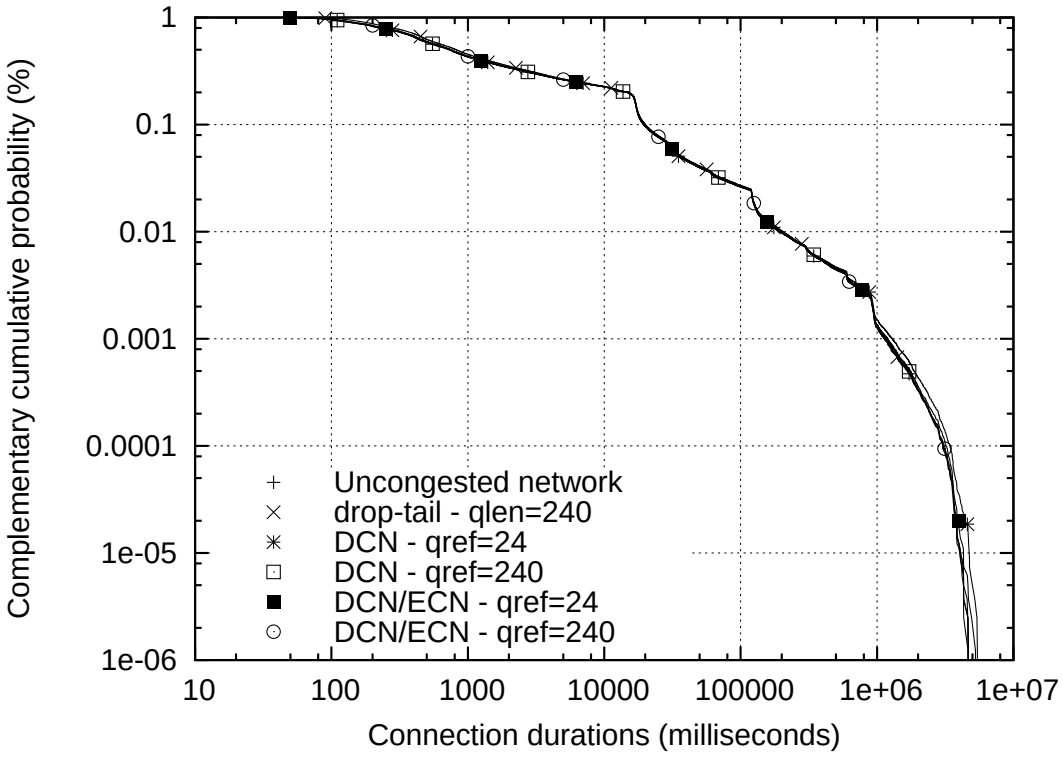

Figure 7.78: DCN/ECN performance at 90% load (CCDF)

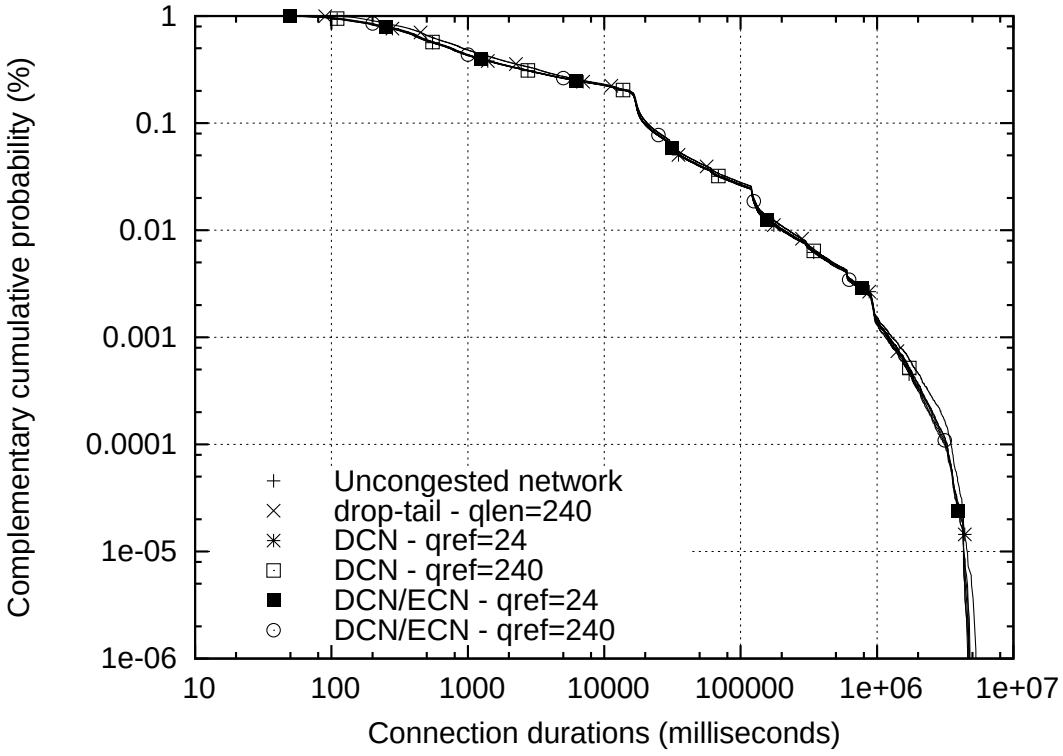

Figure 7.79: DCN/ECN performance at 95% load (CCDF)

7.4 Comparison of All Results

Figures 7.80, 7.81, and 7.82 show experimental results for PI, REM, ARED, LQD, and DCN when they were used with general TCP applications and without link-level buffering at 80%, 90%, and 95% offered loads. These results were obtained with the best parameter settings for each of the AQM algorithms. Experimental results for DCN were shown when DCN operated without ECN to demonstrate the benefits of differential treatment of flows (recall from section 7.3.5 that DCN obtained the same performance with and without ECN). Of note is the fact that PI/ECN obtained its overall best performance with a queue reference of 240 (in contrast to experiments reported in Chapters 4, 5, and 6 where PI/ECN obtained its best performance with a queue reference of 24 packets).

