Fuzzy Logic Based Robust Control of Queue Management and Optimal Treatment of Traffic over TCP/IP Networks

Zhi Li
BSc(Sci), MSc(Eng)

Department of Mathematics and Computing
The University of Southern Queensland
Toowoomba, Australia

Supervised by :
Dr Zhongwei Zhang

A dissertation submitted in partial fulfillment of
the requirements of the degree of Doctor of Philosophy in Computing

2005

Abstract

Improving network performance in terms of efficiency, fairness in the bandwidth, and system stability has been a research issue for decades. Current Internet traffic control maintains sophistication in end TCPs but simplicity in routers. In each router, incoming packets queue up in a buffer for transmission until the buffer is full, and then the packets are dropped. This router queue management strategy is referred to as *Drop Tail*. End TCPs eventually detect packet losses and slow down their sending rates to ease congestion in the network. This way, the aggregate sending rate converges to the network capacity.

In the past, Drop Tail has been adopted in most routers in the Internet due to its simplicity of implementation and practicability with light traffic loads. However Drop Tail, with heavy-loaded traffic, causes not only high loss rate and low network throughput, but also long packet delay and lengthy congestion conditions. To address these problems, active queue management (AQM) has been proposed with the idea of proactively and selectively dropping packets before an output buffer is full. The essence of AQM is to drop packets in such a way that the congestion avoidance strategy of TCP works most effectively. Significant efforts in developing AQM have been made since random early detection (RED), the first prominent AQM other than Drop Tail, was introduced in 1993.

Although various AQMs also tend to improve fairness in bandwidth among flows, the vulnerability of short-lived flows persists due to the conservative nature of TCP. It has been revealed that short-lived flows take up traffic with a relatively small percentage of bytes but in a large number of flows. From the user's point of view, there is an expectation of timely delivery of short-lived flows.

Our approach is to apply artificial intelligence technologies, particularly fuzzy logic (FL), to address these two issues: an effective AQM scheme, and preferential treatment

for short-lived flows. Inspired by the success of FL in the robust control of nonlinear complex systems, our hypothesis is that the Internet is one of the most complex systems and FL can be applied to it. First of all, state of the art AQM schemes outperform Drop Tail, but their performance is not consistent under different network scenarios. Research reveals that this inconsistency is due to the selection of congestion indicators. Most existing AQM schemes are reliant on queue length, input rate, and extreme events occurring in the routers, such as a full queue and an empty queue. This drawback might be overcome by introducing an indicator which takes account of not only input traffic but also queue occupancy for early congestion notification. The congestion indicator chosen in this research is traffic load factor. Traffic load factor is in fact dimensionless and thus independent of link capacity, and also it is easy to use in more complex networks where different traffic classes coexist. The traffic load indicator is a descriptive measure of the complex communication network, and is well suited for use in FL control theory. Based on the traffic load indicator, AQM using FL – or FLAQM – is explored and two FLAQM algorithms are proposed.

Secondly, a mice and elephants (ME) strategy is proposed for addressing the problem of the vulnerability of short-lived flows. The idea behind ME is to treat short-lived flows preferably over bulk flows. ME's operational location is chosen at user premise gateways, where surplus processing resources are available compared to other places. By giving absolute priority to short-lived flows, both short and long-lived flows can benefit. One problem with ME is *starvation* of elephants or long-lived flows. This issue is addressed by dynamically adjusting the threshold distinguishing between mice and elephants with the guarantee that minimum capacity is maintained for elephants. The method used to dynamically adjust the threshold is to apply FL. FLAQM is deployed to control the elephant queue with consideration of capacity usage of mice packets. In addition, flow states in a ME router are periodically updated to maintain the data storage.

The application of the traffic load factor for early congestion notification and the ME strategy have been evaluated via extensive experimental simulations with a range of traffic load conditions. The results show that the proposed two FLAQM algorithms outperform some well-known AQM schemes in all the investigated network circumstances in terms of both user-centric measures and network-centric measures. The ME

strategy, with the use of FLAQM to control long-lived flow queues, improves not only the performance of short-lived flows but also the overall performance of the network without disadvantaging long-lived flows.

Certification of Dissertation

I certify that the ideas, experimental work, results, analyses, software, and conclusions reported in this dissertation are entirely my own effort, except where otherwise acknowledged. I also certify that the work is original and has not been previously submitted for any other award or degree.

Signature of Candidate Date

ENDORSEMENT

Signature of Supervisor/s Date

Acknowledgements

I first of all wish to thank my principal supervisor Dr. Zhongwei Zhang at University of Southern Queensland. Dr. Zhongwei Zhang not only made this thesis possible, but kept me on the right track through his guidance and encouragement. His wide knowledge and research style made this study richer both in content and in form.

I also have been very fortunate to be instructed by my associate supervisor Associate Professor Ron Addie, who has an open mind, insights, and inspiration, especially during the period when Zhongwei was away for six months Academic Development Leave in 2003.

I want to thank Dr. Fabrice Clérot in R&D of France Telecom. Many discussions with him have greatly affected my research from its early stage. I also have been very fortunate to collaborate with him in one paper. I am grateful to the three examiners for constructive comments that led to the final version of this thesis.

I would like to thank the Australian government for awarding me an International Postgraduate Research Scholarship for my three-and-a-half-year PhD study. My gratitude also goes to the Department of Maths and Computing for its financial support for my travels to several conferences during my study.

I am indebted to the Queensland Parallel Supercomputing Foundation (QPSF) to provide our research team an account on the Australian Partnership for Advanced Computing (APAC) national facility at Australian National University. The large amount of simulations could not have been completed without the support from the staff of both QPSF and APAC.

Special thanks go to Mr. Greg Otto for a careful proof-reading of the manuscript. I am also grateful to him and his wife Pirkko for their care and concern. With them and other friends in Toowoomba, I have enjoyed the Aussie culture.

Lastly, I would to express my gratitude to my parents Naiyun Li and Chenyan Lai and my sisters Jun Li and Juan Li. To them I dedicate this thesis.

Acronyms and Abbreviations

QoS	quality of service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
FIFO	first in first out
SYN	TCP synchronization
ACK	acknowledgement
IETF	The Internet Engineering Task Force
AQM	active queue management
RED	random early detection
RTT	round-trip time
FL	fuzzy logic
Diffserv	Differentiated Services
Intserv	Integrated Services
ME	mice and elephants
ECN	Explicit Congestion Notification
FLAQM	FL-based AQM
FLC	fuzzy logic controller
RTO	retransmission timeout
AIMD	additive-increase multiplicative-decrease
VoIP	voice over IP
ATM	asynchronous transfer mode
ITU	the International Telecommunication Union
B-ISDN	Broadband Integrated Services Digital Networks
VPI	virtual path identifier
VCI	virtual circuit identifier
CBR	Constant Bit Rate
rt-VBR	Real-time Variable Bit Rate
nrt-VBR	Non-real-time Variable Bit Rate
UBR	Unspecified Bit Rate
ABR	Available Bit Rate
GFR	Guaranteed Frame Rate
CAC	Connection Admission Control
UPC	Usage Parameter Control
NPC	Network Parameter Control
CLP	cell loss priority
VP	virtual paths
RSVP	Resource Reservation Protocol
DSCP	Diffserv Codepoint
BA	behavior aggregate
PHB	per-hop behavior
MF	multi-field

SLA	service level agreement
AF	Assured Forwarding
EF	Expedited Forwarding
BE	best-effort
BB	bandwidth broker
FEC	Forwarding Equivalence Class
RIO	RED with IN/OUT packets
EPD	Early Packet Discard
FBA	Fair Buffer Allocation
CE	Congestion Experienced
WRED	Weighted RED
ARED	Adaptive RED
SRED	Stabilized RED
ADR	Acceptance and Departure Rate
REM	Random Exponential Marking
AVQ	Adaptive Virtual Queue
PI	Proportional Integral
VS	variable structure
FEM	Fuzzy Explicit Marking
EWAQ	exponentially weighted average queue length
AI	artificial intelligent
NNs	neural networks
GAs	genetic algorithms
ISP	Internet Service Provider

List of Publications

This appendix lists all publications during my PhD study. Papers related to this thesis are marked with asterisk.

Book Chapters

1. Zhongwei Zhang, **Zhi Li**, Shan Suthaharan, Fuzzy Logic Strategy of Prognosticating TCP's Timeout and Retransmission, in Computational Intelligence for Modeling and Predictions, Volume 1.

Published Papers

1. ***Zhi Li**, Zhongwei Zhang, and Ron Addie, A Circumspect Active Queue Management Scheme Based on Fuzzy Logic, International Conference on Computational Intelligence for Modelling, Control and Automation (CIMCA) 2004 pp. 169 - 180

2. ***Zhi Li**, Zhongwei Zhang, and Hong Zhou, Quantitative Performance Analysis of Queue Management Schemes in TCP/IP Networks, Australian Telecommunications, Networks and Applications Conference (ATNAC) 2004.

3. ***Zhi Li**, Zhongwei Zhang, Ron Addie, Fabrice Clerot, Improving the Adaptability of AQM algorithms to traffic loads Using Fuzzy Logic, Australian Telecommunications, Networks and Applications Conference (ATNAC) 2003.

4. **Zhi Li** and Zhongwei Zhang, Monitoring the Conformance of Connections to the Traffic Contract of ATM Networks using Fuzzy Logic, in Proceedings of Knowledge-based Intelligent Information Engineering Systems & Allied Technologies, 2002, pp. 194 - 198.

5. **Zhi Li**, Zhongwei Zhang, An Application of Fuzzy Logic to Usage Parameter control in ATM Networks, in Proceedings of the 1st International Conference on Fuzzy Systems and Knowledge Discovery: Computational Intelligence for the E-Age, 2002.

6. Zhongwei Zhang, **Zhi Li**, Shan Suthaharan, Fuzzy Logic Strategy of Prognosticating TCP's Timeout ad Retransmission, in Proceedings of the 1st International Conference on Fuzzy Systems and Knowledge Discovery: Computational Intelligence for the E-Age, 2002.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The size and speed of the Internet have been growing ever since its inception in the 1960s, and so has the complexity of its traffic. With the emergence of optical fibers and microprocessors running at billions of instructions per second, the future Internet would be expected to have high speed with cheap and infinite bandwidth, and no communication delay other than the speed of light. However, this is unlikely at least in the medium term [6]. There are a variety of network topologies, protocols, and traffic patterns. *Traffic control* thus has to be in place for dealing with congested links, and to provide a certain quality of service (QoS) to meet different user requirements and preferences.

Despite its immensity and heterogeneity, the Internet is remarkably stable. This robustness of the Internet can be attributed to the widespread Transmission Control Protocol (TCP) protocol. The first congestion collapse in the mid 1980s was solved by the congestion avoidance mechanisms introduced by Van Jacobson [60]. These mechanisms took the form of a modification of the TCP protocol, and this modification is often referred to as TCP's congestion avoidance strategy. Since then, traffic control has been the focus of researchers in the networking area. The current Internet primarily deploys TCP end-to-end congestion avoidance algorithms to manage traffic and prevent any congestion collapses. It is worth mentioning that the other important transport protocol, UDP (User Datagram Protocol), does not have a congestion avoidance strategy, but it is expected that users will not resend lost UDP packets. The policy, which is not enforced, is also part of the solution of the congestion collapse problem of the Internet.

We will introduce the primary traffic control strategies in the current Internet and related two problems in Section 1.1. We then will further discuss these two problems and briefly describe the status of previous studies into the problems in Section 1.2 and Section 1.3, respectively. The motivations and contributions of this thesis will be presented, followed by the thesis organization.

1.1 Traffic Control in the Internet

A TCP source empirically adjusts its transmission rate according to its packet losses and delays. Routers however, especially gateway routers, potentially have more knowledge of the network situation so they should play an active role to complement the end-to-end TCP control mechanism.

There are two basic classes of router algorithms related to congestion control, including queue management (QM) and scheduling. In a router, QM decides when to drop a packet and how, while scheduling determines which packet to send next and is used primarily to manage the allocation of bandwidth among flows or aggregates of flows. These algorithms are closely related traffic control mechanisms, but they address rather different performance issues [16].

In the current Internet, the combination of TCP and router algorithms such as Drop Tail and FIFO (first in first out) strives to operate robustly and to sustain implementation simplicity. However, some problems have been observed. First of all, Drop Tail is not an effective queuing solution anyway when congestion occurs. The problem is that, under heavy-loaded traffic, Drop Tail most likely causes low link utilization, high packet delay and jitter, and high loss rate. Second, the conservative nature of TCP impacts heavily on short-lived connections, and these connections usually make up the majority of traffic in number but a small fraction in bytes. Short-lived flows will experience packet losses, and this has a dramatic effect on their performance. A direct consequence is that short-lived connections rarely reach their bandwidth share in competition with their long-lived counterparts. To solve the former problem, active queue management (AQM) [16] has been proposed as a suggested replacement for Drop Tail. Meanwhile, preferential treatment for short-lived connections seems an intuitive answer for the latter problem.

1.2 Drop Tail and Active Queue Management

The traditional queue management method, Drop Tail, takes the KISS [1] approach that incoming packets are dropped once the buffer is full. Though Drop Tail is easy to implement and works effectively in lightly loaded traffic conditions, there are some drawbacks with this conventional scheme when the traffic load is heavy.

Firstly, **long packet delay**. In an Internet router, unlike a circuit switch, buffers are necessary in the output port for storing packets due to the unpredictable nature of incoming traffic. Most of the time, packets arrive at routers in bursts. In order to absorb the burstiness of incoming packets, these packets are queued in the buffers when the link capacity is not adequate. Especially in high-speed networks, a gateway is likely to be configured with a correspondingly large maximum buffer to accommodate transient traffic [51]. With heavy traffic however, the advantage of allocating a long buffer no longer holds true. Instead, packets are likely to be delayed in the queue for long periods of time.

Secondly, **low network throughput and utilization**. When the link is overloaded, probably more than one packet in a single window of data will be dropped. It is hard for the TCP source to recover from multiple drops [3, 39]. In this case the stable state, which the congestion avoidance algorithms regard as a safer operating point, is cut down continuously. Also, the retransmission timer is often used to infer the packet losses and exponential retransmit timer backoff is validated. The effective throughput or goodput of flow thus shrinks. Especially with heavy traffic load, it is possible that TCP synchronization (SYN) packets, or the corresponding acknowledgment (ACK) packets, remain lost on the way to their destination. As a consequence, the TCP connections have never had a chance to transport their data packets. The same problem often simultaneously happens to many active flows. This results in a phenomenon of *global synchronization*, where many active flows arise and backoff their transmission rates almost at the same time. Due to the synchronously shrunk transmission rates and retransmission of the undelivered or presumably undelivered packets, the network effective throughput as a whole is dramatically decreased and the network resources are significantly wasted.

[1] Keep it Simple and Stupid

Thirdly, **high packet loss rate and deteriorated network conditions**. A significant time interval passes from the time a buffer overflows until the time when the TCP sources sending packets through the congested link sense the congestion and back off the data rates. During this time interval, there are more packets than what the current stable state allows to be injected into the network. These packets are likely to be dropped as well, simply because there are insufficient network resources to handle the given traffic load. The network congestion becomes more severe.

With regard to these problems, the Internet Engineering Task Force (IETF) recommends that Internet routers should implement some queue management mechanism which has the capability of controlling queue lengths and informing end hosts of incipient congestion. This mechanism is called *active queue management (AQM)*.

Among the available AQM schemes, the first milestone work was random early detection (RED) [51] published in 1993. RED has been proposed by IETF as the default AQM scheme mainly because of the effectiveness of managing queue size and the simplicity in implementation. Thus far, RED has been widely studied via theoretical analysis and simulations. RED is not immune from problems, and it has been conceded that the main problem of RED is *parameter tuning*. That is, with different levels of congestion, different parameter settings are needed. Due to the fact that Internet traffic is ever-changing or dynamic, this parameter tuning issue thus has been a major hurdle to the deployment of RED in reality. Significant efforts have been made in the auto-configuration of RED parameters, such as self-configuring RED [34] and ARED [50]. Meanwhile, the problem of queue management has been addressed from different standpoints. Some have addressed it in an intuitively and empirical way, such as BLUE [41], SRED [77], and adaptive Drop-Tail [23]. Some solutions regard it as an optimization problem such as REM [27, 4, 73] and AVQ [66]. A quite high proportion of work in this area is based on classical control theory such as PI [19] and VS [95]. Replacing Drop Tail management of buffers by a better AQM can be expected to improve link utilization, end-to-end packet delay, jitter, and loss.

Due to the nonlinear nature of TCP dynamics, and network complicity such as varying round-trip time (RTT) and different traffic patterns, and various kinds of TCP variants, Internet traffic control can take advantage of human knowledge. For instance, one modern control method using fuzzy logic (FL) has been brought into the AQM

research area. In our research, we have designed a fuzzy logic controller which integrates expert knowledge into queue management to achieve robust control.

1.3 The vulnerability of short-lived connections

It has been noticed as early as 1995 [97, 81] that most Internet flows are short in duration, but a small number of long flows contribute a large proportion of the traffic load. The short-lived flows are referred to as mice and the long flows as elephants. The problem is that the large number of mice suffer low goodputs and high latency due to the conservative nature of TCP. To avoid congesting the intermediate routers, a TCP connection starts from *slow-start*, in which the initial number of sending packets is at the minimum possible value regardless of what capacity is available in the network. We know that any loss event in the network will be detected by duplicate ACK packets or a retransmission timer. Since the receiving side can create a duplicate ACK only in the case that it receives an out-of-order packet, the duplicate ACK shows the sending side that one packet has been left out the network. Due to modest congestion, the sending side does not need to drastically decrease sending rate and restart slow-start. Moreover, the sender can only decrease sending rate roughly by half. On the other hand, the retransmission timeout informs the sending TCP of the occurrence of heavy congestion. Accordingly, the sending TCP falls to slow-start again. We noticed that the TCP end-to-end congestion avoidance algorithms favor elephants to fast recover from the reduction in sending rate due to packet losses. Since there are not enough packets available to form duplicate ACK packets however, a mice connection waits until its retransmission timer expires and follows slow-start upon the detection of packet losses.

Consequently, the user expectation of quick mice data transfer cannot be met. Although AQM tries to penalize the connections with heavy traffic, it hardly solves the problem. Thus, preferential treatment for mice has been proposed independently in [54] and [7].

1.4 Motivation

In this research, we put our efforts mainly in applying intelligent technology FL for AQM and investigating a plausible treatment in favor of mice and with minimal performance impact on elephants. The motivations of this research are as follows.

- An optimal AQM solution is critical to enhance overall performance of the Internet under congestion.

- AQM is a key component in providing quality of service (QoS) in the Internet. Many multimedia applications, such as audio conferencing and video streaming, have been developed to run over the Internet. However, the lack of guaranteed QoS is inhibiting their wide use. On the other hand, different users have different requirements and preferences. QoS has been brought forward as the next target of the Internet. If the performance of the Internet under congestion can be significantly improved, new services, such as the ones just mentioned, will be much more usable and their development will be greatly facilitated.

- Some well-known AQM schemes are sensitive to traffic load fluctuations in a realistic traffic environment that impact on the AQM's potential for the simultaneous achievement of low delay and high network throughput.

- FL is a well accepted intelligent technology for solving complex nonlinear problems such as optimal control of traffic in the Internet, without the needs of precise and comprehensive information and the mathematical model of objects being controlled. It is challenging to derive a mathematical model to approximate TCP/AQM dynamics and to use the model for traffic control on the basis of classical control theory. FL is much more tolerant to ambiguous information and is effective in conducting robust control by emulating the cognitive processes of a domain expert. Therefore, it has the potential to achieve consistent performance in all kinds of traffic conditions by establishing an FL based AQM controller.

- The mice and elephants phenomenon, in which the majority of flows are short but the majority of bytes are in a small number of large flows, can be deployed to meet user preferences and expectations.

- There are some hurdles in employing the existing preferential treatment for mice in [54, 7]. First, two sets of control parameters in a chosen AQM scheme need to be configured for mice and elephant packets respectively in a FIFO queue. Second, one FIFO queue may still cause the delay of mice packets since some elephant packets may still be in the front of the queue. However, timely packet delivery is more critical to mice than elephants in order to meet user expectations.

- The network context in which the preferential treatment for mice operates also needs to be carefully chosen, to make implementation feasible.

A study of the literature shows that AQM is a challenging and complex problem. We argue that FL, capable of integrating expert knowledge to control a complex non-linear system, is a promising tool to achieve robust control of queue management, and ultimately improvement of Internet performance. On the other hand, the mice and elephants phenomenon in Internet traffic provides us with the possibility of meeting user expectations by differentiating traffic flows according to their behavior.

We restrict our research in TCP/IP networks, where all the flows are TCP or TCP compatible [16]. Note that the TCP congestion avoidance mechanisms are one of the keys to the stability and success of the Internet. TCP has been used by most traffic in the Internet (95% of bytes, 90% of packets, and 80% of flows) [8]. Therefore, it is appropriate to concentrate our attention on TCP when studying congestion control. Also, TCP remains dominant, despite the emergence of other transport protocols. For those unresponsive flows and flows that are responsive but are not TCP-compatible, scheduling algorithms are needed to combine AQM in order to offer fairness among all active connections. We do not address this problem in this dissertation.

1.5 Contributions

The contributions of this research are as follows.

- Conduct a survey of existing AQM schemes. The significance and limitations of AQM are discussed compared to the traditional Drop Tail strategy, which formed the basis of our motivation.