At 80% offered load, drop-tail with a queue length of 240 packets gave performance that was undistinguishable from that of the uncongested network and of various AQM algorithms. Thus, AQM appears to have no advantage over drop-tail for general TCP applications at 80% offered load.

At 90% offered load, all AQM algorithms and drop-tail obtained similar performance and came close to the performance of the uncongested network. However, DCN delivered slightly better performance than other AQM algorithms even when DCN was used without ECN. Drop-tail gave about the same performance as ARED/ECN "new gentle" but slightly underperformed other AQM algorithms.

At 95% offered load, DCN still delivered performance that was competitive with that of the uncongested network even when DCN was used without ECN. Further, DCN outperformed drop-tail and other AQM algorithms at this load. REM with ECN gave slightly worse performance than DCN but better than drop-tail and other AQM algorithms. Drop-tail and ARED/ECN "new gentle" obtained the same performance and underperformed other AQM algorithms at this load.

7.5 Summary

This Chapter presents experimental results for PI, REM, ARED, LQD, and DCN with general TCP traffic when link-level buffering was eliminated. Under the assumption that connection durations are the most important performance metric (and other performance metrics such as loss rate and link utilization are secondary) in evaluating AQM algorithms, following conclusions can be drawn from the results presented in this Chapter.

- Overall, link-level buffering appears to have little effects on the performance of TCP application. Experimental results for AQM algorithms that were obtained without link-level buffering were comparable to results for AQM algorithms in Chapter 6 when a link-level buffer of 254 packets was used.

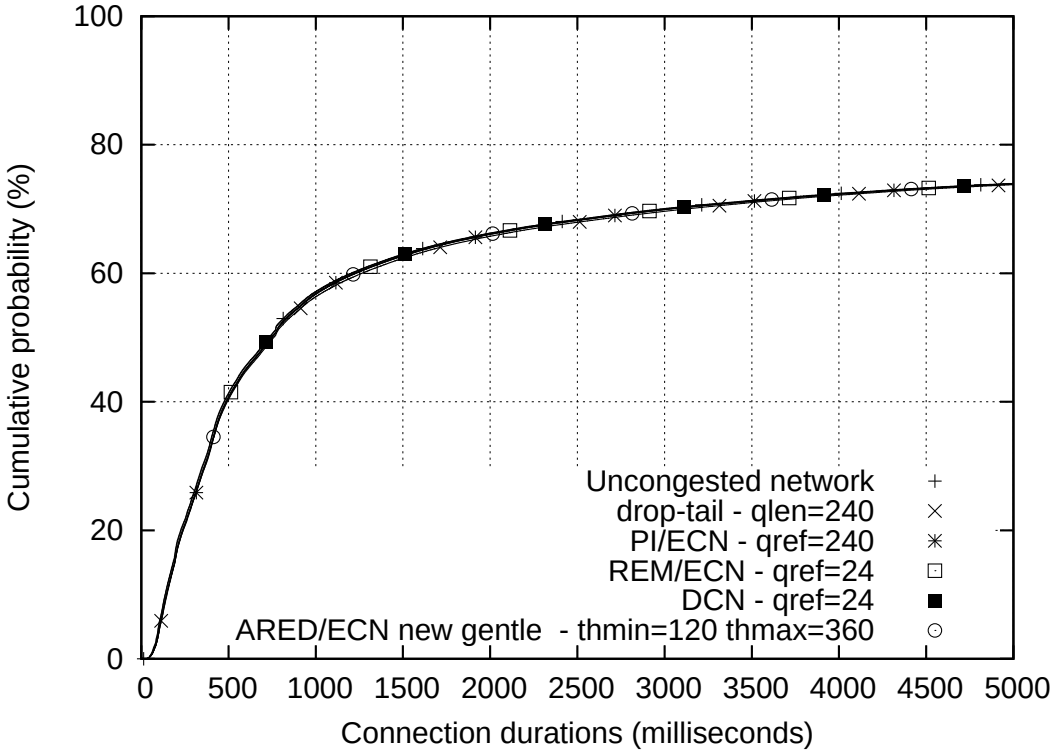

Figure 7.80: Comparison of all AQM algorithms at 80% load

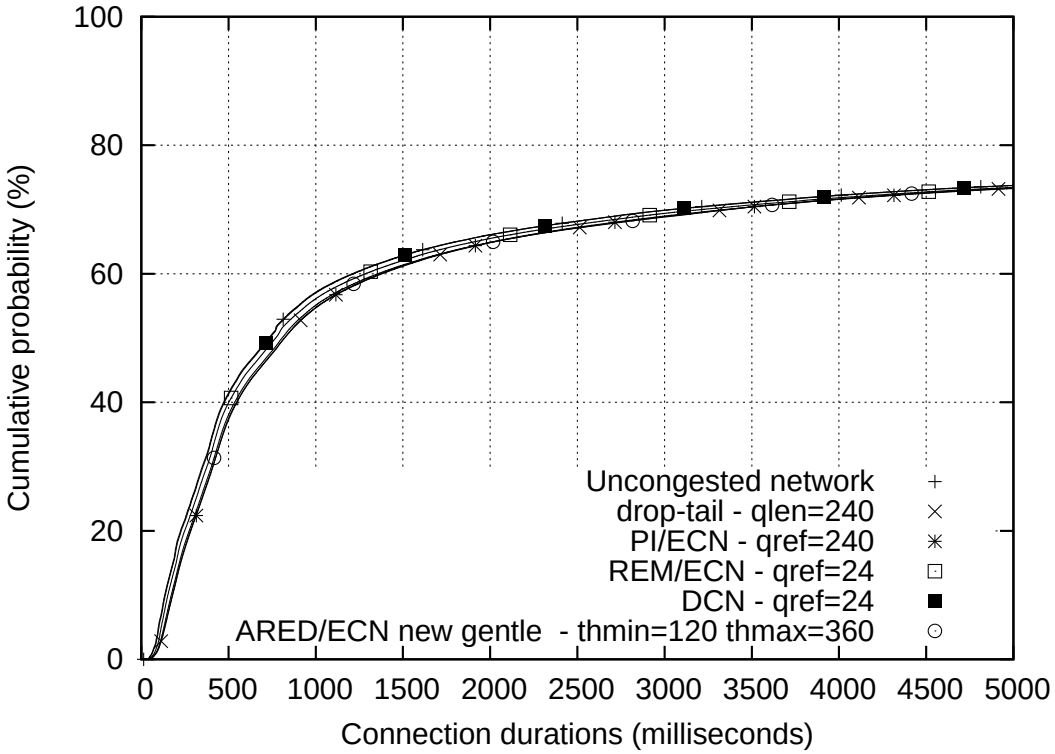

Figure 7.81: Comparison of all AQM algorithms at 90% load

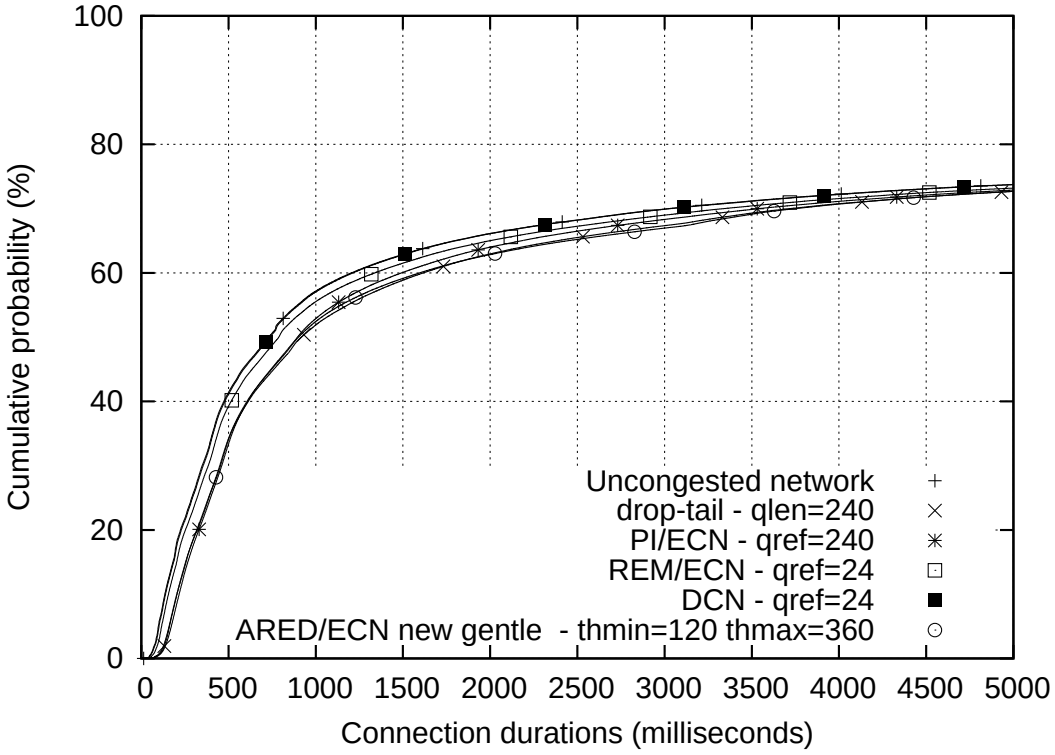

Figure 7.82: Comparison of all AQM algorithms at 95% load

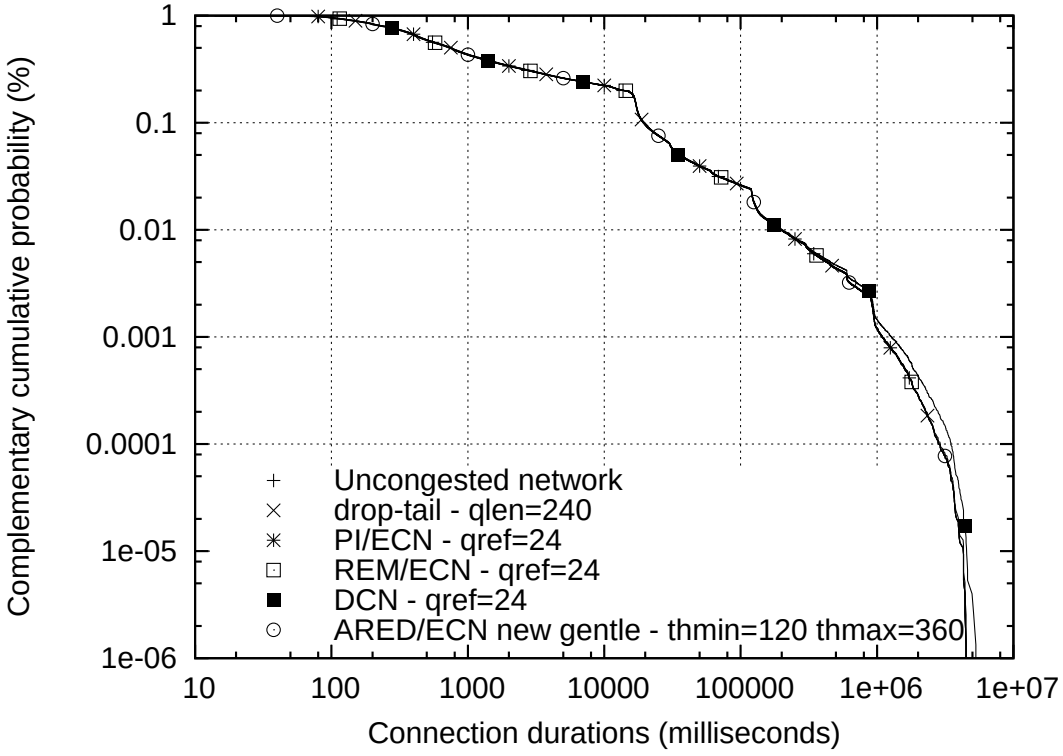

Figure 7.83: Comparison of all AQM algorithms at 80% load (CCDF)