- The use of traffic load factor as a congestion indicator has been shown to be effective in this research. Traffic load factor is defined as the ratio of input rate to target capacity, where target capacity is the leftover link capacity after draining the existing queue in an output buffer. There are some benefits of using traffic load factor as a congestion indicator. First, traffic load factor is dimensionless so that a control algorithm based on such a measure is robust against link capacity changes [21]. Second, the calculation of the traffic load factor can be easily extended to deal with the scenario where best-effort traffic coexists with other QoS traffic with reserved bandwidth, and available capacity for best-effort traffic is ever-changing.

- Design two FL-based AQM algorithms. Both algorithms periodically adjust dropping probability for incoming packets to suit the network situations. The main goals of both algorithms are to maintain low queuing delay and high link utilization. Also, it is observed that the second algorithm improves the stability of the control system.

- Conceptualize the mice and elephants (ME) strategy. More importantly, the ME strategy performs at the most convenient location – the gateway router. It is shown that it is possible to implement an effective control strategy for providing better performance for mice flows without sacrificing that of elephants.

- Integrate the developed FL-based AQM scheme with the ME strategy. The FL-based AQM scheme carries out control on elephant connections only, but with consideration of the capacity consumption of mice.

- Periodically update the databases for mice and elephants in the ME strategy to reduce any unnecessary increase in database size.

- Dynamically adjust the file length threshold in the ME strategy to solve the possibility of elephant starvation on the basis of traffic conditions, using FL.

- All the studies have been conducted via extensive simulations on the platform of the NS2 simulator.

1.6 Thesis Organization

The rest of this dissertation is organized as follows.

Chapter 2 describes the underlying traffic control technologies in the best-effort Internet and other QoS-support network architectures. In the Internet, TCP and Drop Tail queue management with FIFO scheduling so far have been dominantly used in the end user and the intermediate routers, respectively. This combination has contributed to the robustness and implementation simplification of the current Internet. However, to achieve differentiated services and further end-to-end QoS, more sophisticated router mechanisms are necessary to be involved in networks such as ATM, Intserv, and Diffserv. Furthermore, effective queuing is one of the fundamental building blocks for both the Internet and QoS-support networks in terms of the provision of high goodput and low latency in connections and simultaneously high degree network resources utilization. This research has been motivated on the basis of this fact.

Chapter 3 describes the state of the art of queue management. First, Drop Tail and its limitations with heavy-loaded traffic are described. These limitations are heavily related to the reactive manner of Drop Tail. Based on this observation, proactive response to congestion is proposed. This is referred to as active queue management (AQM). One of the first AQM candidates is RED. An in-depth introduction to RED and related issues including its gentle mode, Explicit Congestion Notification (ECN), and the issue of parameter settings, is given. Due to the hurdle of parameter settings in RED, two directions arise in the design of AQM schemes: autotuning RED parameters, and complete new methods from RED. The congestion indicators, control principles, and marking probability of some typical AQM schemes are presented. The traffic modelling used for generating realistic Internet traffic in simulations is also discussed.

In chapter 4, the impact of various AQM schemes is quantitatively studied via simulations under a traffic model of Poisson-Pareto. The necessity of AQM, particularly in the congested links, is shown by the simulation results. In addition, some well-known AQM schemes present sensitivity to traffic load fluctuation in a realistic traffic environment. Also, we are able to observe the way in which short-lived connections are vulnerable in comparison to their long-live counterparts. Solutions to these problems are proposed and studied in Chapter 5 and Chapter 6, respectively.

Chapter 5 presents two FL-based AQM algorithms FLAQM(I) and FLAQM(II). Traffic load factor, which indicates the traffic load conditions on the bottleneck link by taking account of the combination of draining existing queue and accommodating incoming packets, is adopted to offer early congestion notification. FLAQM(I) directly utilizes the traffic load factor and its change as input variables, while in FLAQM(II) the reciprocal of the traffic load factor and the corresponding change are chosen as inputs to implicitly realize input normalization and thus improve control system stability and robustness. The performance of the two proposed FL-based AQM algorithms is evaluated via a wide range of simulation experiments. The simulation scenarios include not only a variety of traffic load but also changed traffic load conditions.

Chapter 6 proposes the preferential treatment approach (ME) that operates in firewall routers on the premise side of links joining premises to the Internet. Such access links are often congested in contrast to well-equipped core networks. The best location for monitoring and controlling this congestion is clearly at edge routers, i.e. the routers on the Internet side of access links. However, edge routers are likely to be busy with data transfer since there are many links to different premises, so we would like to see if the ME strategy can be made to work just as well even when it is executed on the premise side of access links. To implement the ME strategy, we first classify flows as mice or elephants by counting the amount of bytes, and put mice packets into a separate queue that has priority and the remaining packets in an elephant queue. Moreover, FLAQM is deployed to perform elephant buffer management, with consideration of the capacity consumption of mice traffic.

Chapter 7 concludes this dissertation and discusses future work.

Appendix A includes our C++ programs for implementing FLAQM(I) and FLAQM(II) and Tcl programs for testing the performance of them against some well-known AQM schemes.

Chapter 2

Traffic Control and QoS

Since we are now seeing convergence of all services onto the Internet, quality of service (QoS) and traffic control are becoming increasingly important and are a focus for a considerable amount of research [74]. The Internet traditionally provides a simple service – best-effort, with which every packet is equal and networks try their best to deliver packets as soon as possible and as much as possible. With the high-speed development of computer software and hardware, providing more advanced services than best-effort becomes technically possible. Meanwhile, demand from customers makes ISPs keen to bring QoS into the market after the broadband service. Either the best-effort Internet or advanced QoS networks need to be supported by traffic control. Traffic control aims to avoid any congestion collapses and to maintain a healthy environment for data transfer. The two components for realizing traffic control are hosts at the end and routers in the middle. These components are functionally complementary. For the traditional best-effort service, TCP (transmission control protocol) plays a key role to maintain robustness in the Internet by operating end-to-end flow and congestion control at end hosts, while router mechanisms select next the link for delivering packets and buffer them as required. At present, the buffering and delivery is FIFO (first in first out) and when buffers overflow, the packets which arrive while the buffer is full are the ones to be dropped. However, altering the behaviour of routers might be one way to realize advanced QoS.

In this chapter, we first investigate the traffic control strategies offering the best-effort service in the current Internet. Most network users have been and will remain satisfied with the best-effort service most of the time due to its low cost and conve-

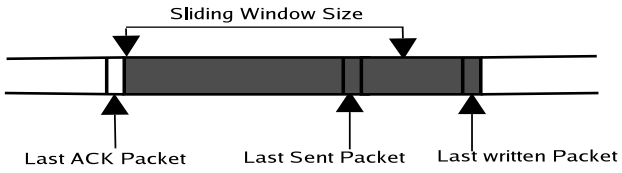
nience. For instance, the quality of applications such as email, Web surfing, and file transfer have been acceptable most times. Best-effort is thus expected to continue as the dominant service for the foreseeable future. However, some businesses and individuals are willing to pay more to obtain more sophisticated services for certain applications or in certain circumstances. The advanced traffic control schemes in some QoS architectures including ATM, Intserv, Diffserv, and best-effort with service differentiation, are therefore needed.

2.1 Best-Effort Service

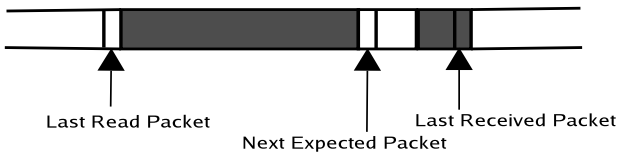
The current Internet offers only the best-effort service with available bandwidth, delay, and loss characteristics dependent on instantaneous traffic level and network conditions. The provision of this service model is caused by the endpoint control design philosophy of the Internet. The idea is that reliable data transfer must be provided by protocols operating at the endpoints, not in the network. Thus, the network can be slow and insensitive to congestion, but intelligence in the endpoints should compensate for this [63]. As a result, end hosts deploy sophisticated TCP end-to-end congestion avoidance algorithms, while routers in the network simply perform Drop Tail queuing, with which the routers accommodate each incoming packet unless the output buffer is full, and serve the packets in the buffer with FIFO order.

2.1.1 TCP end-to-end congestion avoidance algorithms

The congestion avoidance mechanisms of TCP is primarily designed to ensure the stability of the Internet. TCP dominates the Internet by usage, averaging about 95% of the bytes, 90% of the packets, and 80% of the flows over the Internet [8]. TCP also is the only transport protocol that implements congestion control and avoidance mechanisms [40]. The TCP end-to-end congestion avoidance mechanisms were first developped in the late 1980s by Van Jacobson to conquer congestion collapse. TCP uses a concept of sliding window, which restricts the number of bytes in transit and where the window size is adjusted according to network conditions by using acknowledgment (ACK) packets and a retransmission timer. The ACK packets are sent by the TCP receiver to acknowledge the receipt of packets from the sender, and the retransmission



(a) Sending TCP



(b) Receiving TCP

Figure 2.1: TCP window flow control

timer is set by the TCP sender when it sends data. In addition, the arrival of ACK packets has TCP pace its injection of a new window of packets into the network. Figure 2.1 illustrates that the sending TCP keeps records of the last packet acknowledged by the receiver, the last packet sent by the sender, the dynamic sliding window size, and the last packet written by the application running on top of TCP, whereas the receiving TCP maintains the last packet received by the receiver, the next packet expected from the sender, and the last packet read by the application running on top of TCP.

TCP congestion control algorithms consist of *slow start* and *congestion avoidance* for a TCP connection, the use of either of which is decided by two variables: congestion window (*cwnd*) and slow start threshold (*ssthresh*). Variable *cwnd* is a sender-based limit on the amount of data the sender can transmit into the network before receiving an ACK [3]. Variable *ssthresh* reflects a conservative approximation of the window size that the network can support [34]. Another variable, determined by the receiver TCP for limiting the amount of outstanding data, is the receiver's advertised window (*rwnd*). The minimum of *cwnd* and *rwnd*, called window size, governs data transmission.

- **Slow start**: Slow start applies when $cwnd < ssthresh$. Every time a new ACK packet (that acknowledges previously unacknowledged data) is received, *cwnd* is increased by the formula:

$$cwnd+ = 1. \tag{2.1}$$

Variable *cwnd* is virtually doubled every round trip time (RTT) and thereby it is also called the exponential increase phase.

- **Congestion avoidance**: The congestion avoidance phase applies when $cwnd \geq ssthresh$. Every time a new ACK packet is received, *cwnd* is creased by the formula:

$$cwnd+ = 1/cwnd. \tag{2.2}$$

Variable *cwnd* is increased by one packet every round trip time (RTT) and thus it is also called the linear increase phase.

The TCP sender usually counts upon duplicate ACK packets from the receiver to detect network congestion. Upon receiving a certain amount of duplicate ACK packets, TCP infers that a packet is lost on its way due to network congestion and immediately reduces its sending rate roughly in half. TCP first readjusts *ssthresh* to be one half of the current window size (but at least two segments). Then set *cwnd* to *ssthresh* plus the number of segments that have left the network and the receiver has buffered, after retransmitting the lost packet. For each additional duplicated ACK packet received, inflate *cwnd* by one to reflect that another packet has left the network. Upon the arrival of a new ACK packet, *cwnd* is eventually deflated to *ssthresh*. These algorithms are called *fast retransmit* and *fast recovery* [3]. So far, TCP stays in the congestion avoidance phase.

When congestion is severe, both data and ACK packets can get lost. Thus, packet loss cannot be inferred via the above manner. To provide reliability, TCP sets a retransmission timer when it sends data. When a timeout occurs, TCP retransmits the data. Accordingly, *ssthresh* is set to be one half of the current window size and *cwnd* is reduced to one segment [34, 91]. The variable retransmission timeout (RTO) is statistically calculated based on the measurements of the round-trip time (RTT) experience on a given connection. TCP is likely to be silent for a long while due to the inaccurate retransmission timer, before the timer expires [40]. Also, since the data was retransmitted, the exponential RTO backoff algorithm is deployed until a new ACK packet is received [91].

We summarise the TCP algorithms as follows: (1) slow start is only invoked when a TCP initially starts without any knowledge of network conditions or after the retrans-

mission timer expires; (2) Maintaining a TCP connection with a considerable volume in the congestion avoidance phase manifests the stability of the network and desired performance with regard to individual connection throughput and network utilization; (3) In addition, TCP uses an additive-increase multiplicative-decrease (AIMD) policy within the congestion avoidance phase in that AIMD is a relatively moderate approach to adjust its sliding window size.

In order to enhance network performance and provide fast and accurate response to the changes in the network, considerable efforts have gone into improving TCP. Following the first version of TCP, Tahoe TCP in 1988, Reno TCP was implemented in 1990 [39] to fast-recover connection throughput from moderate congestion by performing congestion avoidance after receiving a certain amount of duplicate packets instead of conducting slow start. Note that the previous description of TCP congestion control algorithms is consistent with Reno TCP. Subsequently, NewReno TCP, and SACK TCP were made available for fast recovery from multiple packets loss in a single window to overcome the drawback of Reno TCP. In the current Internet, Reno TCP coexists with NewReno TCP and SACK TCP, although it was the dominant TCP implementation in earlier times.

The maneuver to improve the robustness of TCP has resulted in sophistication in congestion control merely at end hosts. However, there are some disadvantages with the combination of sophisticated TCP and simple router mechanisms including Drop Tail queuing and FIFO scheduling.

- There exist biases or unfairness in the Internet. Firstly, the Internet has a bias against TCP connections. Based on the concepts for flow classes in [16], except "TCP-compatible" flows that behave under congestion like a flow produced by a conformant TCP, there are other flow categories that do not slow down when congestion occurs or are not TCP-compatible. These more aggressive flows could lead to all the active TCP connections backing off their sending rates. Secondly, TCP/IP networks have a bias against connections with longer RTT. Thirdly, TCP/IP networks have a bias against connections passing through multiple congested gateways [48]. Finally, TCP/IP networks have a bias against short transfers, which contain the majority of Internet traffic. These short flows almost always stay in the slow start phase and seldom get a chance to obtain their

share of bandwidth. Detailed study on this final issue will be given in Chapter 6.

- Network capacity is underutilized. A TCP source reduces its transmission rates only after detecting packet losses caused by queue overflow. There is a considerable time interval between the two events: packet drop in the congested router and congestion detection in the end host. During the time interval, the TCP source continues emitting packets at an unaffordable rate to the congested router. Those packets are most likely dropped at the router [34]. The same fate may be shared by many TCPs simultaneously. This eventually leads to global synchronization, in which many TCP sources cut down their sending rates at the same time. Moreover, TCPs often use a timeout mechanism to recover from multiple packet drops in a single data window. Significant time may be experienced before the TCP retransmission timer expires. Therefore, the network capacity is squandered for the transfer of those packets that will be dropped in the downstream and in the subsequently long idle period.

- Performance is unpredictable. TCP views the network as a black box, which it probes by increasing the transmission-window size and looking for packet losses [63]. TCP has no knowledge about the network situation such as the number of its competitors for the capacity and the queue length of the congested routers. It only estimates the available bandwidth in the network by using the AIMD strategy. The individual connection performance in terms of TCP goodput and delay thus is unpredictable.

Consequently, to support more advance services, router congestion control mechanisms need to be modified by making use of routers' potentials of gathering information about network conditions and taking action on even the finest grained individual flows instead of flow aggregations.

2.2 Advanced QoS support

The best-effort mode of transport in the current Internet no longer meets all needs and requirements. For instance, multimedia applications such as video/audio conferencing, interactive games, digital libraries, and distance education, require large bandwidth

and stringent delay. Recently, many ISPs have attempted to offer voice over IP (VoIP) and video streaming to their customers, which require certain QoS technologies to be in place to ensure timely delivery. Therefore, one can define QoS as a set of technologies that enable optimal management of network resources even under congested conditions, and guarantee different performance requirements such as loss, delay, and delay variation (jitter).

Among the existing technologies for supporting advanced QoS, ATM (asynchronous transfer mode) was the first networking technology designed with QoS in mind from the outset [74]. Because of the popularity of the Internet in practice, ATM technology has mostly been adopted in high-speed backbone networks and access niches. This way, the ATM network clouds are merely treated as a high speed element in the Internet. Despite this, ATM has significant impact on the design of other QoS architectures. During the latter half of the 1990s, considerable efforts have been made to add QoS capabilities to the Internet by the Internet Engineering Task Force (IETF). Two wide studied IETF QoS architectures are Integrated Services (Intserv) and Differentiated Services (Diffserv).

2.2.1 ATM

ATM technology has been primarily chosen by the International Telecommunication Union (ITU) for implementing its Broadband Integrated Services Digital Networks (B-ISDN). ATM networks were designed to subsume both the Internet and the telephone network to create a unified infrastructure that carries voice, video, and data with the great benefit of statistical multiplexing. The idea was to combine the flexibility of the Internet with the per-user QoS guarantees of the telephone network.

There are some particular features of ATM. Firstly, ATM is connection-oriented. Before transmitting data, every source applies for a connection request to the network through signaling procedures. The request includes a chosen service type, traffic parameters, and the expected QoS. A connection will be established only when the available capacity in the network meets the demand. Otherwise, the connection will be rejected. Once the network accepts the connection, a traffic contract is formed which records the value of each parameter for traffic specification and desired QoS. The contract needs to be obeyed by both the network and the user.

Secondly, ATM uses small fixed-size packets or cells. The small size helps to give a low delay guarantee to real-time applications such as voice, while the fixed size allows the design of a simple transmission network with low variance in delay [55]. An ATM cell is 53 bytes long including a 5-byte ATM header and a 48-byte payload. In ATM networks, tasks of switching and forwarding cells are conducted by ATM switches, which are the counterpart of routers in IP networks. To find the next hop, an ATM switch uses the virtual path identifier/virtual circuit identifier (VPI/VCI) in the ATM header of an incoming cell, whereas an IP router analyzes the destination IP address of an arrival packet [63]. A VPI/VCI is shorter than an IP address. Also, it is easier to get a match in a routing table for a VPI/VCI than an IP address. Thus, ATM networks have enormous switching capacity. In addition, as in IP networks, ATM allows statistical multiplexing so that network resources can be highly utilized.

There are some ATM standards proposed by two standardization bodies including the ITU Telecommunication Standardization Sector (ITU-T) and the ATM Forum. The former targets global standardization issues involved in telecommunication. ATM is defined by the ITU-T as a component of B-ISDN, and various aspects of B-ISDN are covered by the ITU-T recommendations within Series I. The ITU-T allows subscribers unlimited download of the standards, or purchase of individual documents. Due to the slow standardization process of ATM in the ITU-T, the ATM Forum was formed jointly by a group of computer and communication industries, which saw ATM as a promising technology for high-speed data communications, in 1991. The ATM Forum is now a truly international forum whose members span the industry, government, and education sectors and many countries worldwide. Standardized documentation in the forum, or ATM Forum specifications, can be downloaded free of charge. The following ATM service categories and generic QoS mechanisms in ATM networks are referred to in the traffic management specification of the ATM Forum in [10].

2.2.1.1 ATM Service Categories

The ATM Forum defines a range of service categories employing traffic characteristics and QoS requirements. This multi-service model is based on the fact that ATM networks are designed to be able to transfer many different types of traffic simultaneously, including real-time flows such as voice and video, and bursty TCP flows [90]. The

ATM service categories are defined as follows.

- CBR (Constant Bit Rate): The CBR service is intended for real-time applications that require a fixed data rate that is continuously available during the connection lifetime, and a tightly constrained delay variation. Examples of CBR applications include videoconferencing, distance learning, and video-on-demand.

- rt-VBR (Real-time Variable Bit Rate): The rt-VBR service is intended for real-time applications that transmit at a rate varying with time. The rt-VBR source thus can be characterized as somewhat bursty. The rt-VBR category requires tightly constrained delay and delay variation. This service, unlike CBR, may benefit from statistical multiplexing. Examples of rt-VBR applications include videoconferencing, distance learning, and video-on-demand.

- nrt-VBR (Non-real-time Variable Bit Rate): The nrt-VBR service is intended for non-real-time applications that characterize the expected traffic flow so that the network can provide relatively low delay and minimal cell loss. Within this service, the end system specifies a peak cell rate, a sustainable or average cell rate, and a measure of how bursty or clumped the cells may be. Examples of nrt-VBR applications include airline reservations, banking transactions, and process monitoring.

- UBR (Unspecified Bit Rate): The UBR service is intended for non-real-time applications that can tolerate variable delays and some cell losses, that is, TCP-like traffic. There is no QoS guarantee for UBR, and thus the service is referred to as a best-effort service. Examples of UBR application include text/data/image transfer.

- ABR (Available Bit Rate): The ABR service is intended for non-real-time applications that specify a peak cell rate that it will use and a minimum cell rate that it requires, which can be zero. One distinctive feature of ABR is that ABR employs explicit feedback to sources to ensure that the leftover capacity from CBR and VBR minus bandwidth reserved for ABR is allocated among all ABR sources in a fair manner. Examples of ABR applications include text/data/image transfer.

- GFR (Guaranteed Frame Rate): The GFR service is intended for non-real-time applications that require a guarantee on the basis of immediate upper layer data units or frames, and specify a peak cell rate and a minimum rate guarantee along with a maximum burst size and a maximum frame size. GFR can benefit from accessing additional bandwidth dynamically available in the network. GFR is designed to transfer TCP/IP packets over ATM networks. Examples of GFR applications include text/data/image transfer.

Table 2.1 summarizes the ATM Forum service categories and the corresponding QoS guarantees.

Table 2.1: ATM service categories and QoS guarantees

QoS/Services	CBR	rt-VBR	nrt-VBR	ABR	UBR	GFR
Loss	Yes	Yes	Yes	Yes/No	No	Yes/No
Delay	Yes	Yes	No	No	No	No
Delay variance	Yes	Yes	No	No	No	No
Bandwidth	Yes	Yes	Yes	Yes/No	No	Yes/No

Note that Yes/No stands for Yes or No.

2.2.1.2 Generic QoS Mechanisms in ATM networks

For each service category, traffic management and control functions are, in general, structured differently. There are however some generic functions defined in [10] with the aims of protecting the network and the end-system from congestion and promoting the efficient use of network resources.

- CAC (Connection Admission Control): CAC is a set of actions taken by the network during the call set-up phase in order to determine whether a connection request should be accepted or rejected (or whether a request for re-allocation can accommodated). It is the first line of defense for the network to avoid excessive loads. The connection request includes, implicitly or explicitly, the chosen service category, traffic description, and the requested and acceptable values of QoS parameters such as bandwidth, delay, and jitter.

- Feedback controls: Feedback controls are a set of actions taken by the network and by end-systems to regulate the traffic submitted on ATM connections according to the state of network elements. One instance of feedback controls is the ABR flow control mentioned in the ABR service category.

- UPC (Usage Parameter Control): UPC is a set of actions taken by the network to monitor traffic and enforce the traffic contract at the user network. Correspondingly, at the network node interface, Network Parameter Control (NPC) is performed. Both UPC and NPC aim to protect network resources from malicious as well as unintentional misbehavior which can affect the QoS of other already established connections, by detecting violations of negotiated parameters and taking appropriate actions including cell discard or cell tagging.

- Cell Loss Priority control: For some service categories, the end host may mark the cell loss priority (CLP) bit in the ATM header in some cells, which are extra (beyond the negotiated rate). The network also may set the CLP bit for any data cell that is in violation of an agreement concerning traffic parameters between the user and the network. If congestion is encountered, the ATM switches that recognize the marking may selectively discard CLP-marked cells with a low priority to protect, as far as possible, the QoS objectives of cells with high priority. Alternatively, the CLP marking may be treated transparently.

- Traffic Shaping: Traffic shaping is used to smooth out a traffic flow and to reduce cell clumping and/or to ensure connection traffic conformance at a subsequent interface.

- Network Resource Management: Although cell scheduling and resource provisioning are implementation-dependent and network-specific, they can be utilized to provide appropriate isolation and access to resources. The maximum benefit can be obtained from statistical multiplexing of flows that belong to different service categories.

- Frame Discard: A congested network that discard a cell may discard the frame [1]

[1]The term "frame" here is being used to refer to a generic higher level protocol unit, such as an IP datagram.

that the cell belongs to. Otherwise, the frame will be eventually discarded by the receiver since the receiver recognizes frames as a data unit and conducts error check on frames.

ATM provides a set of reliable methods to guarantee end-to-end QoS. Sender-initiated signaling is performed before any data transmission. After being accepted by the CAC in each ATM switch on the path, the connection is established with a confirmation message sent back to the sender and the flow state related to traffic pattern, requested QoS, its path, and reserved capacity, maintained in these switches. The reliable signaling procedures and the provisioning of capacity to any accepted connections are the first-step contributions to the success of ATM in the handling of QoS. Subsequently, ATM conducts traffic management and control including classification, monitoring, policing, queuing, scheduling, and shaping by using the generic functions mentioned above, on the basis of service models towards data delivery with end-to-end QoS guarantees.

The scalability issue is critical for ATM networks, since ATM operates on a per-flow and per-hop basis. When millions of flows share a large trunk, processing of each flow is simply not practical. To alleviate the problem, ATM uses virtual paths (VP) to aggregate flows and enhance efficiency [74].

2.2.2 Intserv

Integrated Services (Intserv) is an IETF effort aimed at providing end-to-end QoS in IP networks. It has been motivated by two requirements. The first is to support real-time applications, such as remote video and multimedia conferencing, with loss and end-to-end packet delay guarantees. The other requirement is from network operators, who want to be able to control the sharing of bandwidth on a particular link among different traffic classes [6]. To address the two issues and provide integrated services in the Internet, the Intserv working group in IETF specified two more services other than the best-effort service, including the *guaranteed QoS* and the *controlled-load service*. Like ATM, Intserv requires a signaling protocol to transport traffic management and control parameters and achieve a guaranteed reservation. Resource Reservation Protocol (RSVP) was chosen by the Intserv working group as it is the most widely

known example of such a setup mechanism, although RSVP and Intserv were designed independently. Also, the Intserv working group has proposed some possible network traffic control mechanisms in [6]. We will give brief introductions to these in the next subsections.

2.2.2.1 Intserv Service Model

The guaranteed QoS service and the controlled-load service are defined in Intserv.

- guaranteed QoS: This service is intended for delay-sensitive real-time applications that require reserved bandwidth and firm bounds on end-to-end packet delay. The traffic is characterized in the form of a token bucket (r, b) with a bucket depth b and a bucket rate r, a minimum counted unit (m), and a maximum packet size (M). The end-to-end delay bound is computed based on the fluid model. Interestingly, the guaranteed QoS equals rt-VBR in ATM, but is supported in the Internet. Examples of this service include audio and video play-back application.

- controlled-load service: This service is intended for adaptive real-time applications that require unloaded best-effort service. Although this definition may seem somewhat vague, it is not. This notion is rather precisely defined by using queuing theory in terms of the likelihood of packet delivery and statistics regarding the delay incurred [74]. The traffic is specified in the same way as guaranteed QoS.

RFC 2211 and RFC 2212 also require that the packets, which fail to conform to the traffic pattern specified during setup phase, are treated on a best-effort basis.

2.2.2.2 RSVP

RSVP is the default option of Intserv signaling protocols that can be used by hosts to request resource reservations through a network. It has the following attributes compared to the ATM signaling protocol [26, 74].

- Receiver-initiated. The receiver initiates and maintains the resource reservation used for the flow. This feature is specially designed for accommodating multicast and also suitable for the trivial case unicast. In contrast, ATM employs sender-initiated signaling.

- Unidirectional. RSVP signals independently for each direction of a flow, whereas an ATM sender may signal for both directions of a full-duplex connection.

- Soft state. RSVP maintains "soft" state in routers and hosts, to provide graceful support for dynamic membership changes and automatic adaptation to routing changes. RSVP thus requires that the receiver periodically refreshes reservation requests. Otherwise, any states will automatically expire after some time and reservations are automatically torn down. On the other hand, in the case of ATM, the hard state is kept in the switches along the end-to-end path until the termination of the connection or the occurrence of a network failure. With this consistently reserved capacity in all the nodes on the path, ATM provides a more predictable network.

The RSVP model better supports the native IP interface present in many applications [74].

2.2.2.3 Intserv Traffic Control Mechanisms

To support end-to-end QoS delivery in IP networks, network elements first apply Intserv admission control to signaled resource requests. Traffic control mechanisms on the network elements then are configured to ensure that each admitted flow receives the service requested in strict isolation from other traffic [6].

- Packet scheduling: Scheduling primarily allocates capacity among flows by reordering the output queue. The approaches range from the simplest priority scheme to weighted round-robin (*wrr*) and weighted fair queuing (*wfq*). On the contrary, the current IP network performs FIFO scheduling, in which no reordering actions are taken.

- Packet queuing: Queuing is the set of actions taken by the network on incoming packets, including accommodating and dropping. Since dropping a packet is taken by TCP as a signal of congestion and causes it to reduce its load on the network, picking a packet to drop is the same as picking a source to throttle. Also, if a queue builds up, dropping one packet reduces the delay of all the packets behind it in the queue. To support integrated services including real-time applications, therefore, an intelligent queuing mechanism is pivotal, which

does not have to rely on buffer overflow as the only indication of congestion. In addition, packet drops mean the waste of upstream bandwidth and sometimes packet delay, and thus passing the congestion notification to sources by marking an ECN field in the IP header instead of dropping is recommended in [86, 84].

- Packet classification: Packet classification is the set of actions taken by the network to sort out packets into some flow or sequence of packets that should be treated in a specified way. Classification can be done simply based on IP layer information such as source and/or destination addresses. Alternatively, packets could be classified in a more complex way according to flow identification consisting of not only source and destination addresses and protocol type, but also some transport layer information including source and destination port number. There is a trade-off between processing overhead and implementation efficiency.

- Admission control: Admission control is the set of actions taken by the network to make a decision about resource availability for a new service request. It is operated by each router on the end-to-end path of data delivery during the RSVP signaling procedures. Different admission control strategies may be applied to different QoS requirements. A computation is based on the worst-case bounds for admitting new flows in some cases, while in the others measured information is used.

Like ATM, scalability is still an issue for Intserv, because of its per-flow and per-hop operation. As stated in the RSVP RFC, some form of aggregation is essential to build a scalable network [74].

2.2.3 Diffserv

Differentiated Services (Diffserv) is the latest IETF effort for QoS-enabled IP networks. Although Intserv has been standardized, its scalability limitation has been a major hurdle for its wide deployment. To meet the immediate need of QoS delivery for differentiated services, the Diffserv working group was formed in 1998. In addition, Diffserv is expected to permit differentiated pricing of Internet service [29].

Due to the continuous growth in the number of users, the variety of applications, and the capacity of the network infrastructure, Diffserv was developed with scalability in

mind. Firstly, Diffserv networks group packets into one of a small number of aggregated flows based on the Diffserv Codepoint (DSCP) value in the Diffserv field (DS field) in the IP header. This is known as behavior aggregate (BA) classification, where the DS field consists of the six most significant bits of the IPV4 Type of Service (TOS) octet or the IPV6 Traffic Class octet [53]. For each aggregation, a selected per-hop behavior (PHB) or a PHB group is applied to forward the traffic by using a particular scheduling treatment and in some cases packet dropping strategy. Secondly, Diffserv carries out complex traffic classification and conditioning functions including metering, marking, shaping and policing, all only at network boundary nodes, i.e. ingress and egress routers. Meanwhile, simple packet classification and forwarding functions are located in the core of the network to cope with a large amount of active traffic flows at coarse granularity. The ingress and egress nodes of Diffserv networks may deploy a multi-field (MF) classifier to examine the content of some arbitrary number of header fields. For instance, 5-tuple (source and destination address, source and destination port, and protocol type) is used for classification. In a more complicated case, information about the incoming interface is also needed. Note that ATM and Intserv all have their own solutions to the scalability issue by coping with traffic at aggregate levels. However, this improved granularity comes at the cost of additional management and configuration requirements. Additionally, the amount of forwarding state maintained at each node in an ATM or an Intserv network scales in proportion to the number of edge nodes of the network, even in the best case.

In addition to the scalability feature, the Diffserv model has the following features compared to ATM and Intserv studied previously [29].

- With no consideration of applying any signaling protocols, Diffserv uses static resource provisioning based on a service level agreement (SLA) between a customer and an ISP or between two adjoining ISPs. Diffserv attempts to work with existing applications without the need for making them signaling-compatible. By contrast, the explicit and dynamic admission control and resource reservation in ATM and Intserv help to assure that a specified QoS guarantee is achieved and at the same time network resources are optimally used [35]. However, ATM and Intserv have difficulty catering for certain services because those services do not know how to provide the information used in the ATM and Intserv signaling.

- Service provisioning and traffic conditioning policies are sufficiently decoupled from the forwarding behaviors within the network interior. This way, implementation of a wide variety of service behaviors is permitted with possibilities for future expansion.

- Supported services are decoupled from the particular applications in use. Diffserv is able to provide a service that avoids assumptions about the type of traffic using it. Thus, Diffserv caters for a variety of users with different service preferences and expectations.

In a Diffserv network, the combination of more sophisticated functions at the edge, and a number of PHBs and PHB groups at the core, works towards resource allocation and service differentiation.

2.2.3.1 Basic Functions in Boundary Nodes

The basic functionalities of boundary nodes of Diffserv networks are packet classification and traffic conditioning operated based on SLAs. Upon the arrival of a packet a MF classifier, or at trust boundary a BA classifier, is used to search for a matched entry associated with a SLA for the packet. The packet is subsequently steered to be further processed by one or more of the traffic conditioning functions based on this SLA, such as metering, marking, policing, or shaping. If the search fails, the packet will be treated with best-effort and marked with a corresponding DSCP value in the DS field.

- Metering: The process of measuring the properties of a traffic stream against its traffic profile, which characterizes the traffic as specified in the SLA.

- Marking: The process of setting the DSCP value for packets on the basis of the SLA and the metering results.

- Policing: The process of discarding packets within a traffic stream in accordance with the state of a corresponding meter enforcing the specified traffic profile. Policing is often operated in ingress nodes.

- Shaping: The process of delaying packets within a traffic stream to cause it to conform to the defined traffic description. Shaping is often operated in egress nodes.

All these four functions are not compulsory to condition a particular traffic stream. For instance, policing can be omitted, given marking is done differently for the packets conforming to the specified traffic description and the packets not conforming to it.

The boundary functions in Diffserv can be mirrored to their counterparts in ATM and Intserv with the one exception of the marking function. In fact, ATM also accommodates the marking function for setting the CLP bit, while in Intserv and TCP/IP it is proposed to mark the ECN field in the IP packet header to inform sources of incipient congestion. However, marking the DS field with an appropriate DSCP value is more critical for Diffserv, since it is the DSCP value on which the core network bases its conducts of service differentiation. Any misconfiguration of the DS field could make the Diffserv network misbehave. As a matter of fact, interior nodes select a forwarding behavior for packets based on their DSCPs, mapping that value to one of the supported PHBs using either the recommended standards or a local policy [29]. The supported PHBs in the core network have no knowledge about SLAs. Thus far, the Assured Forwarding (AF) PHB group and the Expedited Forwarding (EF) PHB have been standardized by the Diffserv working group with best-effort (BE) as a default PHB.

2.2.3.2 EF and AF

The EF PHB and the AF PHB group are described as follows.

- EF: EF provides a building block for low loss, low delay, and low jitter services. To ensure that queues encountered by EF packets are usually short, it is necessary to ensure that the service rate of EF packets on a given output interface exceeds their arrival rate at that interface over long and short time intervals, independent of the load of other (non-EF) traffic. The standard DSCP of EF is 101110.

- AF: AF provides assured forwarding of IP packets with an expected transmission capacity. While an AF user may transmit more traffic than the subscribed

information rate, it is with the understanding that the excess traffic may not be delivered with as high a probability as the traffic that is within the rate. To this end, a set of PHBs are specified and implemented simultaneously as a group with different levels of drop precedence. Currently, three PHBs comprise one AF PHB group, and each PHB corresponds to a certain drop precedence level and uses a certain DSCP value.

The Internet community has been enthusiastic about Diffserv due to its scalability feature. However, there are some limitations in Diffserv for provisioning end-to-end QoS guarantees. Diffserv deploys static resource reservation with no need of any signaling procedures or dynamic admission control [36]. Even with the assumption of adequate, or even over-provisioning, higher-quality guarantees may still not be ensured. For instance, with resources reserved for handling 10 IP telephony sessions on one path, the join of the 11th session may degrade the performance of all the previous existing sessions. On the other hand, a PHB or a PHB group only defines the behavior of a single node. Also, each router makes its own decision about the next hop for a packet due to the decentralized feature of IP networks. Thus, no QoS is guaranteed in a single Diffserv administrative network, let alone end-to-end QoS guarantees.

2.2.3.3 Improvement of QoS Handling in Diffserv

There are some technologies that can be utilized individually or jointly to alleviate the limitations of the capability of Diffserv for QoS handling. Among them, one can use dynamic provisioning and topology-aware admission control for some services with high quantitative QoS requirements. Dynamic admission control in the network can be evoked upon the receipt of a service request. The request can be passed in a number of ways including using the semantics of RSVP, SNMP, or directly set by a network administrator in some other way. The agent carrying out admission control is a centralized oracle of the Diffserv network, the bandwidth broker (BB) as proposed in [76]. The local BB has sufficient knowledge of resource availability and network topology. If there are adequate resources in the network, the request is accepted. Otherwise, there are two possibilities. If network resources are statically allocated, the request will be rejected. Alternatively, in the case of dynamic allocation, the SLA will be renegotiated based on the request and the final decision will be made accordingly.

Another direction is to combine Diffserv with MPLS. MPLS stands for Multiprotocol Label Switching Architecture. MPLS is named after its capabilities of supporting multiple network layer protocols and also operating over virtually any link layer protocol. It primarily is a label-switching protocol and, as with other label switching technologies such as ATM, it is capable of higher performance switching. At the edge of an MPLS-capable network, the ingress node assigns a particular packet to a particular FEC (Forwarding Equivalence Class). For each FEC, there is a fixed path for forwarding its traffic by using labels, and resources can be allocated along the way. MPLS supports explicit routing for traffic engineering and aggregation by using the concept of FECs in order to be scalable [88]. Moreover, RFC 3270 has specified a solution for supporting the Diffserv behavior aggregates whose corresponding PHBs are defined over an MPLS network [20], by providing mappings to ensure that packets marked with various DSCPs receive the appropriate QoS treatment at each router in the network [11].

Note that the above approaches increase complexity only in the control plane of the network, whereas the scalability and simplicity remain in the data-plane, since traffic is forwarded based on DSCPs and the corresponding PHBs.

2.2.4 Best-effort with Service Differentiation

It needs time to achieve agreement on the deployment of Diffserv for the accomplishment of end-to-end QoS delivery among ISPs, despite available technologies. Several transient architectures offering best-effort service but with service differentiation have been proposed before the implementation of any end-to-end QoS infrastructure. Instead of endeavoring to provide end-to-end QoS, these approaches offer different service preferences and meet user expectations without guarantee. This way, some complex functions such as admission control and resource reservation can be omitted. Besides, there are no requirements for their implementation in all the network elements transversed by a connection. With only the operational necessity of some sophisticated buffer management and scheduling schemes, their deployment is expected to be immediate. Three of them are introduced below in turn.

- Proportional Differentiated Services. The Proportional Differentiated Services ar-

chitecture aims to control the ratios of some basic performance measures among all the traffic classes based on certain class differentiation parameters specified by network operators. This way, traffic performance can be predictable and controllable. When average loss rate and average queuing delay are selected as such measures respectively, two different models have been proposed in [13] and [12] in sequence. In the proportional delay differentiation model, three scheduling schemes were proposed and evaluated, while two dropping mechanisms were options for the proportional loss rate differentiation model.

- Application Characteristics Based Service Differentiation. Both the Best-Effort Differentiated Services (BEDS) model [43] and the Asymmetric Best-Effort (ABE) model [59] tend to differentiate traffic services based on the nature of traffic, either loss-sensitive or delay-sensitive. To this end, they combine a certain queuing strategy and scheduling mechanism, with a difference in the implementation details and an additional assumption of traffic being TCP-friendly conformant in ABE. Intuitively, in the over-loaded network conditions, more packets will be dropped from delay-sensitive traffic than from loss-sensitive traffic, while the scheduler picks up the accepted delay-sensitive packets more quickly than the loss-sensitive ones.

- Flow Length Based Service Differentiation. It is well observed that Internet traffic is composed of a large number of short-lived connections (or mice) and a small number of long-lived connections (or elephants). The large number of mice only contribute to a small fraction of the total traffic bytes, whereas the larger amount of traffic comes from elephants. It is known as the mice and elephants phenomenon. Despite the user expectation that shorter flows should fly through the network with minimum packet loss and delay compared with their longer counterparts, mice rarely obtain their bandwidth share in competition with elephants due to the conservative nature of TCP. An infrastructure was established in two simultaneous and independent works [54] and [7] on the basis of this mice and elephants phenomenon, through forwarding treatment being biased in favor of mice. Flows are first classified to be either a mouse or an elephant, based on its length and intensity, and marking them as IN packets

and OUT packets, respectively. Then packets are served in a RIO (RED with IN/OUT packets) queue with preferential treatment for packets from mice.

2.3 Summary

In this chapter, we have given an overview of the current Internet and some QoS schemes including ATM, Intserv, Diffserv, and best-effort with service differentiation, with regard of their supported services and especially the traffic control mechanisms for realizing these services. It appears that ATM technology has the most comprehensive and reliable features to support the integrated services of voice, video, and bursty data. Due to the widespread use of IP networks, ATM clouds have been used as a high speed link in backbones and more and more access network infrastructures, where they find a nice niche due to their subtle way of managing a small number of flows in almost static routing architectures. Intserv has been designed for the Internet in a similar way to ATM in terms of handling QoS, except that Intserv has to deal with variable packet size and maintain soft states in each node.

Both ATM and Intserv have considered QoS management at an aggregate level, however some limitations exist. As a result, Diffserv has become the technology of choice overwhelming in the Internet community. To date, Diffserv combined with some new technologies, particularly MPLS, has been an active research area and is expected to yield a scalable QoS architecture. There are also some service differentiation proposals for best-effort Diffserv prior to the implementation of Diffserv with the compromise of QoS. More than one QoS architecture has been proposed, discussed, and advocated. It is deemed that they will coexist for the near future both in academic and realistic environments.

One of the underlying traffic control mechanisms in the best-effort service model and the aforementioned QoS architectures is queuing. In the case of requiring low delay and low loss, a short or even empty queue is expected by allocating adequate bandwidth and policing or shaping against a corresponding traffic profile. Queuing thus mainly performs on traffic with non-stringent QoS requirements. In ATM, since ABR employs explicit feedback for sources adjusting their transmission rate, UBR and GFR queue management is necessary. Besides the best-effort service, the controlled service

in Intserv and AF in Diffserv also deploy a queuing strategy. The implementation of any queuing mechanism is ISP- or vendor-specific. In the current Internet, Drop Tail queuing is used, but poor performance has been observed with heavy load traffic. Thus, dropping packets before buffers overflow is proposed, and such a proactive approach is called active queue management (AQM) [16]. Especially, the combination of TCP and AQM is one of the main candidates for QoS deployment in any TCP-based environment. In the next chapter, existing AQM schemes and the significance of AQM will be studied in details.