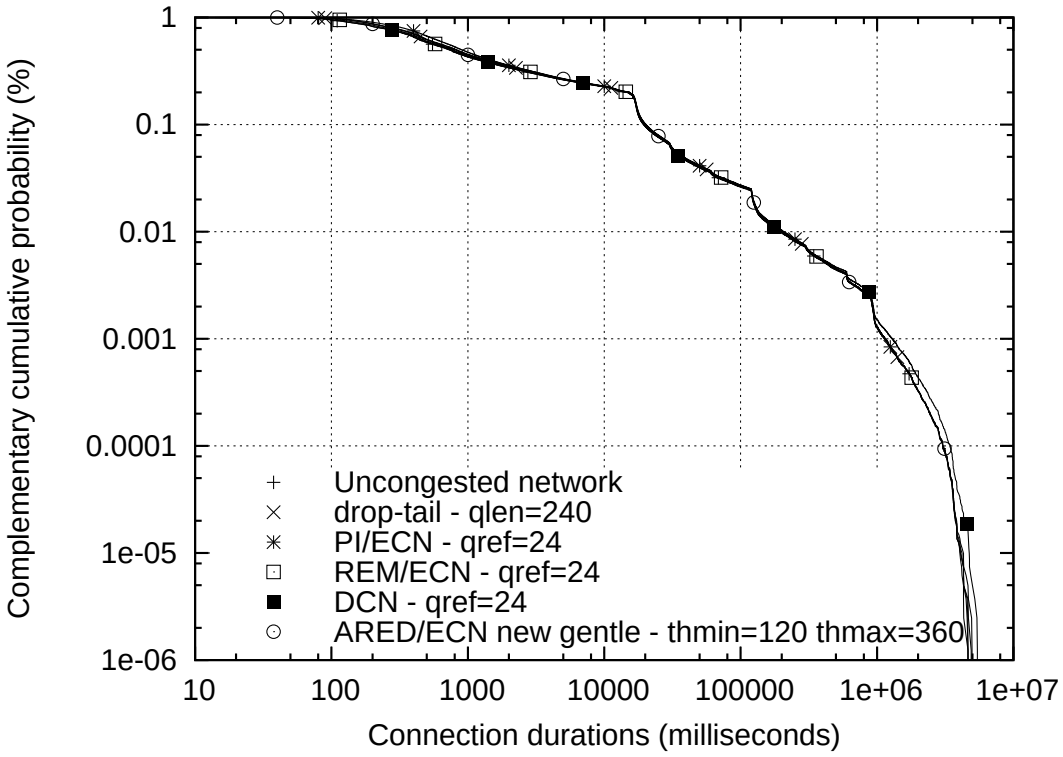

Figure 7.84: Comparison of all AQM algorithms at 90% load (CCDF)

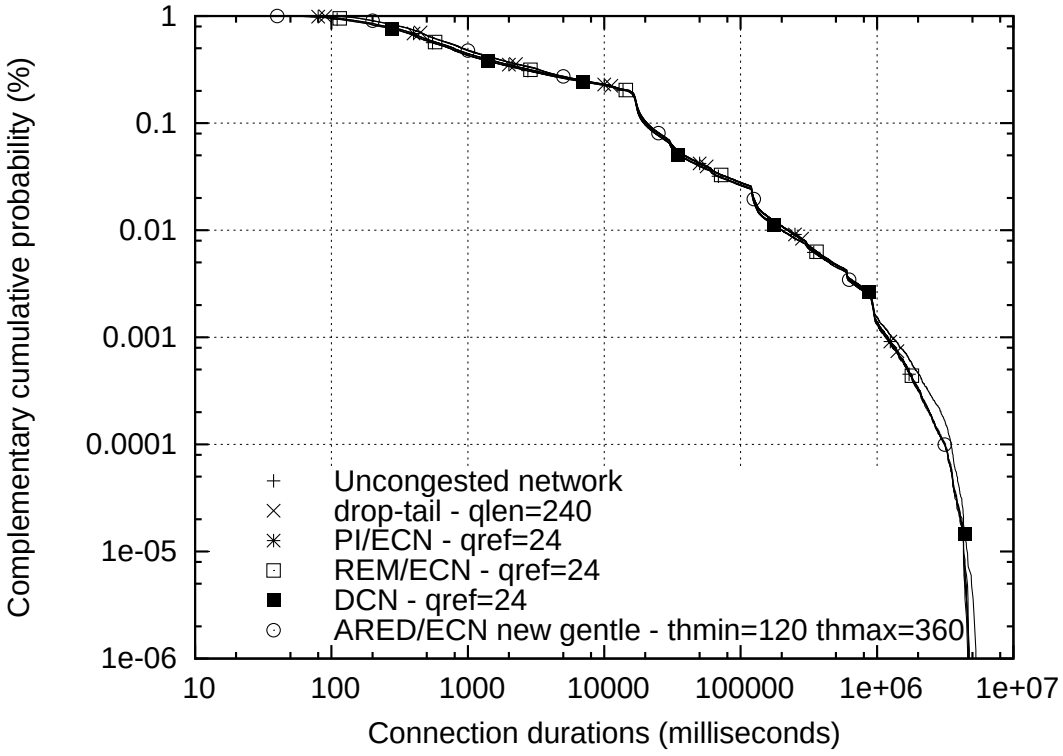

Figure 7.85: Comparison of all AQM algorithms at 95% load (CCDF)

- Contrary to results obtained in Chapter 6, drop-tail with a queue length of 24 packets was underprovisioned when used without link-level buffering. However, a drop-tail of 240 packets delivered performance that was competitive to that of AQM algorithms and the uncongested network at 80% load even when link-level buffering was eliminated. Thus, it appears that network operators could run their networks at 80% offered load without performance degradation for general TCP applications.
- At 90% of higher offered loads, AQM algorithms gave a small performance improvement for general TCP applications over drop-tail. Even with the addition of the ECN signaling protocol, the performance gain of AQM algorithms over drop-tail remained marginal.
- The DCN algorithm demonstrated the benefits of differential treatment of flows. Even without the ECN signaling protocol, DCN obtained performance that was undistinguishable from that of the uncongested network at 95% offered load.