Chapter 3

Queue Management and Internet Traffic Modelling

Although the TCP end-to-end congestion control algorithms reviewed in the previous chapter have played an important role in preventing congestion collapse in the Internet, there is a widespread belief that it is necessary to supplement the end-to-end TCP control mechanisms by proactive control mechanisms inside the network in order to achieve network efficiency and QoS guarantees. It is a very active topic of research to extend the current Drop Tail queue management scheme in routers. Random Early Detection (RED) was one of the first queuing schemes which realizes active queue management (AQM). RED has been extensively studied through simulations, experiments, and theoretical analysis. Despite support from the IETF, the main difficulty of RED is the configuration of its control parameters.

In this chapter, we first describe Drop Tail and its limitations with heavy-loaded traffic. Then we provide an in-depth introduction to RED and related issues including its gentle mode, Explicit Congestion Notification (ECN), and the issue of parameter settings in RED. RED variants and other AQM schemes, arising on the basis of various theories in such as statistics and closed-loop control to seek better solutions in queue management other than RED, are then discussed, followed by a survey of some typical AQM schemes. The survey will look into these AQM schemes from three different points of view including congestion indicators, control principles, and marking probability calculations. Finally, we discuss the traffic modelling used for generating realistic Internet traffic in simulations.

3.1 Drop Tail

Internet traffic inputs to routers in bursts with idle periods in between, over a wide range of time scales [82, 33]. This traffic behavior is characterized as *self-similarity*. The burstiness is mainly contributed by the TCP congestion control mechanism [78, 42], as well as heavy-tailed distribution of file size [93] and compressed ACK [98]. Due to this salient feature of the Internet, therefore, it is essential for routers to deploy output queues to buffer bursty traffic and forward the excessive packets during the idle intervals.

Most operational routers currently use Drop Tail queuing coupled with FIFO (first in first out) scheduling. In Drop Tail, all packets are accommodated in a buffer until it is full. Packet losses from a Drop Tail queue will usually be detected by a corresponding TCP via duplicated ACK packets or its retransmission timer. The TCP then takes this signal as a congestion indication and reduces its sending rate in accordance with the TCP congestion avoidance algorithm, i.e. the algorithm which sets *cwnd* which was described on Page 13 to mitigate congestion in the network. Therefore, in general, a queuing strategy can be viewed as a queue management (QM) controller, and TCP/QM can be discussed in the framework of feedback control systems [19, 87]. The control object is the traffic plant representing the TCP/QM system. Within the control-loop depicted in Figure 3.1, the QM controller makes decisions on dropping probability of arrival packets based on the feedback from the traffic plant and then sends a control signal to the traffic plant informing it of the dropping probability. Note that valid feedback from the plant is delayed at least one round trip time (RTT) for each connection. The QM controller based on Drop Tail actually conducts on-off control in that dropping probability is set to either 0 or 1. Not only is it easy to implement Drop Tail queuing in routers but also Drop Tail works well with TCP in over-provision networks.

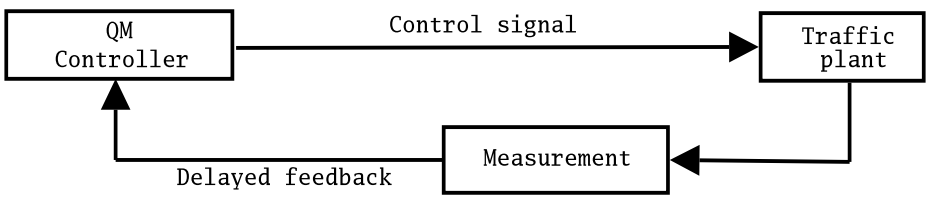


Figure 3.1: Block diagram of a generic queue management control system

However, Drop Tail queue management leads to oscillations that can exhibit complex and chaotic behavior [18]. There are some consequences of Drop Tail queuing as follows.

- Global synchronization. With Drop Tail, loss events in a router tend to involve many packets due to the burstiness of packet arrivals. Since some of the dropping signals may be delayed due to longer RTTs, the corresponding TCP connections can transmit even more data before the cut-off response. This will worsen the congestion, and result in more packet losses and thus more connections involved. By then, many TCP connections decrease their sending rate in synchrony. In the worst case, they have to wait for retransmission timeout and enter slow start to recover. This leads to severe link underutilization.

- Full queues with a long period of time. Large buffer capacity is needed especially in a bottleneck point to absorb bursty traffic. In an over-loaded network, however, large buffer capacity only contributes to longer delay. Also, on the basis of queuing analysis, queue length distribution decays much more slowly for Internet traffic [78].

- Unfairness. With Drop Tail, incoming packets are discarded whenever the buffer is full. The arbitrary selection of connections to throttle back results in penalizing bursty flows and large differences in the number of dropped packets among competing connections [72].

It is thus likely that bottleneck Drop Tail buffers experience long term block and subsequently long term emptiness before reaching any steady states. Therefore, performance is considerably degraded with the coupled mechanisms of TCP and Drop Tail queuing under heavily loaded traffic conditions.

3.2 RED

RED has been designed to avoid some of the drawbacks of Drop Tail by means of congestion avoidance [51, 61]. Since the failures of Drop Tail are related to late congestion indication and consecutive packet drops, RED deploys early congestion detection and random drop.

3.2.1 RED Parameters and Dropping Probability

Upon the arrival of packets, RED first computes the average queue size $\bar{q}$ by means of an exponential weighted moving average:

$$\bar{q} = (1 - w_q)\bar{q} + w_q q \tag{3.1}$$

where w_q is the time constant of the low-pass filter and q is instantaneous queue length. The average queue size $\bar{q}$ thus tends to smooth out traffic fluctuations to prevent any biases against bursty traffic and filter out temporary congestion.

The variable $\bar{q}$ is then used to estimate the congestion level by comparing it with two predefined thresholds, a minimum threshold min_{th} and a maximum threshold max_{th}. More specifically, the dropping probability pr is determined for the incoming packets as follows.

- $\bar{q} < min_{th}$ (congestion free phase): There is no congestion and pr is 0.
- $min_{th} \leq \bar{q} < max_{th}$ (congestion avoidance phase): Incipient congestion occurs and pr increases linearly as $\bar{q}$ increases until $\bar{q}$ reaches the maximum threshold max_{th}. The determination of pr is based on the formula:

  $$pr = max_p \frac{\bar{q} - min_{th}}{max_{th} - min_{th}}, \tag{3.2}$$

  where max_p is the maximum predefined value for pr, when $\bar{q}$ is within the range of $[min_{th}, max_{th})$.
- $\bar{q} \geq max_{th}$ (congestion control phase): There is persistent congestion and pr is 1.

The function for calculating pr is also depicted in Figure 3.2. However, in the case of $\bar{q}$ within the range of $[min_{th}, max_{th})$, the value is only an initial dropping probability. To further avoid consecutive packet drops, the computation of the final value takes account of the number of packets enqueued since the last drop, denoted as *count* on the basis of the formula:

$$pr = \frac{pr}{1 - count \cdot pr}. \tag{3.3}$$

This way, at a given average queue size level, the number of arriving packets between dropped packets is a uniform random variable, with which dropping events occur at fairly regular intervals rather than close together.

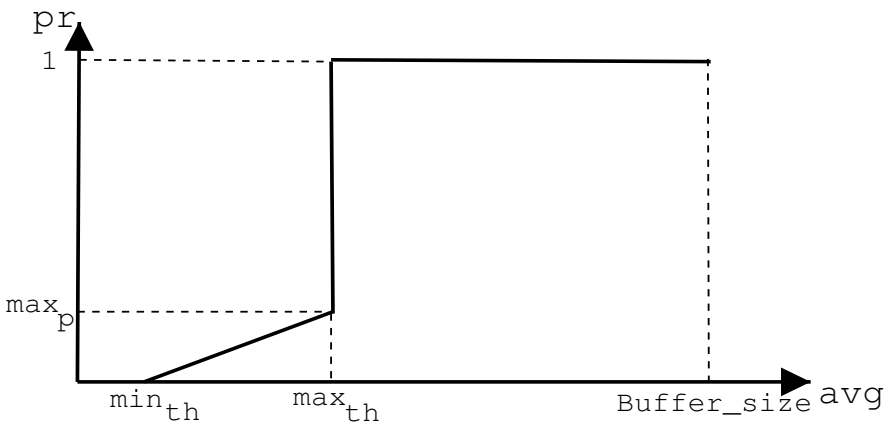


Figure 3.2: RED dropping probability

Ideally, RED controls the average queue length within the range of $(0, max_{th})$ to avoid persistent congestion. Thus RED operates in the congestion free phase and the congestion avoidance phase. Nevertheless, for severe congestion, RED adopts a strong reaction and performs in the same way as Drop Tail.

RED, unlike Drop Tail, thus proactively detects any incipient congestion by estimating average queue length and informs those active connections with higher data intensity to back off their sending rates. RED leaves time for the endpoints of the TCP connections to respond to network congestion and avoids congestion becoming severe while maintaining high link utilization. It is stated in [51] that, with appropriate settings of all its control parameters, RED is able to eliminate global synchronization and achieve fairness among competing connections with simplicity of implementation. Table 3.1 summarizes the control parameters of RED.

Table 3.1: RED Parameters

Parameter	Function
w_q	Weight for estimating average queue size
max_p	Maximum dropping probability
min_{th}	Lower threshold
max_{th}	Upper threshold

3.2.2 Gentle Mode of RED

In the original RED, the dropping function is linear between 0 and max_p as the average queue size varies from min_{th} to max_{th}, while it steps from max_p to 1 after the average queue size reaches max_{th}. As recommended later on in [46], the default value of max_p should be 0.1. Therefore, to smooth out the large jump from 0.1 to 1 and achieve robust performance, the dropping function is also linear between max_p to 1 when the average queue size varies from max_{th} to $2 * max_{th}$ as shown in Figure 3.3. This removes the rigidity in the calculation of pr in RED and hence is called gentle RED.

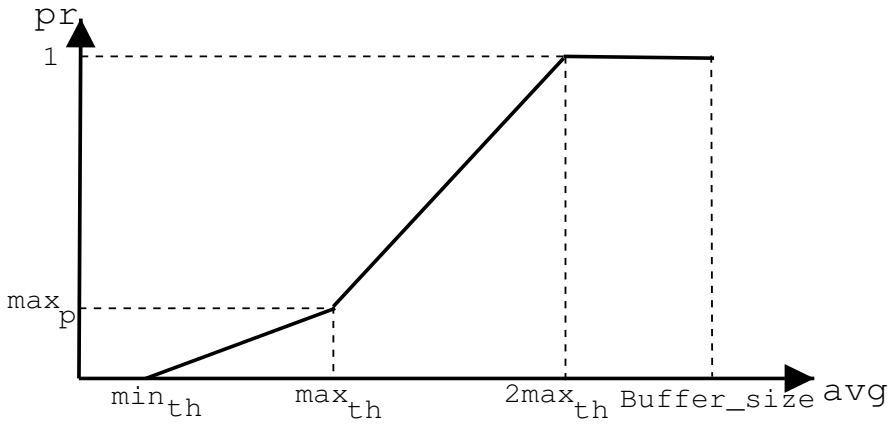


Figure 3.3: RED dropping probability with gentle mode

One of the first adoptions of gentle RED appeared in [89]. The paper investigates the performance of gentle RED in ATM UBR networks supporting TCP traffic by means of experimental simulations. The Simulation results were compared with those of plain UBR, Early Packet Discard (EPD), and Fair Buffer Allocation (FBA). The author concluded that gentle RED is able to achieve low delays while offering high TCP goodputs and link utilization.

How to implement RED with gentle mode for the best behavior of RED is discussed in [45].

3.2.3 Explicit Congestion Notification

TCP has exhibited robust features in congestion control. Packet losses, detected through duplicated ACK packets or the retransmission timer, are regarded as an indication of congestion. TCP accordingly throttles back its sending rate. The approach

of using packet losses as control feedback for TCP, however, is suboptimal. One reason is that once a packet is dropped before it reaches its destination, all the resources it has consumed in transit are squandered. In this aspect, it is expensive to lose a packet on its way, although dropping a packet is a very cheap operation in terms of router processing [83]. Apart from the waste of network resources and low network throughput, performance in delay is also degraded, which will impact on delay-sensitive TCP connections. In addition, for short-lived traffic, which is loss-sensitive, one packet drop can significantly affect the overall flow transmission performance.

With an intention to improve network efficiency and the performance of loss and delay-sensitive applications, explicit congestion notification or ECN has been proposed. This proposal first appeared in RFC 2481 [85] as an Experimental Protocol for the Internet Community. Later on it was extended as a Standard Track in RFC 3168 [86], where the usage of the ECN field in the IP header was clarified. The ECN field consists of two bits and makes four ECN codepoints, '00' to '11'. With ECN, once an IP router detects impending congestion before the buffer overflows, messages are passed on to TCP sources through marking with the Congestion Experienced (CE) codepoint 11 in the ECN field of the IP header, instead of dropping packets. Moreover, ECN requires support from the transport protocol. TCP thus has to be modified to be ECN-capable. Negotiation between the TCP endpoints might be involved during setup through SYN (synchronization) and SYN-ACK packets to determine that all of the TCP endpoints are ECN-capable, so that the sender can inform the network of its ECN capability through marking with the ECT codepoint 01 or 10 in the ECN field of transmitted packets. Upon the receipt of a packet with the CE codepoint set, the TCP receiver forwards the congestion information in the header of its next TCP ACK sent to the TCP sender. The TCP sender reacts to the congestion as if a packet had been dropped. Also, the TCP sender informs the receiver in its next packet of its acknowledgment and reaction. In addition, the end-systems should react to congestion at most once per RTT to avoid reacting multiple times to multiple indications of congestion within a RTT. Note that if severe congestion occurs or the buffer runs out of space, then the router has no choice but to drop the new arrival packet.

Clearly, it is required that the queue management in ECN-capable routers is capable of detecting incipient congestion before buffers overflow, which the traditional Drop Tail

scheme fails to do. The implementation of ECN thus has to be coupled with an AQM scheme. Consequently, the performance of ECN is dependent on the associated AQM mechanism.

The use of ECN prevents unnecessary packet drops and packet delay. ECN also benefits sources being informed of congestion quickly and unambiguously, without the source having to wait for either the retransmit timeout or duplicated ACK packets to infer a packet loss [49]. Moreover, the robustness of the ECN protocol allows the incremental deployment of ECN in both end-systems and in routers.

The deployment of ECN has been strongly supported by some studies but not all. For instance, in [83] the authors first illustrated that global synchronization, which RED targets to solve, is no longer an issue. The syndrome could happen with Tahoe TCP and has been addressed by the current TCP and it subsequent variants with the fast retransmission and fast recovery algorithms. The simulation results then showed that although ECN leads to fewer packets drops, TCP goodput is not guaranteed to improve under uniform condition where all the hosts and routers are ECN compatible. On the other hand, in no case does ECN seem to lead to an actual degradation in performance. The explanation for this was that TCP/ECN behaves very similarly to standard TCP for detecting network conditions. Also, it was concluded that the presence of the TCP/ECN sender can have significant impact on the performance of the ECN-unaware senders since it reacts faster to incipient congestion and can thus avoid unsuccessful data transmission. However, one positive sign in the paper is the achievement of a fairer resource allocation among competing connections under uniform condition.

From now on, we use the terms ‘dropping’ and ‘marking’ in queue management as exchangeable for either dropping a packet or setting the ECN field in its header, unless otherwise specified.

3.2.4 Parameter Settings in RED

The paper [51] elaborated the motivations and the operation of the RED algorithm, and demonstrated its merits with several simple simulations through the measurements of network throughput and queuing delay. In addition, the paper analyzed RED control parameter settings and presented a guideline based on networking experience. Since

then, many studies have been carried out to evaluate its performance by means of experiments, simulations, or theoretical analysis.

Despite the guideline in [51] and subsequent discussions of setting parameters in [46], RED parameters are sensitive to network configuration and traffic mix. For instance, the observations from the simulation results in [34] suggest that for traffic patterns different in the number of active TCP connections, different values of $\max_p$ in RED are required. By using a simple testbed made of up to 16 PCs and two CISCO 7500 routers, [25] presents that the RED parameters have minor impact on the performance with small buffers compared to Drop Tail, while with large buffers RED indeed can improve the systems performance. However, [25] also indicates that RED is difficult to calibrate and parameter tuning in RED remains an inexact science.

Given the sensitiveness of the RED parameters, some configuration recommendations have been made based on control theory. For example, in [44], based on the drop-conservative policy and the delay-conservative policy, two stability conditions were introduced to derive the configuration for RED parameters: max_p, min_{th}, and max_{th}, respectively. Also, the averaging weight w_q is configured by using averaging interval and sampling interval. Based on their observations in [31] that the RED queue converges to a state of heavy oscillation in the presence of two-way bulk-data TCP traffic in a range of traffic conditions, the stability criteria of RED with TCP traffic was obtained in [99] for setting the RED parameters, and for ensuring the queue size to achieve a state of bounded oscillation around a desired equilibrium point, i.e. $(min_{th} + max_{th})/2$. A conservative stability condition is formulated in [87] with buffer size as one variable, by using a discrete-time map to model the TCP-RED interaction. While the model in [87] is first-order discrete nonlinear, a linearized model was adopted to mimic the interconnection of TCP and a bottlenecked queue in [18] and to derive a configuration recommendation for the RED parameters.

It would be problematic in the implementation of RED, if one uses the existing RED configuration recommendations, due to the requirements of the network information in the majority of these models. Especially, in the case of the need of the number of active connections and flow RTT because of the constantly changing nature of the Internet, RED configuration becomes even more complicated. Since their research results have not agreed with the claims made in [51] about overcoming the problems of

Drop Tail queuing, some researchers have not recommended that RED be deployed. On the basis of queuing theory simple analytic models were developed in [5] for the RED and Drop Tail queue management schemes. Both numerical and simulation results did not show clear benefits of RED over Drop Tail, in terms of loss rates, the number of consecutively lost packets, mean delay, and delay jitter. Moreover, replacing average queue size with instantaneous queue size, RED effectively avoid global synchronization. Due to the popularity of Web traffic in the Internet, the effects of RED on the performance of Web browsing has been studied via extensive Lab experiments, with regards to end-user response times. The RED performance was assessed across a range of parameter settings and given traffic loads. For the given traffic loads up to 90% of link capacity, there is little difference in response times between RED and Drop Tail, while with the given traffic loads which approach the link capacity, RED is able to yield better performance. However, the better performance in response time was achieved by carefully tuning the RED parameters based on a trial-and-error approach, and by sacrifying link untilization. It was suggested that adequate provisioning is more efficient for transmitting Web traffic than tuning the RED parameters.

Although RED suffers the difficulty of tuning its parameters, RED itself has highlighted the necessity and potential benefits of conducting queue management in a proactive way. This suggests two possible directions in the development of AQM: first, making static RED becoming adaptive to network conditions; second, development of new AQM schemes. A survey of these two approaches to the development of active queue management is given [1] in the following section.

3.3 RED Variants and Other AQM Schemes

Due to the difficulty in finding a set of suitable parameters for RED under different congestion scenarios and the dynamic nature of network communication, constant self-tuning of these parameters is needed to improve the RED performance. One modified version of RED was reported in [34], where the adjustment of max_p is done by examining the variations in average queue length. This way, whether RED should become

[1] A distinction is made here between AQM, the general concept, and an AQM, a specific active queue management scheme.

either aggressive or conservative is based on the feedback signal of the changes in traffic loads. In the paper, the authors also emphasized the significance of combining AQM with ECN to effectively reduce packet loss in congested networks. Another RED variant is Adaptive RED (ARED) presented in [50], which has the goal of gaining predictable average queuing delay besides high TCP goodput and link utilization. To achieve these goals, ARED attempts to maintain the average queue length around a desired target queue length within a certain deviation. ARED adopts AIMD policy that additively increases max_p if the average queue size is above the target range, whereas multiplicatively decreasing max_p if the average queue size is below the target range. In addition, ARED also auto-tunes the RED weight parameter w_q related to link capacity in a way that higher-speed links require smaller values for w_q. Furthermore, since using a constant for the increase step size in the AIMD policy of ARED is not optimal in a variety of traffic scenarios, a fuzzy logic (FL) approach, FLARED [38], has been introduced with an intention to modify the ARED algorithm by adaptively obtaining a value for the increase step size.

The essence of these schemes is to remove the sensitivity of the RED parameters while retaining RED's spirit. More importantly, these variants use the change in queue size along with average queue length to estimate traffic loads in the network and the effect of the current value of the dropping probability pr on it, and adjust pr accordingly. It is obvious that RED using queue size alone fails to accurately gain such information that the performance of RED is sensitive to the parameter settings and traffic loads.

Besides these modified RED mechanisms, some new AQM schemes have been devised. Most of them have been designed on the basis of domain knowledge, experience, or statistical methods. For instance, instead of using either average queue length or instantaneous queue length, BLUE [41] performs queue management on the basis of two extreme events: packet losses and link idle with an additive-increase additive-decrease (AIAD) policy. In particular, pr is additively incremented, when the buffer is full and a certain time has passed since the immediately previous increase action. In the case of an idle link or empty queue event, the action of additive decrease of pr is taken if necessary. However, BLUE suffers the problem of oscillation in transient conditions. To alleviate the oscillation between loss and under-utilization, GREEN [94], upon the receipt of a new packet, uses exponential averaging to estimate input rates, increments

pr by a predefined constant if the estimate is below a target link capacity, and decrements it by the same value otherwise. Based on a steady-state formulation of TCP, another version of GREEN [92] computes the marking probability for each flow by taking account of both the number of flows and its RTT, provided that routers are willing to collect such information. To distinguish these two GREEN algorithms, the former is called GREEN1 while the latter GREEN2. Stabilized RED (SRED) [77] computes *pr* based on the instantaneous buffer occupation and on the statistically estimated number of active connections N. Two dimensionless measures, offered load and link occupancy, are used for congestion notification by Acceptance and Departure Rate (ADR) in [21].

Random Exponential Marking (REM) [4] is a practical implementation of a representative optimization approach [73] to reactive flow control, where the goal is to calculate source rates that maximize the total user utility over the aggregate source rate at each link subject to capacity constraints. The flow control is derived by solving a dual problem using gradient projection algorithm. Structurally, REM consists of a link algorithm and a source algorithm. In the link algorithm, first congestion is measured by a variable, called link price, based on the difference between input rate and link capacity (the difference is called rate mismatch), and the difference between queue length and target (the difference is called queue mismatch). The link price is increased if the weighted sum of rate and queue mismatches is positive, and decreased otherwise. Then the marking probability is computed by an exponential function of the current price; each arrival packet is marked accordingly if the upstreams have not done so. The adoption of the exponential marking probability is due to the fact that it allows the end-to-end marking probability to be exponentially increasing in the sum of the link prices at all the congested links in its path. Consequently, the endpoint, which executes the source algorithm, is able to estimate the path congestion measure and adjusts its rate based on the estimate. In REM, the link algorithm works as an AQM scheme [27] in a router. The term 'REM' from now on is used for the link algorithm unless otherwise specified. However, the derivation from a static model implies that the performance of REM may suffer in transient network conditions.

Using a penalty function approach for the same optimization problem in REM, the Adaptive Virtual Queue (AVQ) link algorithm was derived. AVQ has been discussed in [65, 64, 66] along with the issues including an analytic solution and a simplified

practical solution, stability, and parameter configuration guidelines. Corresponding to the concept of link price in REM, AVQ deploys virtual capacity or marking level to reflect the network conditions. Virtual capacity, which is non-negative and less than the actual capacity, is adapted as a function of the difference between a desired link capacity and the total arrival rate in a way that virtual capacity is increased when the difference is positive, and decrease otherwise. The idea behind AVQ is that when the arrival rate exceeds virtual capacity, marking should take place so that the total arrival rate on each link is regulated towards a target link capacity and eventually the desired utilization is achieved. Taking the adaptive virtual capacity and the real buffer size into account, a virtual buffer admits arrival packets until it overflows. Thus, the marking probability in AVQ is implicit. Moreover, it has been stated that in order to avoid successive packet drops from the same flow, AVQ could use an appropriate random dropping mechanism like the one in RED. To be robust, AVQ chooses the desired utilization to be less than one to accommodate the transient behavior due to the presence of extremely short flows or variability in the number of long flows in the network.

A Proportional Integral (PI) controller [19] is another AQM solution from the classical control theory standpoint. In a linearized TCP/AQM control system, integral plus proportional control is selected for queue management with the potentials of decoupling the queue size and the dropping probability, reaching a desired reference value of queue size or a set-point, and achieving faster responsiveness. Interestingly, the dropping probability of PI is in the same form of link price in REM but with more robust transient performance. Another example of a control theory based AQM scheme is a variable structure (VS) based control approach proposed in [96] to accommodate the nonlinear nature of TCP/AQM.

In the literature, Fuzzy Explicit Marking (FEM) [17] is based on a modern control theory, fuzzy logic (FL). FEM calculates dropping probability pr based on queue states in a certain sampling period. The error on the queue length, the difference between a desired value and the current instantaneous queue length, is regarded as an important feedback variable. The errors from two consecutive sample periods are also used as an input in FEM.

3.4 Survey of AQM Schemes

All the AQM schemes address three issues in order to do queue management, including choosing appropriate congestion indicators, selecting control principles, and calculating marking probability. In the following two subsections, some of the existing AQM mechanisms are reviewed in these three aspects.

3.4.1 Congestion Indicators and Control Principles

Congestion indicators in AQM are used to detect incipient congestion and reflect congestion severity. For example, in RED, exponentially weighted average queue length (EWAQ) is used as a congestion indicator. When traffic load increases and the output buffer is built up, the marking probability of RED thus is increased to alleviate the incipient congestion. RED thus has to firstly experience longer queuing delay before conducting control. In addition, queue length gives very little information about the severity of congestion, the number of active flows passing the link [41]. Therefore, many other AQM schemes try to avoid the coupling of congestion indication and performance measure such as delay and loss rate, and adopt other variables such as the number of active flows and input rate as congestion indicators. Apart from congestion indicators, control principles direct the design and eventually the control system moves towards or around a desired steady state. In Table 3.2, the congestion indicators and control principles of those typical AQM mechanisms have been given. Note that in the table queue mismatch refers to difference between queue length and a target, while rate mismatch is the difference between input rate and a target link capacity, as in REM and GREEN1.

Note that in Table 3.2, queue length might mean different concepts for each AQM scheme. In particular, for RED, SRED, and ARED, it refers to EWAQ; for REM, it is smoothed or instant queue length [4]; for PI, it is instant queue length.

3.4.2 Marking Probability

Each AQM scheme calculates marking probability based on chosen congestion indicators and control principles. First, a list of commonly used notions is given below.

Table 3.2: Congestion indicators

AQM schemes	Congestion indicators	Control principles
RED	Queue length	Keep queue length within a range $[min_{th}, max_{th}]$
SRED	Queue length and statistically estimated active flow number	Keep queue length stable and independent of the number of active connections
ARED	Queue length and queue mismatch	Provide average queuing delay
REM	Link price	Match rate clear buffer
PI	Queue mismatch and change in queue length at two consecutive instants	Match rate clear buffer
BLUE	Packet loss and link idle	Prevent packet loss and keep link utilization high
AVQ	Virtual capacity	Regulate link utilization to a desired value
GREEN1	Rate mismatch	Match rate clear buffer
GREEN2	Active flow number and RTT	Apply the knowledge of steady-state TCP congestion control behavior in routers
ADR	Offered load and link occupancy	Become scalable with varying link capacities, buffer sizes, and traffic loads

c – link capacity
$\bar{q}$ – EWAQ
pr – dropping probability
N – the number of active flows
RTT – round trip time
q – queue size
q_0 – desired queue size
b – packet size
MSS – maximum segment size
p – packet loss probability

In RED, first $\bar{q}$ is calculated by $(1 - w_q)\bar{q} + w_q q$, where w_q is time constant. The pr can be obtained by two steps as follows:

$$p_b = max_p(avg - min_{th}/(max_{th} - min_{th}))pr = pr/(1 - count \cdot p_b) \tag{3.4}$$

where max_p is the predefined maximum value for pr, and count is packet number since the last marked packet.

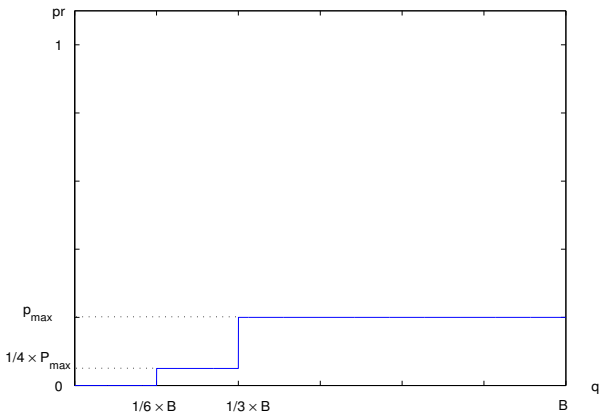
SRED statistically estimates the number of active flows N as the inverse of hit frequency P. SRED maintains the information of the recently seen flows in a "Zombie List". If incoming packet flow information matches a randomly chosen zombie in the

list, a "hit" is declared. Otherwise "no hit" is declared.

$$hit = \begin{cases} 0 & \text{if } no\ hit \\ 1 & \text{if } hit \end{cases} \tag{3.5}$$

Average P by $P = (1-\alpha)P + \alpha Hit$. Then, pr is calculated in two steps as follows [2]:

$$p_b = \begin{cases} p_{max} & \text{if } \frac{1}{3}B \le q < B, \\ \frac{1}{4} \times p_{max} & \text{if } \frac{1}{6}B \le q < \frac{1}{3}B, \\ 0 & \text{if } 0 \le q < \frac{1}{6}B. \end{cases} \tag{3.6}$$



$$pr = \begin{cases} \frac{p_b}{65{,}536} \times N^2 & \frac{1}{256} \le P \le 1 \\ p_b & 0 \le P < \frac{1}{256} \end{cases} \tag{3.7}$$

where B is buffer capacity.

The main difference between ARED and RED is that ARED periodically adjust max_p based on the difference between $\bar{q}$ and a target range of queue length, $[q1, q2]$, as follows.

$$max_p = \begin{cases} max_p + \alpha & \bar{q} > q2 \\ max_p * \beta & \bar{q} < q1 \\ max_p & q1 \le \bar{q} \ge q2 \end{cases} \tag{3.8}$$

where α and β are constant.

REM periodically adapts link price p based on the weighted sum of rate mismatch between input rate λ and c, and queue mismatch between queue size and q_0 as follows:

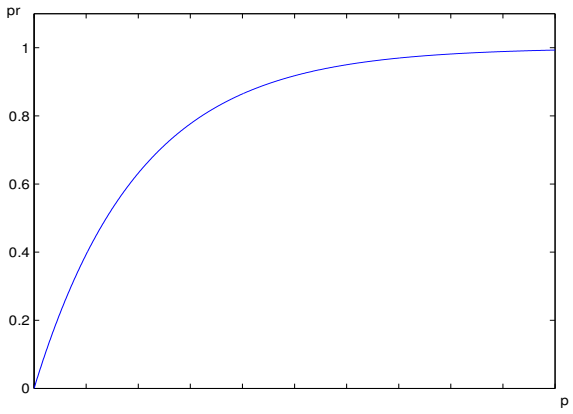
$$p = [p + \gamma(\alpha(q - q_0) + (\lambda - c))]^+ \tag{3.9}$$

where $\gamma > 0$ and $\alpha > 0$ are small constants and $[z]^+ = \max\{z, 0\}$. Then pr is calculated

[2] The figure for the first step of the calculation is plotted for reference.

as follows [3]:

$$pr = 1 - \phi^{-p} \tag{3.10}$$



where $\phi > 1$ is a constant.

PI uses two consecutive queue mismatch to update pr as follows in a period basis:

$$pr = pr + \alpha * (q - q_0) - \beta * (q_{old} - q_0) \tag{3.11}$$

where q_{old} is the queue size in the last instant, α and β are PI coefficients.

BLUE periodically adjusts pr based on packet loss and link idle events. If a packet loss event occurs, additively increase the marking probability by a constant $d1$, while if a link idle event occurs, additively decrease the marking probability by a constant $d2$.

AVQ updates virtual queue size by $VQ = max(VQ - \tilde{C}(t - s), 0)$ upon a packet arrival, where $\tilde{C}$ is virtual capacity, t is the current instant, and s is the last packet arrival time. If the new packet can not be accommodated by the virtual buffer due to the buffer size limit, it is dropped. Otherwise, the virtual queue size is increased by its size b. Moreover, virtual capacity $\tilde{C}$ is adapted upon the admission of an arrival packet as follows:

$$\tilde{C} = max(min(\tilde{C} + \alpha * \gamma * C(t - s), C) - \alpha * b, 0) \tag{3.12}$$

where α is the damping factor and γ is the desired link utilization.

GREEN1 computes pr as follows:

$$pr = pr + \Delta P \cdot U(x_{est} - c_t) \tag{3.13}$$

[3]The figure for the calculation is plotted for reference.

where ΔP is a constant and c_t is the target link capacity, which is just below the actual link capacity c. The estimated input rate x_{est} is set as $(1 - exp(-Del/K)) * (b/Del) + exp(-Del/K) * x_{est}$, where Del is the inter-packet delay and K is the time constant. The step function $U(x)$ is defined by:

$$U(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0. \end{cases} \tag{3.14}$$

GREEN2 derives the computation formula for pr on the basis of the TCP steady-state behavior modeled in the form of

$$throughput = \frac{MSS \times C}{RTT \times \sqrt{p}}, \tag{3.15}$$

where C is a constant. By replacing $throughput$ by $\frac{c}{N}$, the dropping probability for flow i, pr_i, is

$$pr_i = (\frac{N \times MSS_i \times C}{c \times RTT_i})^2. \tag{3.16}$$

ADR can be timer-based, where the fraction of traffic that can be allowed f is computed periodically. First ADR measures offered load α by acceptance rate normalized by c and link occupancy β by departure rate normalized by c. Then f is adapted with MIMD policy as follows:

$$f = [f \times \min\{\phi_{ar}, \phi_{dr}, \phi_{\max}\}]^1_{f_{min}}, \tag{3.17}$$

where ϕ_{ar} is the ratio of a threshold α_{peak} to α, ϕ_{dr} is the ratio of a line threshold β_{thresh} to β, ϕ_{max} is an predefined upper bound on ϕ_{ar} and ϕ_{dr}, f_{min} is the lower limit on f, and 1 is the upper limit on f. In the deterministic scheme for the decision making for the acceptance of arrival packets, a variable *deterministic_pr* is calculated based on the following procedure.

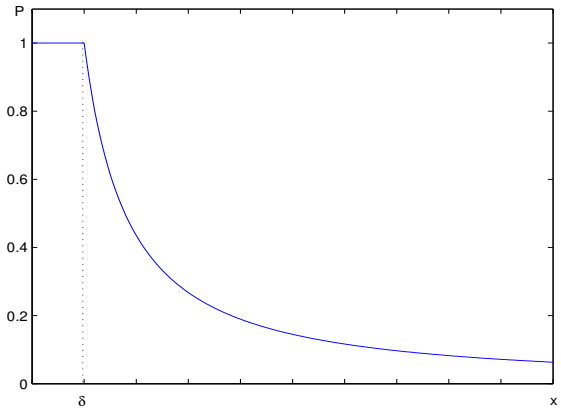
```
deterministic_pr += f
if (deterministic_pr >= 1)
        deterministic_pr -= 1
        accept packet
else
        reject packet
endif
```

3.5 Traffic Modelling

Internet traffic has been modelled as a Poisson process in the past. At the present time, the composition of Internet traffic has been dramatically changed with a lot of Web-like short-lived connections. We use an elegant mathematical model, namely *Poisson arrival process of Pareto length bursts (Poisson-Pareto)*, in modelling Internet traffic. The model is based on a number of characteristics, which have been observed in Internet traffic. First, traffic manifests self-similarity, this is, traffic is bursty over a wide range of time scales [93]. This phenomenon was first noticed in LAN [33] and later on WAN [82]. Second, heavy-tailed property is associated with file sizes. It is well known that Internet traffic tends to be made up of a large number of quite small flows, a small number of very large flows, and nothing much in between. This two kinds of flows are called mice and elephants, respectively. Despite the small number of elephants, the bulk of packets in the Internet are from elephants. Third, session arrivals are subject to Poisson process [52]. Here sessions are made up of multiple connections. The first two observations are also related. [93] reveals that the heavy-tailed property is a main contributor to aggregate traffic self-similarity.

With the Poisson-Pareto traffic model , traffic is generated in a way that flow arrivals form a Poisson process with an arrival rate of λ, so the intervals between adjacent flow arrival times are exponentially distributed with the mean of $1/\lambda$. In addition, flow lengths are distributed according to a Pareto distribution with shape parameter γ and scale parameter δ, which is the (necessarily positive) minimum possible value [2]. The complementary distribution function of a Pareto distribution is given by [4]:

$$P\{L > x\} = \begin{cases} 1, & x < \delta \\ \left(\frac{x}{\delta}\right)^{-\gamma}, & x \geq \delta \end{cases} \qquad (3.18)$$



[4]The figure for the distribution is plotted for reference.

with the mean is $E(L) = \frac{\delta\gamma}{(\gamma-1)}$.

Using a Pareto distribution with $\gamma < 2$ allows for the significant long bursts that characterize heavy-tailed traffic. Note that, provided research on structural models for Internet traffic is still at the preliminary stage, the Poisson-Pareto traffic model used is far from accurate. In fact, as claimed in [52], session arrivals are well described as Poisson but not at the connection level. However, it is concluded in [57] that although the arrival times of flows are correlated, the error in ignoring this with the use of a Poisson process is not nearly as important as the heavy-tailed nature of the flow length distribution. Moreover, in this research, a precise traffic model is not necessary for performance evaluation of some investigated traffic control schemes as long as the model is able to generate realistic traffic consisting of both mice and elephants with certain traffic loads at a large time scale.

3.6 Summary

A survey of existing AQM schemes has been carried out with regard to congestion indication, control principles, and marking probability calculation. Thus, the state of the art of AQM has been presented. In addition, traffic modelling has been discussed with the selection of the Poisson-Pareto model for traffic generation. In this research, we will use this traffic model in simulations for performance evaluation.

Chapter 4

Quantitative Analysis of AQM using Network Simulations

Having analyzed various AQM schemes qualitatively in Chapter 3, we will quantitatively analyze the performance of some typical AQM schemes in comparison with that of Drop Tail using simulations in this chapter. First, the experimental methodology used in our research is described in terms of network topology, traffic pattern, and performance metrics. The significance of AQM replacing Drop Tail and the limitations of some well-known AQM schemes are investigated through a wide range of experiments.

4.1 Experimental Methodology

Simulation is a cheap and flexible means of exploring proposed schemes in a varying environment in comparison to testbeds and laboratory experiments.

4.1.1 Network Simulator *NS2*

In this study, we use *NS2* [1] version 2.26 as the simulation platform to evaluate the performance of various queue management mechanisms. *NS2* is a discrete event simulator widely used in the networking research community. *NS2* is a collective effort from not only its four geographically-dispersed groups of developers including LBL, Xerox PARC, UCB, and USC/ISI but also its user community consisting of more than 200 institutions [28]. *NS2* is a framework supporting multiple protocols at

different layers. Thus, it is easy for researchers to incorporate the implementations of their proposals into *NS2*, observe the behavior of their designs, and compare with the performance of others.

NS2 is implemented in two languages: C++ and Tcl. The kernel of *NS2*, consisting of simulation primitives such as packet forwarding and low-level event processing, requires high computational performance and thus is implemented in the compiled language C++. On the other hand the user interface, including network configuration for protocols and traffic patterns, needs an implementation in a flexible and interactive language such as Tcl. It is worth mentioning that some well-known queuing management strategies including Drop Tail, RED, ARED, REM, PI, and BLUE are available in *NS2*.

4.1.2 Network Topology

Throughout our research, we intend to use a simple network topology as depicted in Figure 4.1 to simulate a network where a bottleneck link lies between a premise gateway in the network client side and an edge router in an ISP side. Studies show that such access links are among the most cost-sensitive and bandwidth constrained components in the Internet, and performance of such links remains a major concern of ISPs.

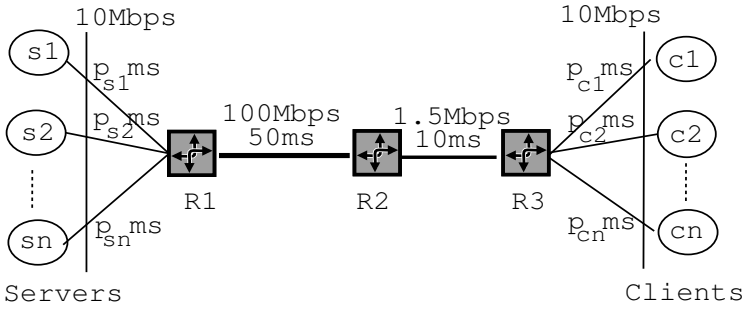


Figure 4.1: Network topology

In Figure 4.1, there are n servers, n clients, and three routers, where n is a positive number. $R1$ is a core router, while $R2$ is an edge router and $R3$ is a premise gateway. The access link between $R2$ and $R3$ is the assumed bottleneck with a link capacity of $1.5Mbps$, while the core link is over-provisioned with a capacity of $100Mbps$ and $10Mbps$ link capacity is the interface speed of each server and client. The propagation delay on the link from $R1$ to $R2$ and that on the link from $R2$ to $R3$ are 40ms and

10ms, respectively. The propagation delays on the two end links can be varied in order to get different RTTs among connections.

Traffic streams are from the servers to the clients and they all use TCP as transport protocol. For instance, Server si sends data to Client ci. We apply different queue management schemes in the output queue of router $R2$ towards $R3$ to conduct congestion avoidance in cooperation with TCP in the manners of not marking but dropping (unless specified otherwise), while using Drop Tail elsewhere. The network configuration parameters are given in Table 4.1. The maximum window size of each TCP connection is set with such a large value that TCP window size is only dependant on TCP and router queuing strategy. The buffer size of the bottleneck is around four times the product of bandwidth of $1.5Mbps$ and delay of around $200ms$ to allow traffic fluctuations from Web traffic and TCP bursty data.