Table 7.1: Loss rate, completed requests, and link utilization

	Offered load	Forward path loss rate (%)	Reverse path loss rate (%)	Completed connections (millions)	Forward path throughput (Mbps)	Reverse path throughput (Mbps)
Uncongested 1 Gbps network	80%	0.0	0.0	0.75	80.8	76.9
	90%	0.0	0.0	0.81	91.4	85.5
	95%	0.0	0.0	0.83	98.7	90.7
Drop-tail queue size = 24	80%	4.3	3.9	0.75	71.8	71.4
	90%	5.5	4.7	0.81	75.1	73.6
	95%	6.2	5.6	0.83	76.6	76.1
Drop-tail queue size = 240	80%	0.1	0.0	0.75	79.9	76.0
	90%	0.8	0.3	0.81	87.3	82.4
	95%	1.9	1.0	0.83	88.2	86.9
PI $q_{ref} = 24$	80%	0.1	0.0	0.75	79.6	76.3
	90%	0.8	0.1	0.81	85.2	82.6
	95%	1.4	0.7	0.83	86.1	84.7
PI/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	79.7	76.8
	90%	0.0	0.0	0.81	87.5	80.8
	95%	0.1	0.0	0.83	87.8	86.0
PI $q_{ref} = 240$	80%	0.1	0.0	0.75	75.7	72.8
	90%	0.2	0.0	0.81	86.8	83.0
	95%	0.7	0.1	0.83	87.5	86.7
PI/ECN $q_{ref} = 240$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.0	0.0	0.81	86.4	82.9
	95%	0.1	0.0	0.83	87.8	86.7
REM $q_{ref} = 24$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.6	0.2	0.81	85.5	82.3
	95%	1.3	0.6	0.83	85.5	84.6
REM/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	75.7	72.8
	90%	0.0	0.0	0.81	86.7	82.8
	95%	0.1	0.0	0.83	86.9	86.9

Continued on next page

	Offered load	Forward path loss rate (%)	Reverse path loss rate (%)	Completed connections (millions)	Forward path throughput (Mbps)	Reverse path throughput (Mbps)
REM	80%	0.0	0.0	0.75	75.7	72.8
$q_{ref} = 240$	90%	0.2	0.0	0.81	86.7	82.6
	95%	0.7	0.4	0.83	86.9	84.4
REM/ECN	80%	0.0	0.0	0.75	75.7	72.8
$q_{ref} = 240$	90%	0.0	0.0	0.81	87.2	82.9
	95%	0.0	0.0	0.83	88.2	86.6
ARED	80%	0.2	0.1	0.75	75.5	72.6
$th_{min} = 12$	90%	2.6	0.8	0.81	80.2	79.8
$th_{max} = 36$	95%	4.1	2.8	0.83	80.3	79.9
ARED/ECN	80%	0.1	0.0	0.75	75.6	72.7
$th_{min} = 12$	90%	2.4	0.7	0.81	79.3	78.7
$th_{max} = 36$	95%	3.7	2.6	0.83	79.7	79.0
ARED	80%	0.0	0.0	0.75	75.7	72.8
$th_{min} = 120$	90%	0.4	0.1	0.81	86.8	82.8
$th_{max} = 360$	95%	1.1	0.4	0.83	86.9	86.2
ARED/ECN	80%	0.0	0.0	0.75	75.8	73.0
$th_{min} = 120$	90%	0.3	0.0	0.81	86.6	82.9
$th_{max} = 360$	95%	1.0	0.3	0.83	86.8	84.8
ARED "byte	80%	0.2	0.0	0.75	75.5	72.4
mode" $th_{min} = 12$	90%	1.6	0.8	0.81	80.8	79.2
$th_{max} = 36$	95%	2.2	1.6	0.83	81.5	80.5
ARED "byte	80%	0.0	0.0	0.75	75.7	72.8
mode" $th_{min} = 120$	90%	0.4	0.1	0.81	86.1	82.1
$th_{max} = 360$	95%	1.4	0.9	0.83	86.8	82.4
ARED/ECN "new	80%	0.1	0.0	0.75	74.5	72.4
gentle" $th_{min} = 12$	90%	1.6	0.7	0.81	80.8	79.2
$th_{max} = 36$	95%	2.2	1.6	0.83	81.4	80.5
ARED/ECN "new	80%	0.0	0.0	0.75	75.7	72.8
gentle" $th_{min} = 120$	90%	0.2	0.0	0.81	86.1	82.1
$th_{max} = 360$	95%	0.9	0.2	0.83	86.4	82.3

Continued on next page

	Offered load	Forward path loss rate (%)	Reverse path loss rate (%)	Completed connections (millions)	Forward path throughput (Mbps)	Reverse path throughput (Mbps)
LQD $q_{ref} = 24$	80%	0.0	0.0	0.75	79.8	76.9
	90%	0.2	0.0	0.81	89.1	83.4
	95%	0.5	0.0	0.83	91.1	86.0
LQD/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	79.9	77.0
	90%	0.0	0.0	0.81	89.2	83.4
	95%	0.0	0.0	0.83	91.2	86.2
LQD $q_{ref} = 240$	80%	0.0	0.0	0.75	79.9	77.1
	90%	0.1	0.0	0.81	89.4	83.4
	95%	0.3	0.0	0.83	91.2	86.2
LQD/ECN $q_{ref} = 240$	80%	0.0	0.0	0.75	79.9	77.2
	90%	0.0	0.0	0.81	89.5	83.6
	95%	0.0	0.0	0.83	91.3	86.4
DCN $q_{ref} = 24$	80%	0.1	0.0	0.75	79.8	76.6
	90%	0.4	0.1	0.81	89.2	82.9
	95%	0.8	0.4	0.83	90.1	85.7
DCN/ECN $q_{ref} = 24$	80%	0.0	0.0	0.75	79.9	76.8
	90%	0.2	0.0	0.81	89.4	83.2
	95%	0.4	0.2	0.83	90.3	85.6
DCN $q_{ref} = 240$	80%	0.0	0.0	0.75	79.9	76.7
	90%	0.3	0.1	0.81	90.1	83.1
	95%	0.6	0.3	0.83	90.2	85.7
DCN/ECN $q_{ref} = 240$	80%	0.0	0.0	0.75	79.9	76.7
	90%	0.1	0.0	0.81	90.2	83.2
	95%	0.3	0.1	0.83	90.4	85.8