Table 4.1: Network configuration

Network parameters	Value
TCP version	Reno
Packet size	1000 bytes
Maximum window size	100 packets
Buffer size of the bottleneck link	160 packets

4.1.3 Traffic Pattern

Since the change of applications in the Internet, the characteristics of Internet traffic have changed dramatically. In order to capture the essential features of real traffic we have adopted the Poisson-Pareto model mentioned on Page 52 for traffic in our simulations. The Pareto distribution we use has an average flow length of 12 packets, shape parameter of 1.2, and minimum value of 2 packets. The other important parameter defining a traffic scenario is the rate of arrival of flows. By varying this parameter we can set the traffic load, in bytes/s, to any desired value. More specifically, the traffic loads can be estimated using the following formula:

$$traffic\_load = \frac{(40 + afs \times 1040) \times 8 \times n \times \lambda}{c}, \tag{4.1}$$

where afs stands for the average flow size and measures in packets, n is the number of servers, λ is flow inter-arrival rate which is the number of arrivals per unit time (second), and c is the bottleneck link capacity. We explain the above formula as follows. SYN control packets are 40 bytes and each data packet is 1040 bytes with 1000 bytes of data and a 40-byte header. For instance, if $afs = 12$ packets, $\lambda = 5s^{-1}$, $c = 1.5Mbps$, and $n = 2$, then

$$\begin{aligned} traffic\_load &= \frac{(40 + 12 \times 1040) \times 8 \times 2 \times 5}{1.5 \times 10^6} \\ &\simeq 0.668 \\ &= 66.8\% \end{aligned}$$

4.1.4 Performance Metrics

To analyze QoS received by clients and network efficiency, two kinds of measures are utilized respectively for performance evaluation. The two measurements are user-centric and network-centric as follows:

- user-centric measures:
  - user goodput – the ratio of the amount of the packets received by the destination, excluding duplicated packets, to the time spent
  - response time – the time spent to complete a response; namely, the elapsed time interval from the server sending out the first packet of a response to the client receiving the last packet of the response
- network-centric measures:
  - network throughput – the ratio of the amount of the packets received by all the destinations, excluding duplicated packets, to the time spent
  - link utilization – the ratio of transmitted rate and link capacity
  - loss rate – the ratio of the dropped bits in the bottleneck link to the total arrival bits in the same measurement duration

Note that the user-centric measures only account for completed flows, whereas the network-centric measures involve all flows.

4.1.5 Confidence Level Analysis of Simulation Results

4.2 Simulations

We have qualitatively reviewed some typical AQM schemes in Chapter 3. In this section, simulations are conducted to compare the performance of some of these AQM schemes with that of Drop Tail. Our purpose is to highlight the importance of using AQM rather than the use of any particular AQM scheme. We select some of the existing AQM schemes mentioned in the previous chapter as representatives, including RED, ARED, REM, PI, BLUE, to do the comparison with Drop Tail. Note that five independent replications of a simulation are carried out to perform statistical analysis.

4.2.1 Basic Parameters and Configuration

On the server side, two servers are used to send data to their corresponding clients with different propagation delays to the core router $R1$, $5ms$ and $40ms$ respectively. On the client side, $1ms$ and $10ms$ are set for the link propagation delays, respectively. These two kinds of traffic are web traffic called http1 and http2 respectively, for convenience. Gentle RED is deployed and the RED parameter settings are given in Table 4.2. The time constant w_q is set as -1, by which RTT is first estimated to be three times the sum of the link propagation delay (PD) and transmission delay (TD) with the minimum value of $100ms$ followed by calculating w_q based on $w_q = 1.0 - e^{-1.0/(10*RTT/TD)}$. Such RED control parameter configuration follows the recommendations combined in [47] and [46] by taking our topology into account. The reference point of queue size is set as 60 packets in ARED, REM, and PI. Besides, we use the default settings for the selected schemes.

Table 4.2: RED parameter settings

Parameter	w_q	max_p	min_{th}	max_{th}
Value	-1	0.1	30	90

The simulation duration is 4000s. The first 2000s is chosen as the warmup time to filter network transience.

4.2.2 Simulation Results

Our primary purpose is to look into whether AQM outperforms Drop Tail in a way that both users and ISPs are satisfied. With this in mind, the performance of the selected AQM representative schemes is compared against that of Drop Tail in different traffic load conditions. We categorize traffic loads as follows. Traffic load of 50% represents light traffic, traffic load of 80% is medium, 90% and 100% is heavy, and 110% and 120% is very heavy, respectively.

1. **Observations on light traffic loads.** When the traffic load is light (with 50% traffic load), we can see that data has been transmitted smoothly mainly based on TCP flow control. The output buffer of the edge router R2, which connects to the bottleneck link, fulfills its purpose of absorbing bursty data. Figure 4.2 shows the typical queue length dynamics of Drop Tail with light traffic loads. The following observations can be drawn in the light traffic load condition.

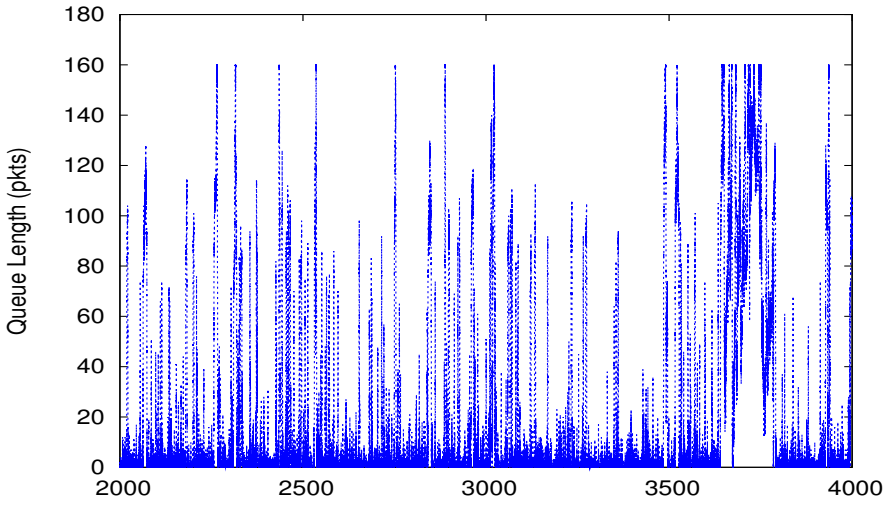


Figure 4.2: Drop Tail queue length with 50% traffic load

- The user TCP goodput of Drop Tail is competitive with those of the selected AQM schemes as shown by the weighted average [1] TCP goodput versus

[1]Weighted average calculated here takes account of the number of flows for a given flow size.

traffic loads in Figure 4.3. The relative TCP goodput performance versus file lengths with 50% traffic load in Figure 4.4 for traffic http1 and Figure 4.5 for traffic http2 also shows that AQM has very marginal improvement on TCP goodput of the mice traffic flows with long RTT.

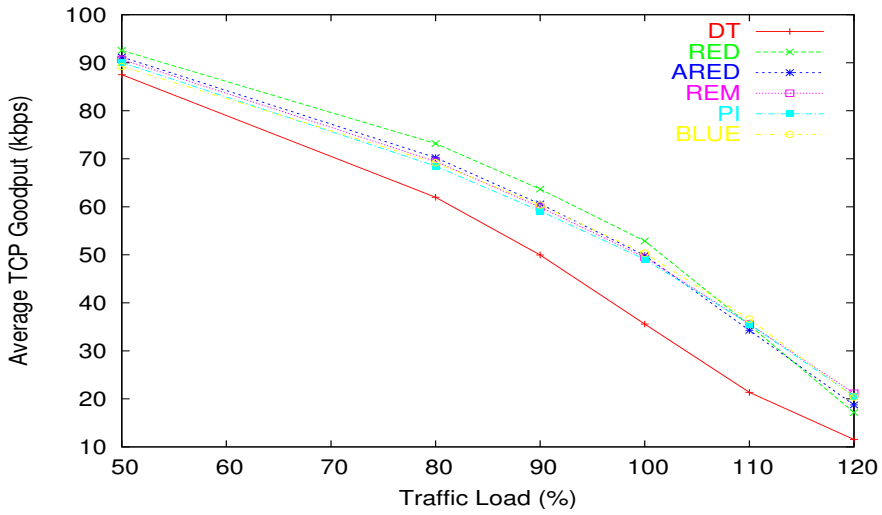


Figure 4.3: Weighted average TCP goodput versus traffic loads

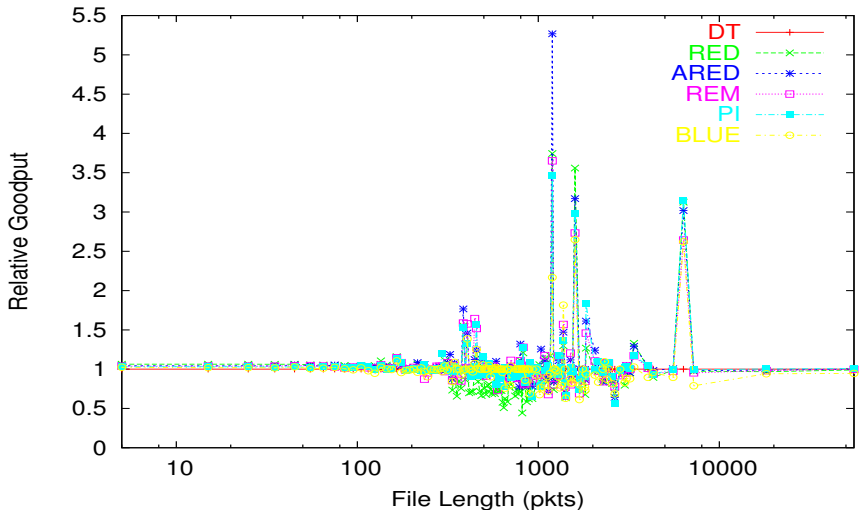


Figure 4.4: TCP goodput of traffic http1 relative to Drop Tail with 50% traffic load versus file lengths

- The user response time of Drop Tail is competitive with those of AQM as shown by the weighted average response time versus traffic loads in Figure 4.6 with the achievement of the lower latency with AQM for the mice of both short RTT and long RTT as shown by relative response time versus

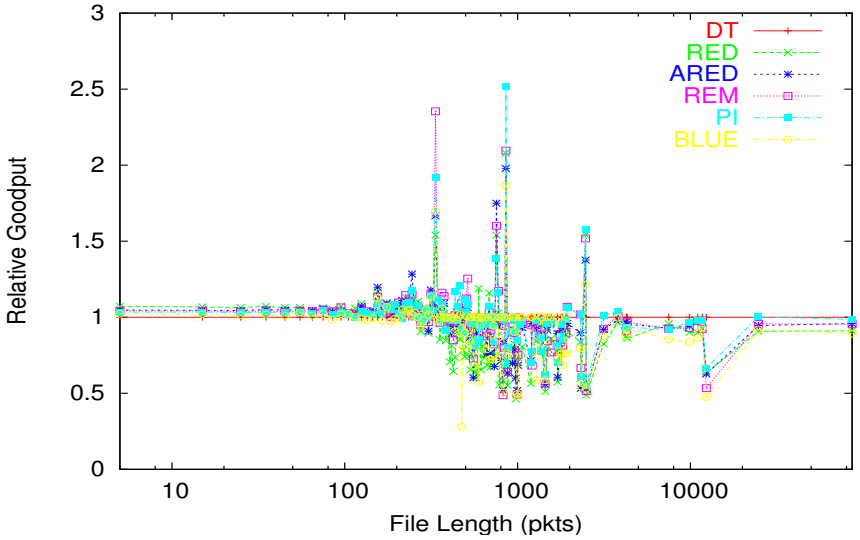


Figure 4.5: TCP goodput of traffic http2 relative to Drop Tail with 50% traffic load versus file lengths

file lengths with 50% traffic load in Figure 4.7 for traffic http1 and Figure 4.8 for traffic http2.

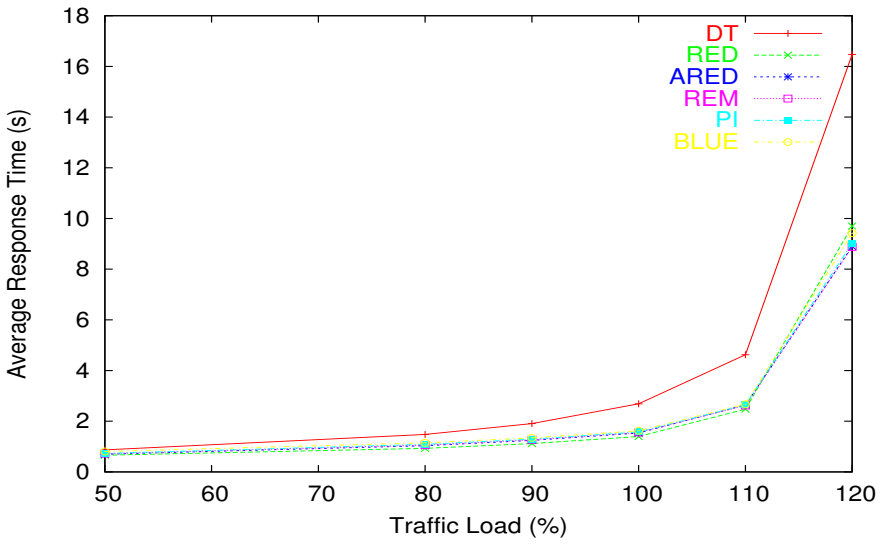


Figure 4.6: Weighted average response time versus traffic loads

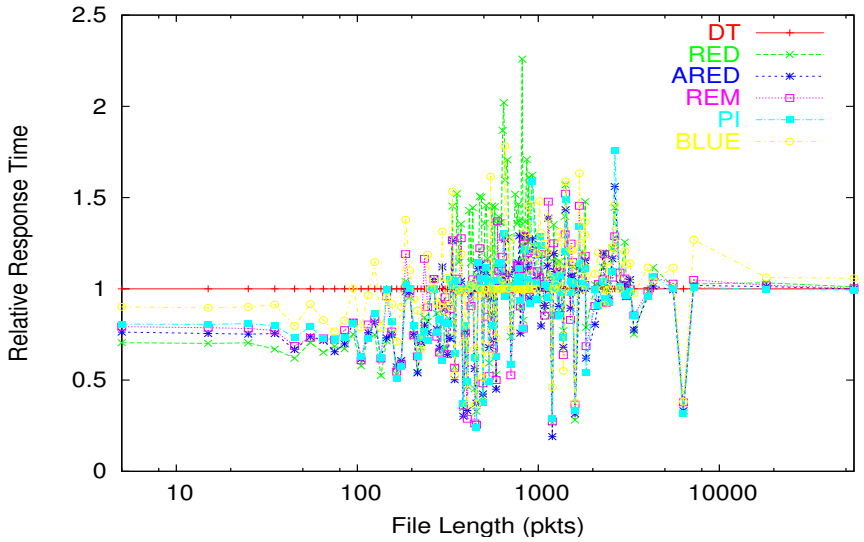


Figure 4.7: Response time of traffic http1 relative to Drop Tail with 50% traffic load versus file lengths

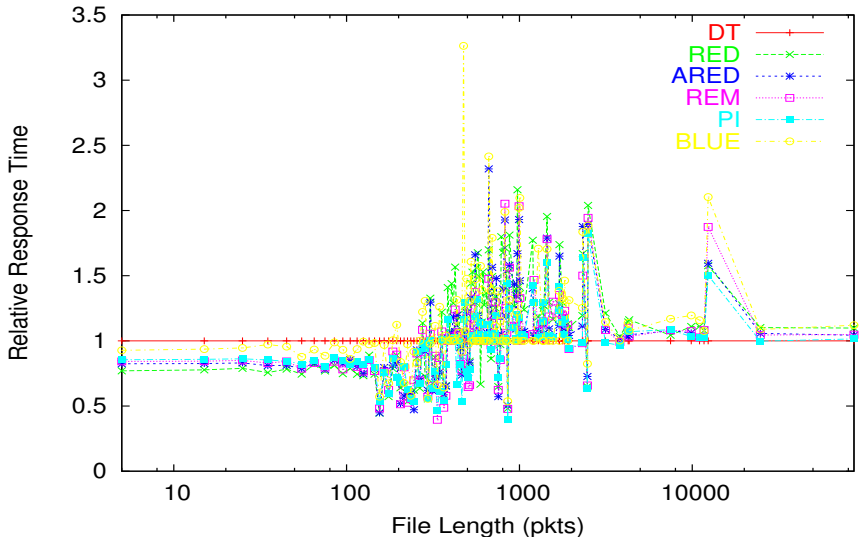


Figure 4.8: Response time of traffic http2 relative to Drop Tail with 50% traffic load versus file lengths

- Short RTT benefits in not only response time but also TCP goodput as shown in Figures 4.9 and 4.10 for weighted average TCP goodput versus traffic loads and Figures 4.11 and 4.12 for weighted average response time versus traffic loads by taking Drop Tail and RED as examples. Figures 4.13 and 4.14 have also shown that with light traffic loads, mice flows suffer from lower goodput with long RTT than with short RTT, and RED increases this bias.

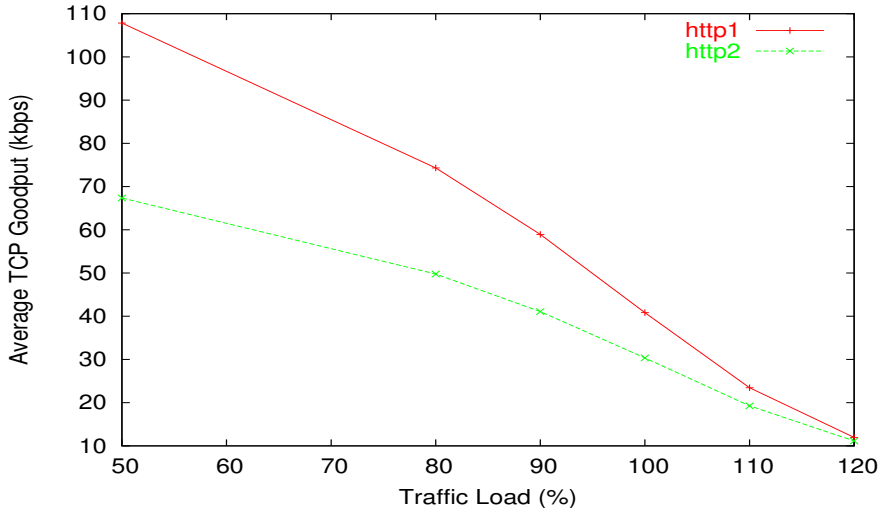


Figure 4.9: Weighted average TCP goodput comparison between traffic http1 and http2 with Drop Tail versus traffic loads

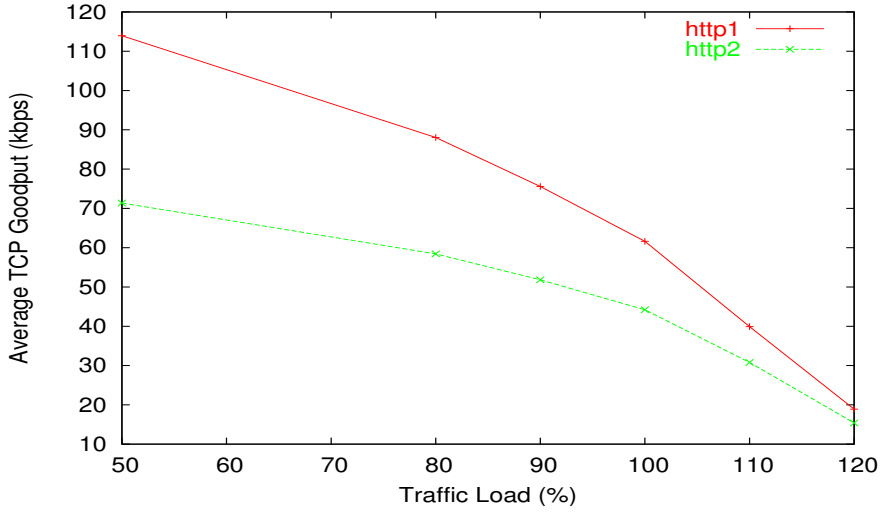


Figure 4.10: Weighted Average TCP goodput comparison between traffic http1 and http2 with RED versus traffic loads

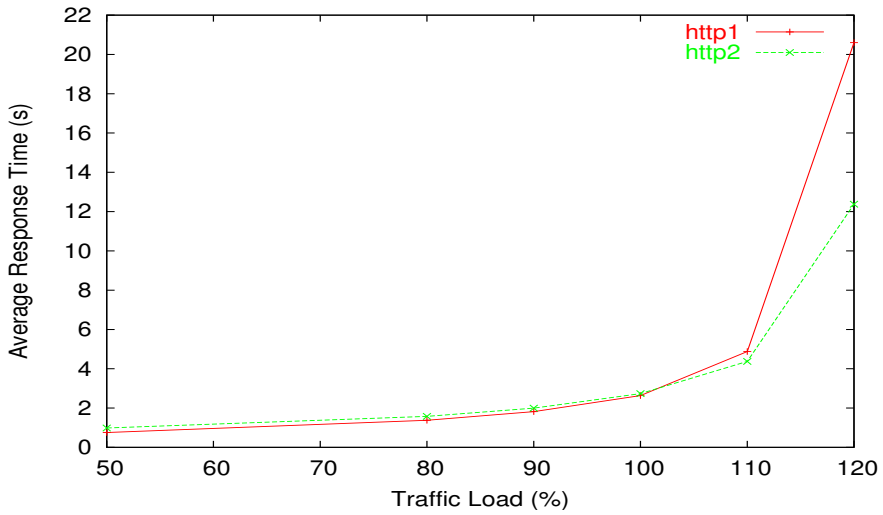


Figure 4.11: Weighted average response time comparison between traffic http1 and http2 with Drop Tail versus traffic loads