Table 7.2: Percentiles of response times

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
Uncongested 1 Gbps network	80%	0.730	5.897	19.365
	90%	0.730	6.152	19.505
	95%	0.730	6.166	19.513
Drop-tail queue size = 24	80%	0.757	6.004	19.498
	90%	0.810	6.589	20.074
	95%	0.913	7.084	20.649
Drop-tail queue size = 240	80%	0.759	6.083	19.563
	90%	0.820	6.525	19.988
	95%	0.916	7.016	20.584
PI $q_{ref} = 24$	80%	0.749	5.981	19.452
	90%	0.790	6.633	20.159
	95%	0.830	7.016	20.673
PI/ECN $q_{ref} = 24$	80%	0.749	5.977	19.449
	90%	0.900	6.692	20.187
	95%	0.788	6.738	20.262
PI $q_{ref} = 240$	80%	0.750	5.985	19.456
	90%	0.818	6.502	19.984
	95%	0.909	7.021	20.644
PI/ECN $q_{ref} = 240$	80%	0.750	5.984	19.453
	90%	0.820	6.585	20.036
	95%	0.895	6.928	20.512
REM $q_{ref} = 24$	80%	0.745	5.975	19.447
	90%	0.787	6.596	20.104
	95%	0.825	7.010	20.659
REM/ECN $q_{ref} = 24$	80%	0.743	5.970	19.440
	90%	0.770	6.437	19.891
	95%	0.788	6.689	20.206
REM $q_{ref} = 240$	80%	0.749	5.979	19.447
	90%	0.804	6.478	19.950
	95%	0.885	6.965	20.573
REM/ECN $q_{ref} = 240$	80%	0.748	5.980	19.445
	90%	0.799	6.428	19.886
	95%	0.858	6.747	20.252

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
ARED	80%	0.750	6.010	19.509
$th_{min} = 12$	90%	0.958	7.346	21.004
$th_{max} = 36$	95%	1.272	8.296	22.189
ARED/ECN	80%	0.748	5.995	19.488
$th_{min} = 12$	90%	0.931	7.331	20.952
$th_{max} = 36$	95%	1.329	8.591	22.517
ARED	80%	0.748	5.978	19.446
$th_{min} = 120$	90%	0.810	6.523	19.995
$th_{max} = 360$	95%	0.916	7.027	20.615
ARED/ECN	80%	0.760	6.094	19.566
$th_{min} = 120$	90%	0.808	6.498	19.962
$th_{max} = 360$	95%	0.923	7.105	20.691
ARED "byte	80%	0.767	6.173	19.690
mode" $th_{min} = 12$	90%	0.845	7.134	20.756
$th_{max} = 36$	95%	0.928	7.658	21.489
ARED "byte	80%	0.747	5.976	19.443
mode" $th_{min} = 120$	90%	0.801	6.526	20.008
$th_{max} = 360$	95%	0.913	7.335	21.073
ARED "new	80%	0.745	5.982	19.467
gentle" $th_{min} = 12$	90%	0.862	7.076	20.495
$th_{max} = 36$	95%	1.037	7.771	21.258
ARED "new	80%	0.747	5.977	19.442
gentle" $th_{min} = 120$	90%	0.806	6.490	19.935
$th_{max} = 360$	95%	0.901	6.997	20.489
LQD	80%	0.740	5.940	19.408
$q_{ref} = 24$	90%	0.762	6.389	19.838
	95%	0.777	6.603	20.105
LQD/ECN	80%	0.740	5.938	19.407
$q_{ref} = 24$	90%	0.810	6.424	19.853
	95%	0.760	6.453	19.880
LQD	80%	0.740	5.941	19.411
$q_{ref} = 240$	90%	0.771	6.330	19.747
	95%	0.816	6.599	20.088

Continued on next page

	Offered loads	50th percentile (seconds)	75th percentile (seconds)	90th percentile (seconds)
LQD/ECN $q_{ref} = 240$	80%	0.740	5.941	19.409
	90%	0.772	6.369	19.780
	95%	0.809	6.552	20.010
DCN $q_{ref} = 24$	80%	0.733	5.941	19.382
	90%	0.735	6.176	19.513
	95%	0.736	6.183	19.516
DCN/ECN $q_{ref} = 24$	80%	0.735	5.943	19.399
	90%	0.736	6.177	19.526
	95%	0.738	6.192	19.540
DCN $q_{ref} = 240$	80%	0.738	5.963	19.429
	90%	0.746	6.235	19.614
	95%	0.750	6.254	19.623
DCN/ECN $q_{ref} = 240$	80%	0.752	6.074	19.554
	90%	0.756	6.252	19.639
	95%	0.765	6.290	19.676

Chapter 8

Conclusions and Future Work

AQM has been proposed by networking researchers and the Internet Engineering Task Force as a mechanism to preserve and improve the performance of Internet applications. AQM algorithms operate on network routers and detect congestion by typically monitoring the instantaneous or average router queue. When AQM algorithms anticipate that congestion is about to occur, they provide end systems with a congestion signal by marking or dropping arriving packets. This proactive approach is in contrast to the reactive approach of pure end systems' congestion control that only reacts to congestion after router queues already overflowed.

Many AQM algorithms have been proposed in recent years but none of them have been thoroughly investigated under comparable (or realistic) conditions in a real network. Moreover, existing performance studies have concentrated on network-centric measures of performance and have not considered application-level performance measures such as response time. In this dissertation, I investigated the effects of a large collection of AQM algorithms on the performance of Web and general TCP applications under realistic conditions in a real network.

Experimental results for AQM algorithms with Web applications, arguably the currently most important Internet application, lead to the following conclusions. These conclusions were drawn under the assumption that response times are the most important performance metrics for Web applications (and network-centric performance metrics such as loss rate and link utilization are secondary) in evaluating the effects of AQM algorithms on applicaiton performance.

- At offered load of 80%, conventional drop-tail queues delivered performance that was competitive to the performance of all AQM algorithms under evaluation. Thus, AQM algorithms appear to have no advantage over drop-tail at offered load of 80% or lower.

- Further, since drop-tail closely approximated the performance of the uncongested network at 80% load, it seems that network operators can run their networks at offered

loads up to 80% with a minimum performance degradation for Web applications.