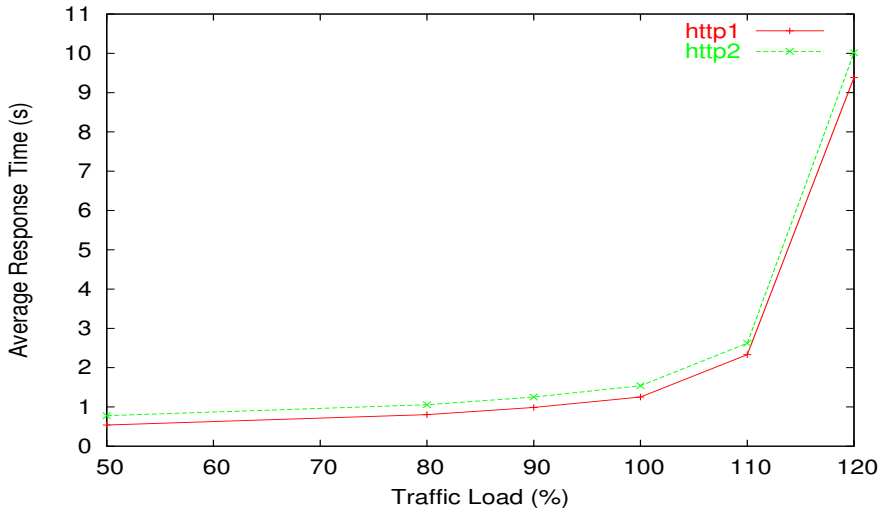


Figure 4.12: Weighted average response time comparison between traffic http1 and http2 with RED versus traffic loads

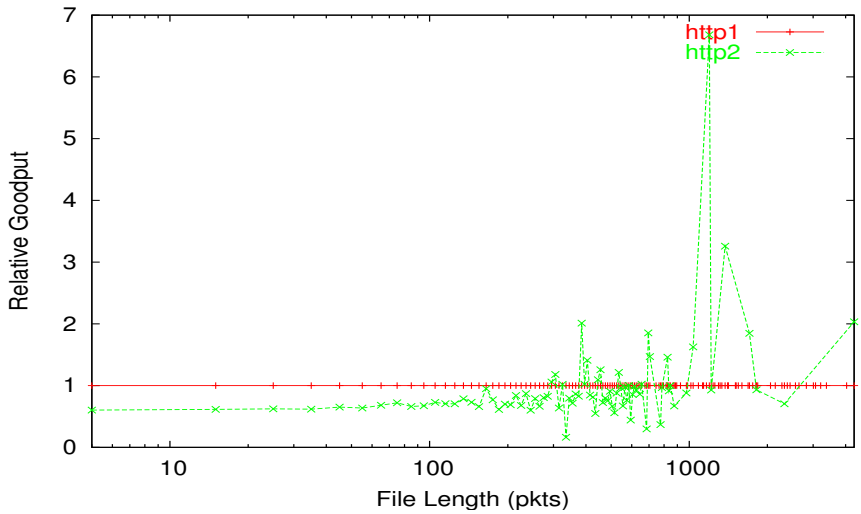


Figure 4.13: TCP goodput relative to traffic http1 with 50% traffic load with Drop Tail versus file lengths

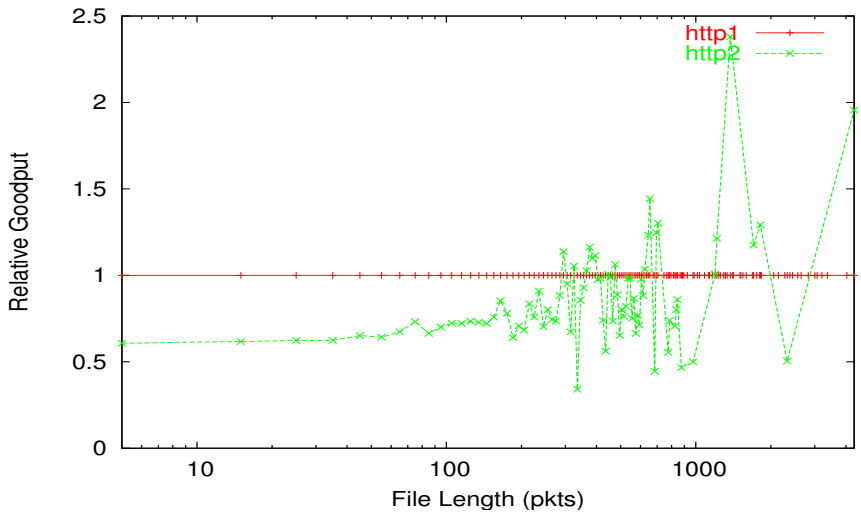


Figure 4.14: TCP goodput relative to traffic http1 with 50% traffic load with RED versus file lengths

- Mice flows in all schemes obtain poor goodput performance compared with their long counterparts as illustrated in Figure 4.15 for traffic http1 and Figure 4.16 for traffic http2 with both Drop Tail and AQM.

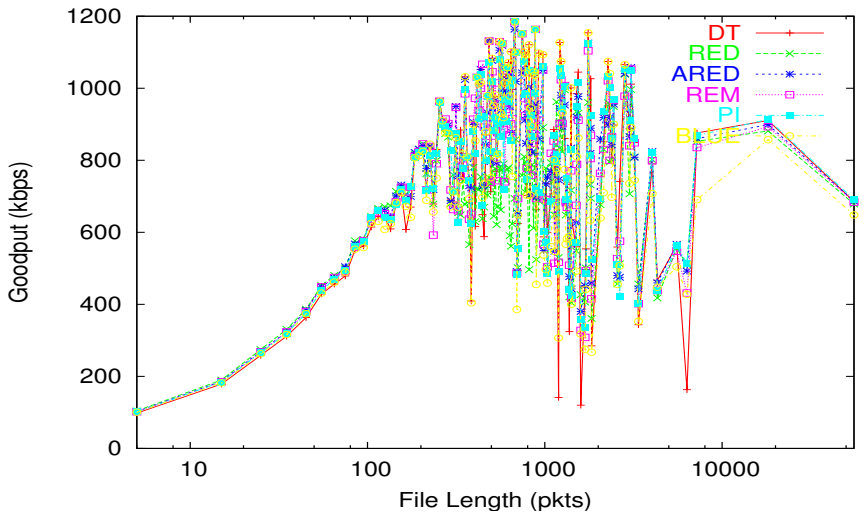


Figure 4.15: TCP goodput of traffic http1 with 50% traffic load

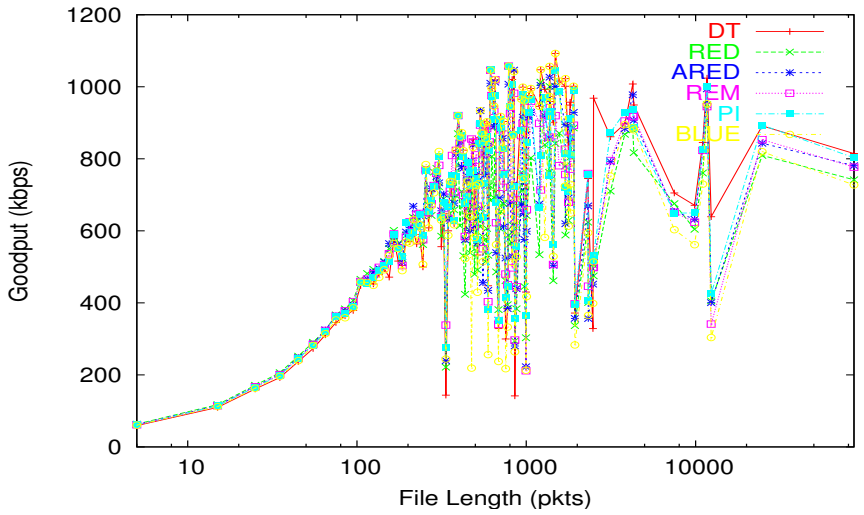


Figure 4.16: TCP goodput of traffic http2 with 50% traffic load

- Both Drop Tail and AQM obtain very similar network performance around 47% network throughput, negligible loss rate, and 50% link utilization as shown in Figures 4.17, 4.18, and 4.19, respectively with relative network throughput shown in 4.20 for comparison.

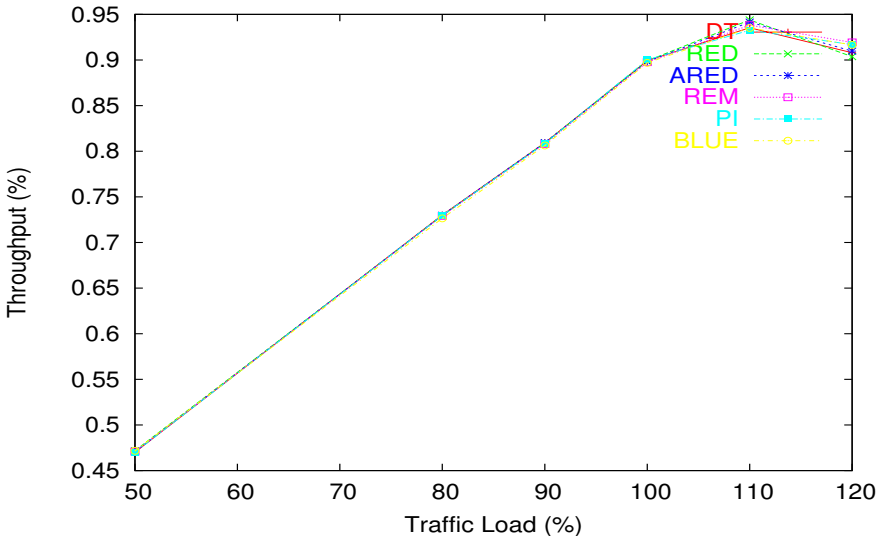


Figure 4.17: Average network throughput versus traffic loads

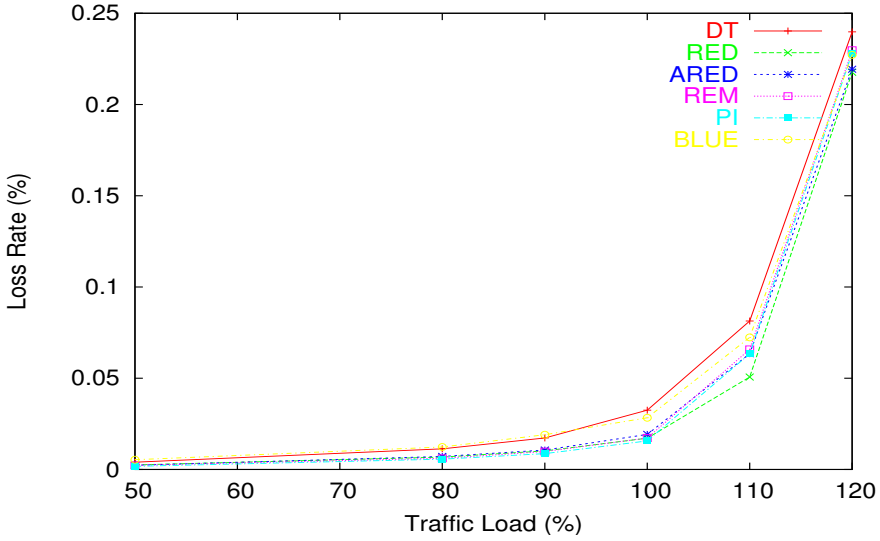


Figure 4.18: Average network loss rate versus traffic loads

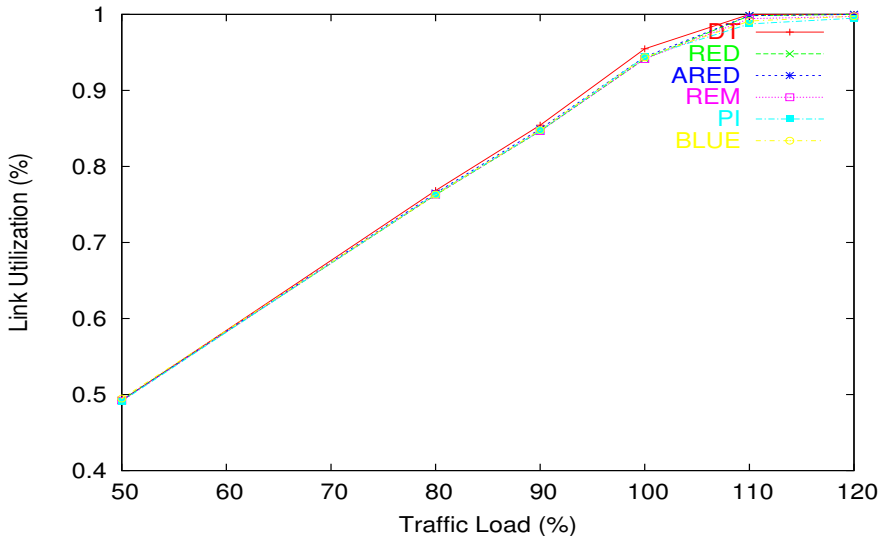


Figure 4.19: Average network link utilization versus traffic loads

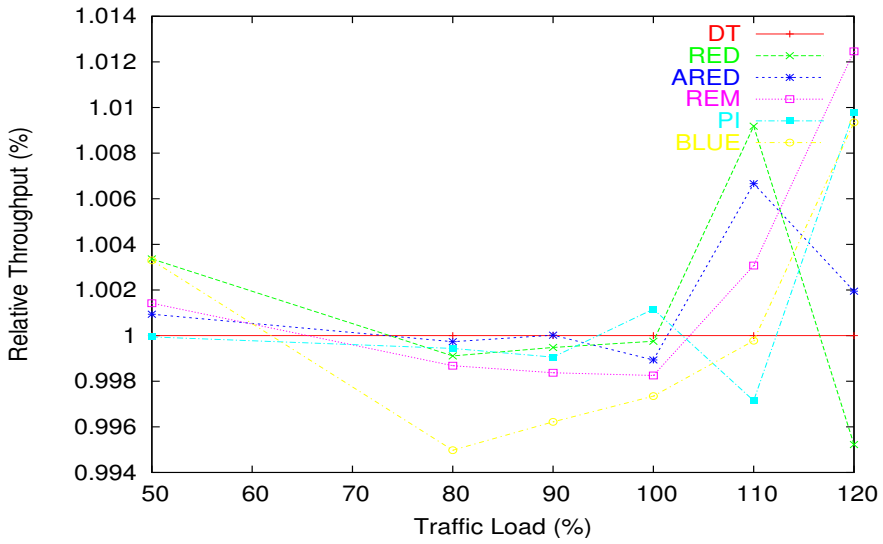
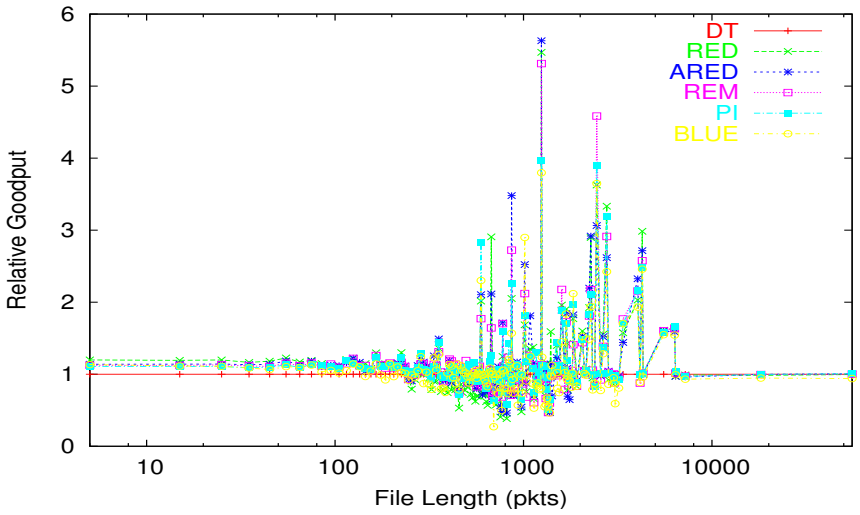
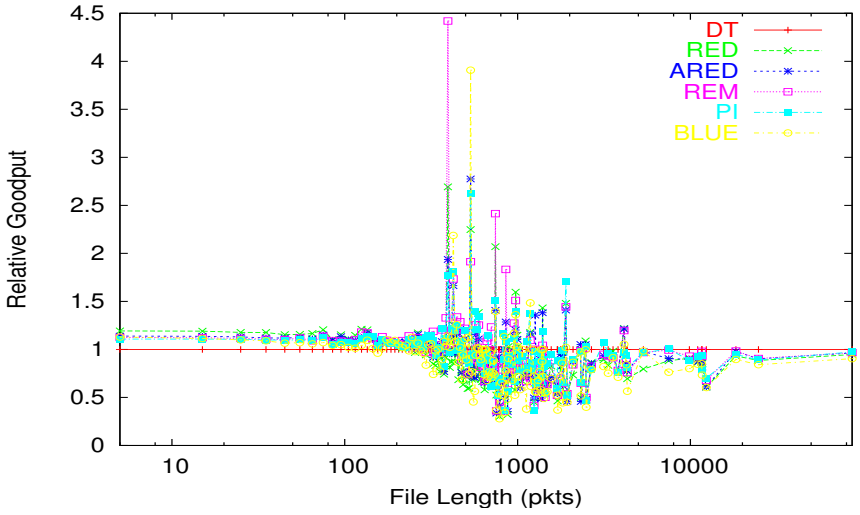


Figure 4.20: Average relative network throughput versus traffic loads

2. **Observations on medium traffic loads.** When the traffic load is medium, the Drop Tail scheme gains a similar performance comparison with AQM to that of the light load traffic condition. Thus with medium load traffic, there is no strong evidence that AQM outperforms Drop Tail, although the performance of mice are fairly improved on user goodput and response time as shown in Figure 4.21 and Figure 4.22, respectively.

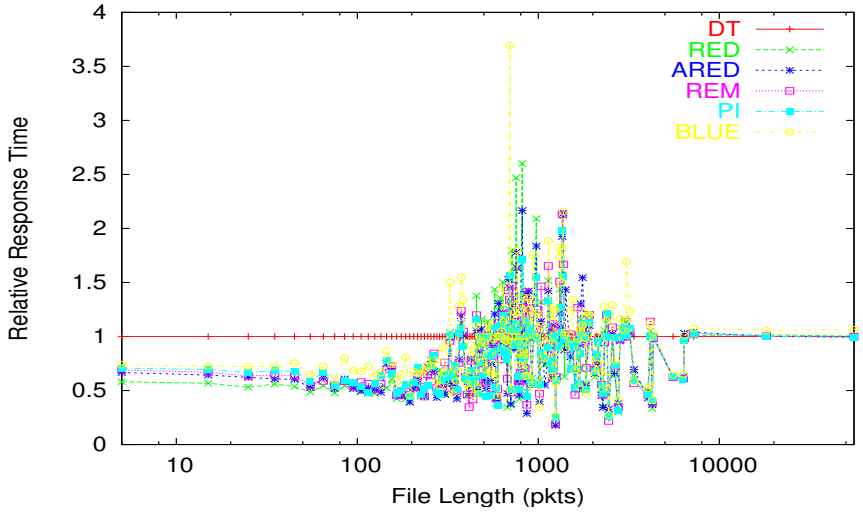


(a) Traffic http1

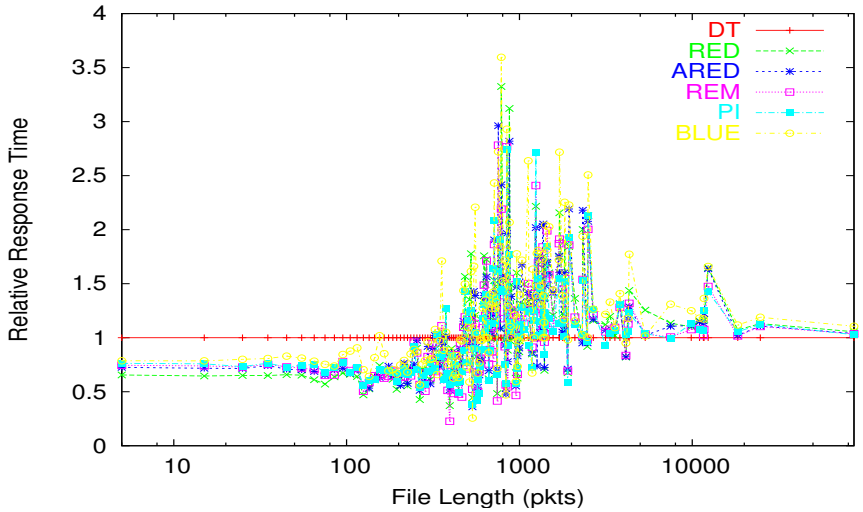


(b) Traffic http2

Figure 4.21: TCP goodput relative to Drop Tail with 80% traffic load



(a) Traffic http1

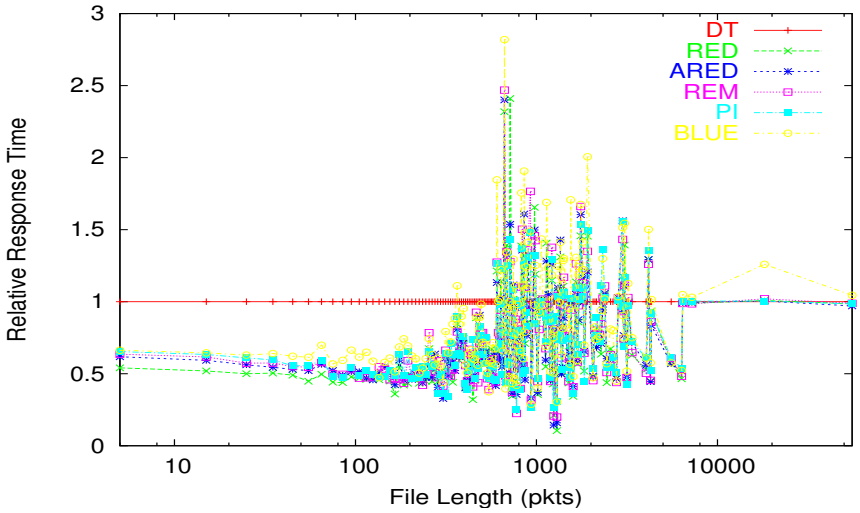


(b) Traffic http2

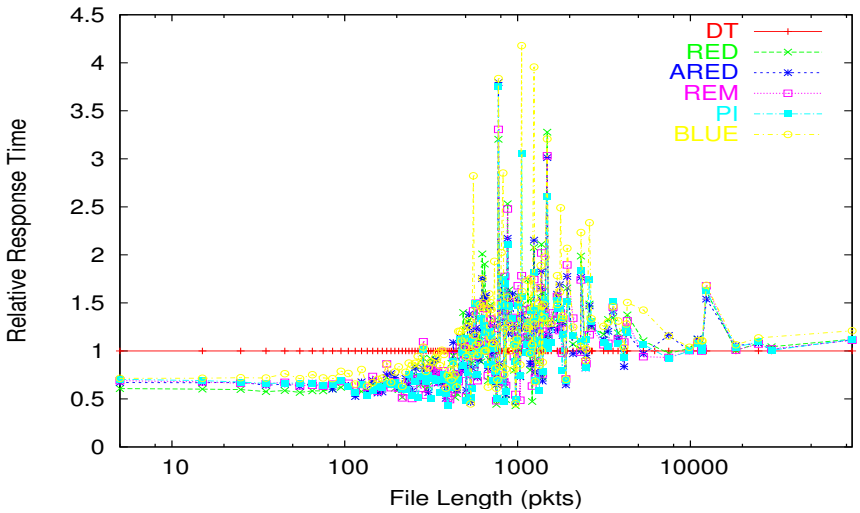
Figure 4.22: Response time relative to Drop Tail with 80% traffic load

3. **Observations on heavy traffic loads.** With heavy-loaded traffic, the advantages of AQM in improving performance emerge as follows.