- When the offered load increased to 90% or higher, AQM algorithms only provided marginal performance improvement over drop-tail when they were used with packet drops. Further, in most cases, the small performance improvement provided by AQM cannot offset the significant performance degradation that all AQM algorithms suffer at these extreme loads.

- However, when AQM algorithms were used with the ECN signaling protocol at 90% load or higher, they provided significant performance improvement over drop-tail. Thus, it appears that when ECN is used, network operators can run their networks near saturation levels with only a small performance degradation for Web applications.

- The ARED algorithm, a contemporary redesign of the classical RED algorithm, gave the poorest performance among all AQM algorithms. Oftentimes, the performance of ARED was worse than that of conventional drop-tail queues. These results were unchanged by the addition of the ECN signaling protocol.

- Further investigations in the poor performance for ARED revealed a number of its design flaws. Two modified algorithms, ARED "byte mode" and ARED "new gentle", were proposed in this dissertation to fix the design flaws for ARED. These new ARED algorithms improved the performance for ARED significantly.

- Thanks to the heavy distribution in flow sizes of Web traffic, differential treatment of flows can improve the performance for Web applications significantly. For example, the DCN algorithm proposed in this thesis outperformed all existing AQM algorithms even when it was used without the ECN signaling protocol.

- When the minimum RTTs used to emulate propagation delays in experiments changed from a uniform to a more general (and more heavy-tailed) distribution, the performance improvement of AQM algorithms over drop-tail becomes less significant. This is because when RTTs increase, they become a more important factor and dominate the effects of packet loss on the performance of Web applications.

When experiments were performed for AQM algorithms with general TCP applications, the following conclusions were drawn from the experimental results. These conclusions were made when assuming that connection durations, a generalized performance metric of response times for Web applications, are the most important performance metrics for general TCP applications.

- At 90% offered load or lower, the performance for drop-tail with a queue length of 240 packets was very close to that of all AQM algorithms and the performance of the

uncongested network. Thus, it appears that Internet Service Providers could operate their network near saturation levels without causing noticeable performance degradation for their customers' applications. Further, AQM seems to have no advantage over drop-tail for general TCP applications at 90% offered load or below.

- As the offered load increased to 95%, AQM algorithms gave very small performance improvement over drop-tail when they were used with packet drops. Further, when AQM algorithms were used with ECN, their performance improvement over drop-tail for general TCP applicaitons remained marginal.

- ARED was once again the worst performing AQM algorithm but the performance for ARED was improved considerably with the two modifications ARED "byte mode" and "new gentle" with ECN.

- The DCN algorithm demonstrated the power of differential treatment of flows. Even without the ECN marking protocol, DCN obtained performance that was indistinguishable from the performance of the uncongested network at 95% offered load.

- When operated without link-level buffering, a short drop-tail queue gave very poor performance that was significantly lower than that of AQM algorithms and of the uncongested network.

Overall, existing AQM algorithms show promising results. When operated in combination with the ECN signaling protocol, existing AQM algorithms reduce loss rates, increase link utilization, and provide significant performance improvement for response times. If packet marking is not possible, the dissertation also shows how a form of different treatment of flows can achieve a similar positive performance improvement.

In future work, it would be interesting to evaluate AQM algorithms under different conditions such as multihop networks, wireless networks, or gigabit networks. Further, it would be very interesting to perform these experiments either with standard TCP protocol or with recently proposed congestion control protocols such as XCP [KHR02], HighSpeed TCP [Flo03], BIC-TCP [XHR04], and FAST [JWL04].

BIBLIOGRAPHY

[AKSJ03] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in TCP round-trip times. In *Proceedings of Internet Measurement Conference*, 2003.

[ALLY01] Sanjeewa Athuraliya, Victor H. Li, Steven H. Low, and Qinghe Yin. REM: Active queue management. *IEEE Network*, 15(3):48–53, May 2001.

[Ath02] Sanjeewa Athurliya. A note on parameter values of REM with Reno-like algorithms. Available at `http://netlab.caltech.edu`, March 2002.

[BB01] Deepak Bansal and Hari Balakrishnan. Binomial congestion control algorithms. In *Proceedings of IEEE INFOCOM*, April 2001.

[BBC+98] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, December 1998.

[BCC+98] Bob Braden, David D. Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. K. Ramakrishnan, Scott Shenker, John Wroclawski, and Lixia Zhang. Recommendations on queue management and congestion avoidance in the Internet. RFC 2309, April 1998.

[BCS94] Bob Braden, David D. Clark, and Scott Shenker. Integrated services in the Internet architecture: An overview. RFC 1633, June 1994.

[BOP94] Lawrence Brakmo, Sean O'Malley, and Larry Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM*, pages 24–35, October 1994.

[CF98] David D. Clark and Wenjia Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, August 1998.

[Cho98] Kenjiro Cho. A framework for alternate queueing: Towards traffic management by PC-UNIX based routers. In *USENIX*, June 1998.

[CJOS01] Mikkel Christiansen, Kevin Jeffay, David Ott, and F. Donelson Smith. Tuning RED for web traffic. *IEEE/ACM Transactions on Networking*, 9(3):249–264, June 2001.

[CJS04] Felix Hernandez Campos, Kevin Jeffay, and F. Donelson Smith. Generating realistic TCP workloads. In *The Computer Measurement Group's 2004 International Conference*, 2004.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM*, September 1989.

[ei05] end2end interest. Is sanity in ns2? Email to end2end-interest mailing list, September 2005.

[EV02] Christian Estan and George Varghese. New directions in traffic measurement and accounting control algorithms. In *Proceedings of ACM SIGCOMM*, August 2002.

[FB00] Victor Firoiu and Marty Borden. A study of active queue management for congestion control. In *Proceedings of IEEE INFOCOM*, March 2000.

[FF99] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4), August 1999.

[FGS01] Sally Floyd, Ramakrishna Gummadi, and Scott Shenker. Adaptive RED: An algorithm for increasing the robustness of RED's active queue management. under submission, August 2001.

[FGW98] Anja Feldmann, Anna Gilbert, and Walter Willinger. Data networks as cascades: Explaining the multifractal nature of Internet WAN traffic. In *Proceedings of ACM SIGCOMM*, September 1998.

[FHGW99] Anja Feldmann, Polly Huang, Anna C. Gilbert, and Walter Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proceedings of ACM SIGCOMM*, August 1999.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[FKSS99] Wu-Chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. A self-configuring RED gateway. In *Proceedings of IEEE INFOCOM*, March 1999.

[FKSS01a] Wu-Chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. Blue: An alternative approach to active queue management. In *Proceedings of Network and Operating System Support for Digital Audio and Video*, June 2001.

[FKSS01b] Wu-Chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. Stochastic Fair Blue: A queue management algorithm for enforcing fairness. In *Proceedings of IEEE INFOCOM*, April 2001.

[FKSS02] Wu-Chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. The Blue active queue management algorithms. *IEEE/ACM Transactions on Networking*, 10(4), August 2002.

[Flo00a] Sally Floyd. Congestion control principles. RFC 2914, September 2000.

[Flo00b] Sally Floyd. Recommendation on using the "gentle_" variant of RED. `http://www.icir.org/floyd/red/gentle.html`, March 2000.

[Flo03] Sally Floyd. Highspeed TCP for large congestion windows. RFC 3649, December 2003.

[GM01] Liang Guo and Ibrahim Matta. The war between mice and elephants. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2001.

[HMTG01] C. V. Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. On designing improved controllers for AQM routers supporting TCP flows. In *Proceedings of IEEE INFOCOM*, April 2001.

[Inc98] Ganymede Software Inc. Chariot 2.2, 1998. `http://www.ganymedesoftware.com/html/chariot.htm` (link is now broken).

[JK88] Van Jacobson and Michael Karels. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, pages 314–329, 1988.

[JWL04] Cheng Jin, David X. Wei, and Steven H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *Proceedings of IEEE INFOCOM*, March 2004.

[KHR02] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM*, August 2002.

[KS01] Srisankar Kunniyur and R. Srikant. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In *Proceedings of ACM SIGCOMM*, August 2001.

[KV03] Jasleen Kaur and Harrick Vin. Core-stateless guaranteed throughput networks. In *Proceedings of IEEE INFOCOM*, April 2003.

[LAJS03] Long Le, Jay Aikat, Kevin Jeffay, and F. Donelson Smith. The effects of active queue management on web performance. In *Proceedings of ACM SIGCOMM*, August 2003.

[LAJS04] Long Le, Jay Aikat, Kevin Jeffay, and F. Donelson Smith. Differential congestion notification: Taming the elephants. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, October 2004.

[LJS06] Long Le, Kevin Jeffay, and F. Donelson Smith. A loss and queuing delay controller for buffer management. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, July 2006.

[LM97] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proceedings of ACM SIGCOMM*, August 1997.

[LPW+02] Steven H. Low, Fernando Paganini, Jiantao Wang, Sachin Adlakha, and John C. Doyle. Dynamics of TCP/RED and a scalable control. In *Proceedings of IEEE INFOCOM*, June 2002.

[MAF05] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the Internet. *ACM Computer Communication Review*, 35(2), April 2005.

[Mah97] Bruce A. Mah. An empirical model of HTTP network traffic. In *Proceedings of IEEE INFOCOM*, pages 592–600, 1997.

[MBDL99] Martin May, Jean Bolot, Christophe Diot, and Bryan Lyles. Reasons not to deploy RED. In *IWQoS*, 1999.

[McK90] Paul McKenney. Stochastic fairness queuing. In *Proceedings of IEEE INFOCOM*, June 1990.

[MFW01] Ratul Mahajan, Sally Floyd, and David Wetherall. Controlling high-bandwidth flows at the congested routers. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, August 2001.

[NGBS+97] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of ACM SIGCOMM*, September 1997.

[NLA05] NLANR. `http://www.nlanr.net`, August 2005.

[NRSA01] Erich M. Nahum, Marcel Rosu, Srinivasan Seshan, and Jussara Almeida. The effects of wide-area conditions on WWW server performance. In *Proceedings of ACM SIGMETRICS*, June 2001.

[OKM96] Teunis J. Ott, J.H.B. Kemperman, and Matt Mathis. The stationary behavior of idealized TCP congestion behavior. Available at `http://web.njit.edu/~ott/Papers/`, 1996.

[OLW99] Teunis J. Ott, T. V. Lakshman, and Larry H. Wong. SRED: Stabilized RED. In *Proceedings of IEEE INFOCOM*, March 1999.

[Pax99] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[PBPS03] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. *ACM Computer Communication Review*, 33(2):23–39, April 2003.

[PF95] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[PF01] Jitendra Padhye and Sally Floyd. On inferring TCP behavior. In *Proceedings of ACM SIGCOMM*, pages 287–298, September 2001.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM*, September 1998.

[PG93] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3), June 1993.

[PPP00] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. CHOKe, a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proceedings of IEEE INFOCOM*, March 2000.

[RFB01] K. K. Ramakrishnan, Sally Floyd, and David L. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, September 2001.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[SA00] Jamal Hadi Salim and Uvaiz Ahmed. Performance evaluation of explicit congestion notification (ECN) in IP networks. RFC 2884, July 2000.

[SCJO01] F. Donelson Smith, Felix Hernandez Campos, Kevin Jeffay, and David Ott. What TCP/IP protocol headers can tell us about the web. In *Proceedings of ACM SIGMETRICS*, pages 245–256, June 2001.

[SSZ98] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proceedings of ACM SIGCOMM*, September 1998.

[SV95] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of ACM SIGCOMM*, September 1995.

[Wei03] Michele Clark Weigle. *Investigating the Use of Synchronized Clocks in TCP Congestion Control*. PhD thesis, University of North Carolina at Chapel Hill, August 2003.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high variability: statistical analysis of ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, February 1997.

[XHR04] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control for fast, long distance networks. In *Proceedings of IEEE INFOCOM*, March 2004.

[ZBPS02] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of Internet flow rates. In *Proceedings of ACM SIGCOMM*, August 2002.

[ZSC91] Lixia Zhang, Scott Shenker, and David Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of ACM SIGCOMM*, 1991.