   - The user response time is lower with AQM than with Drop Tail as shown by the weighted average response time in Figure 4.6, especially for the traffic of short RTT and short-lived flows as shown by the relative response time versus file lengths in Figures 4.23 and 4.24.

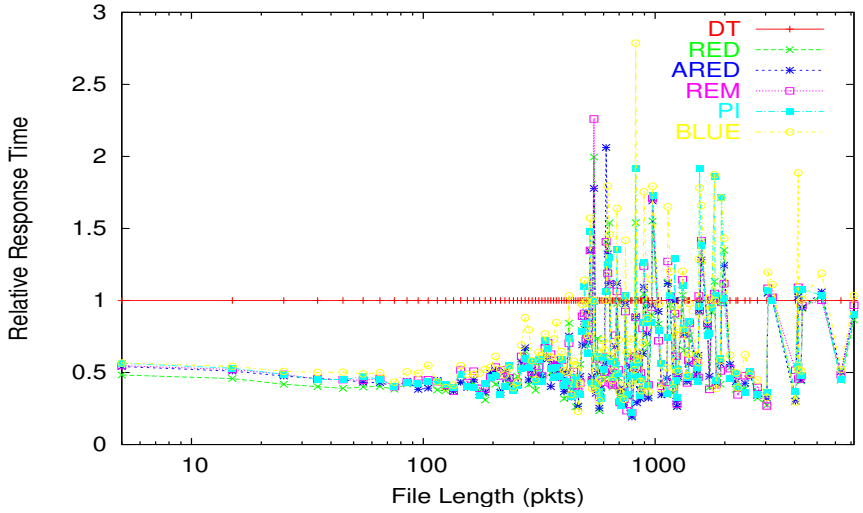


(a) Traffic http1

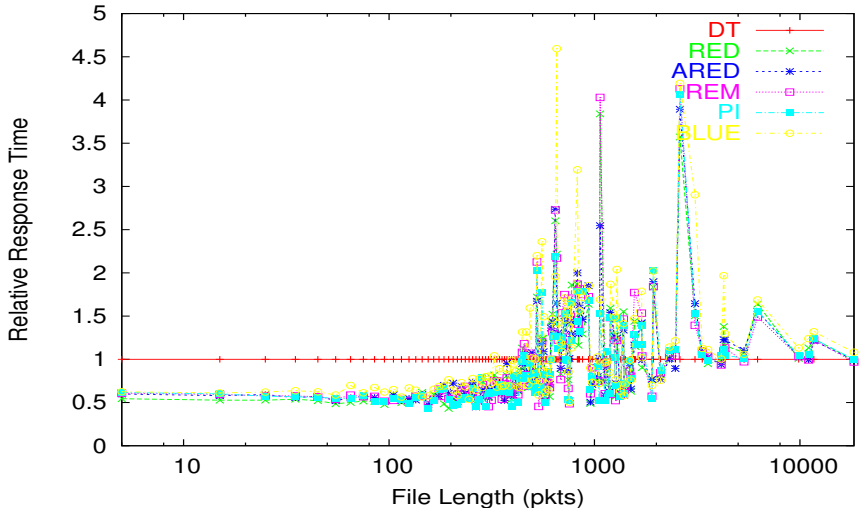


(b) Traffic http2

Figure 4.23: Response time relative to Drop Tail with 90% traffic load



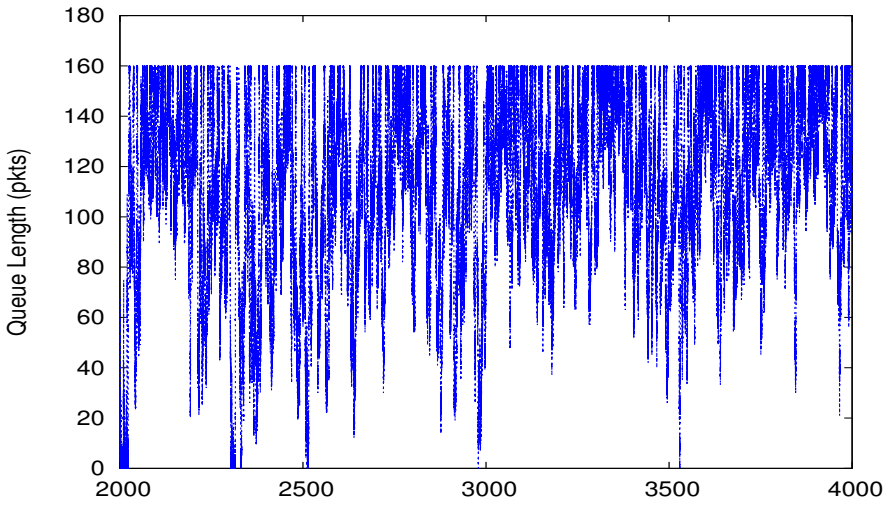
(a) Traffic http1



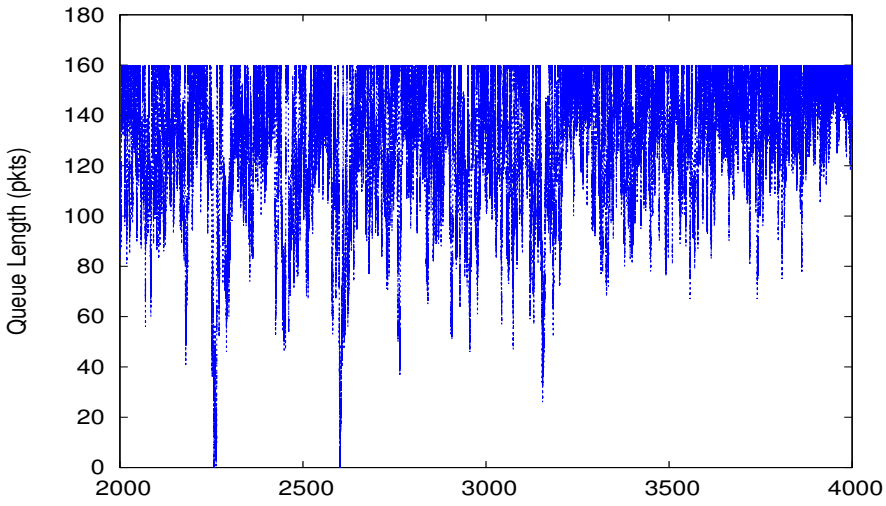
(b) Traffic http2

Figure 4.24: Response time relative to Drop Tail with 100% traffic load

The longer latency of Drop Tail results from high queuing delay in the bottleneck buffer as shown by the queue dynamics in Figure 4.25. By comparison, AQM tends to control the bottleneck queue around a predefined value or within a certain range as shown in Figure 4.26 by taking RED as an example.

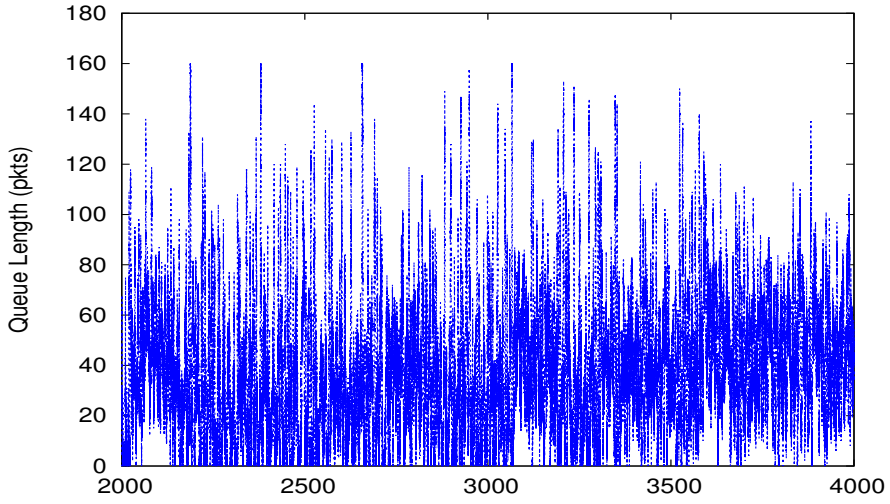


(a) 90% traffic load

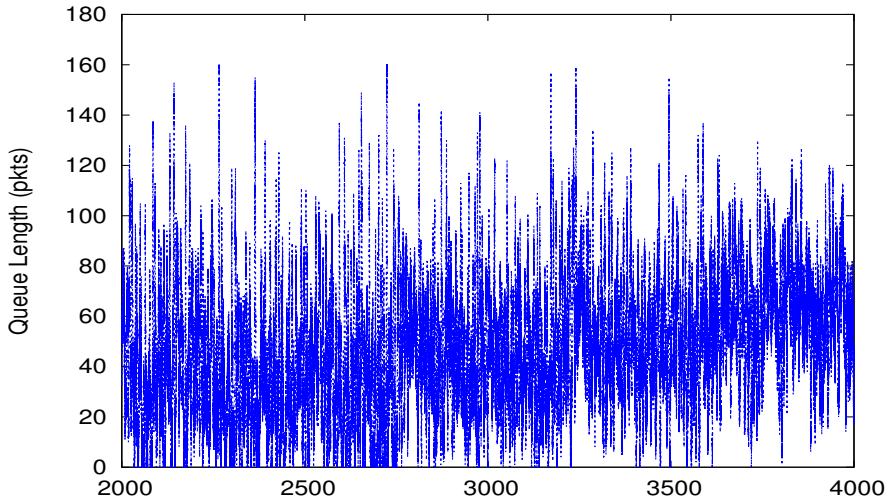


(b) 100% traffic load

Figure 4.25: Queue dynamics of Drop Tail with heavy-loaded traffic



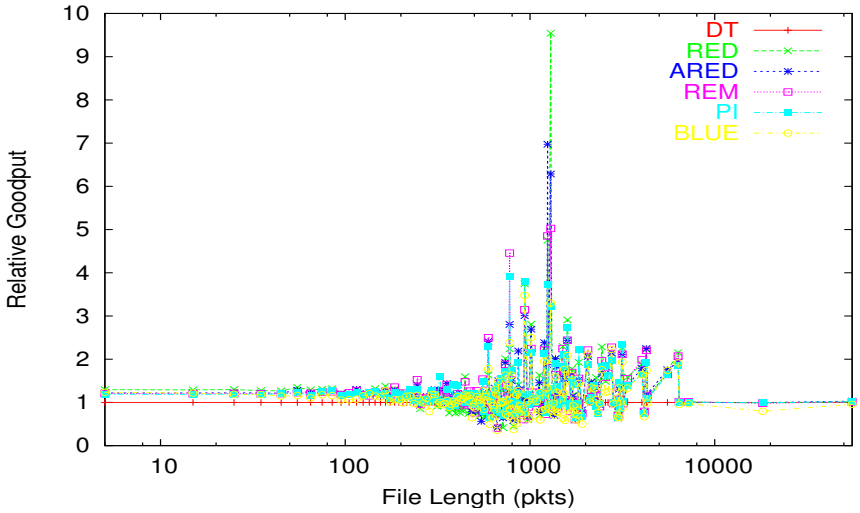
(a) 90% traffic load



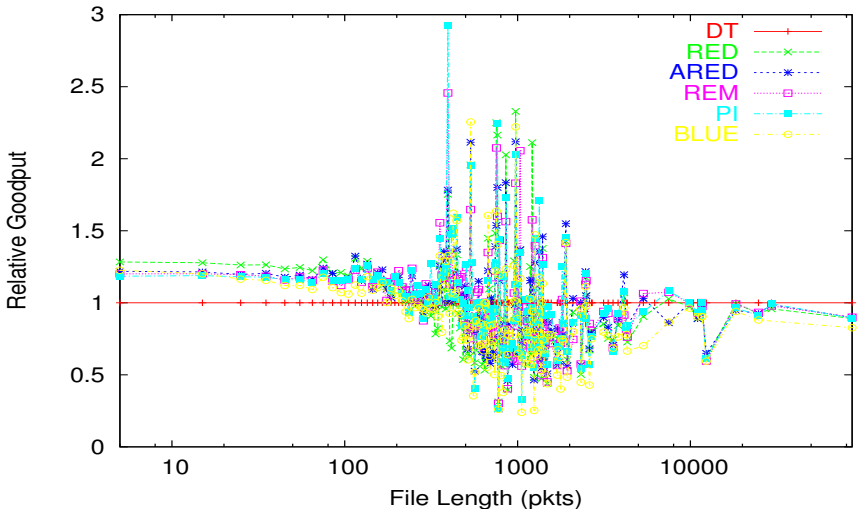
(b) 100% traffic load

Figure 4.26: Queue dynamics of RED with heavy-loaded traffic

- AQM outperforms Drop Tail on TCP goodput as shown by the weighted average TCP goodput in Figure 4.3 and by relative TCP goodput versus file lengths in Figures 4.27 and 4.28.

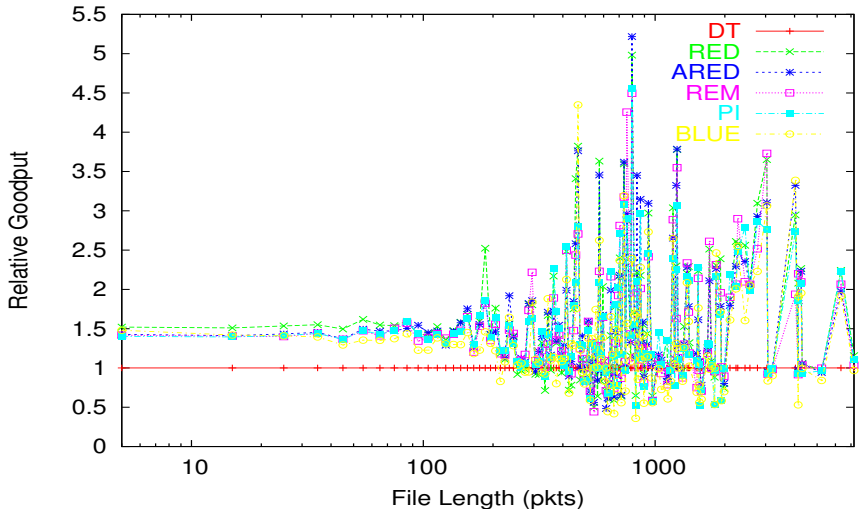


(a) Traffic http1

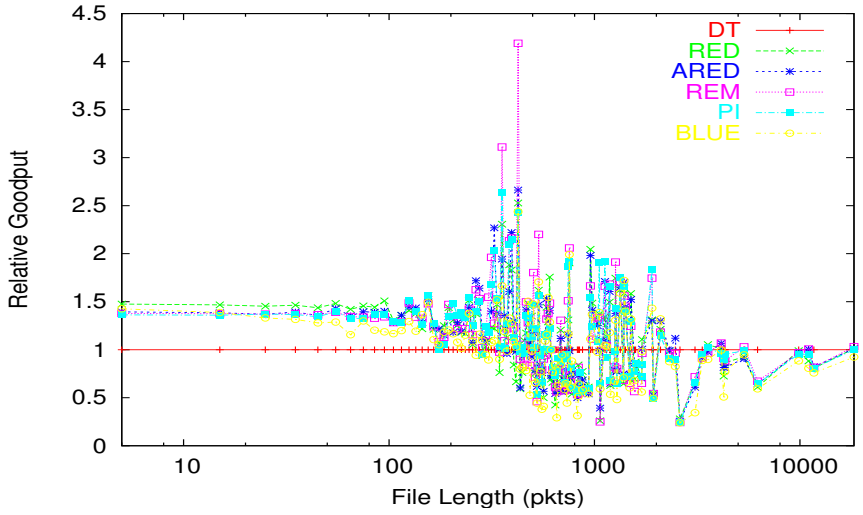


(b) Traffic http2

Figure 4.27: TCP goodput relative to Drop Tail with 90% traffic load



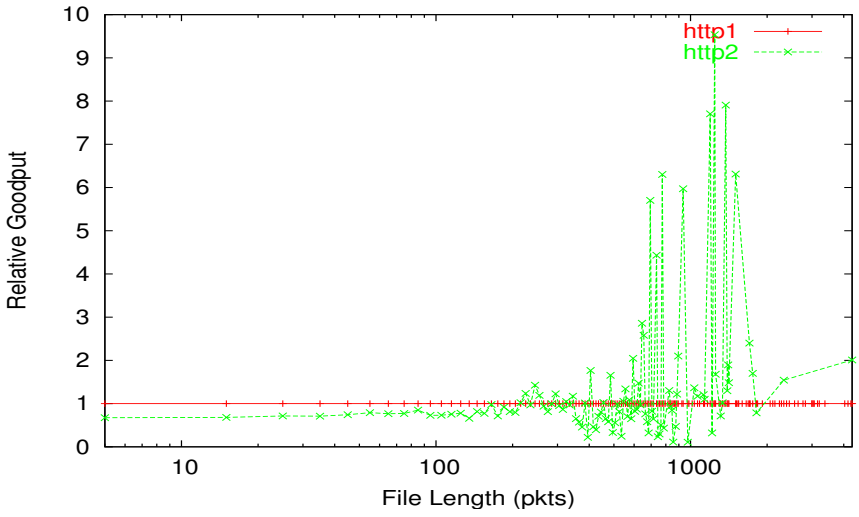
(a) Traffic http1



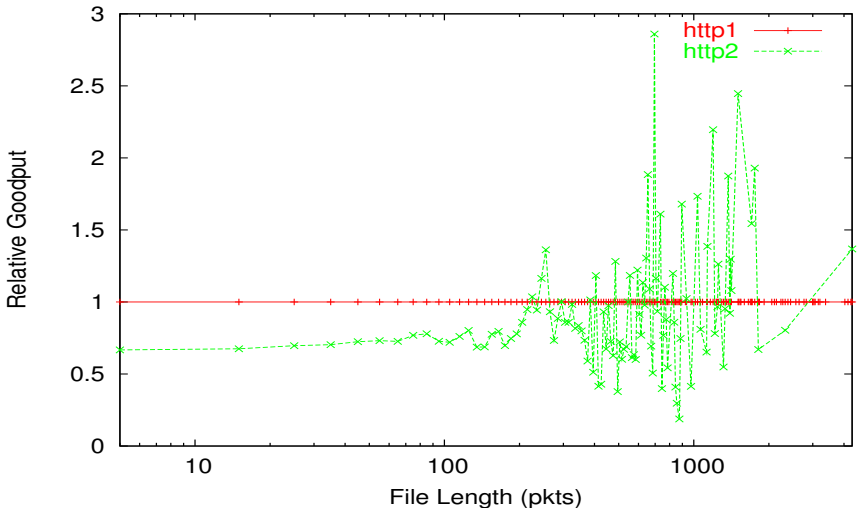
(b) Traffic http2

Figure 4.28: TCP goodput relative to Drop Tail with 100% traffic load

- Short RTT traffic still takes advantage of long RTT traffic to obtain higher performance in both TCP goodput and response time as shown in Figures 4.9, 4.10, 4.11,and 4.12. With the increase of traffic loads, however, the difference between long RTT traffic and short RTT traffic in goodput and response time is getting obscure and unpredictable. Figures 4.29 and 4.30 also demonstrate that with heavy-loaded traffic, the bias in TCP goodput against long RTT traffic might be enlarged by AQM, especially for short-lived flows.

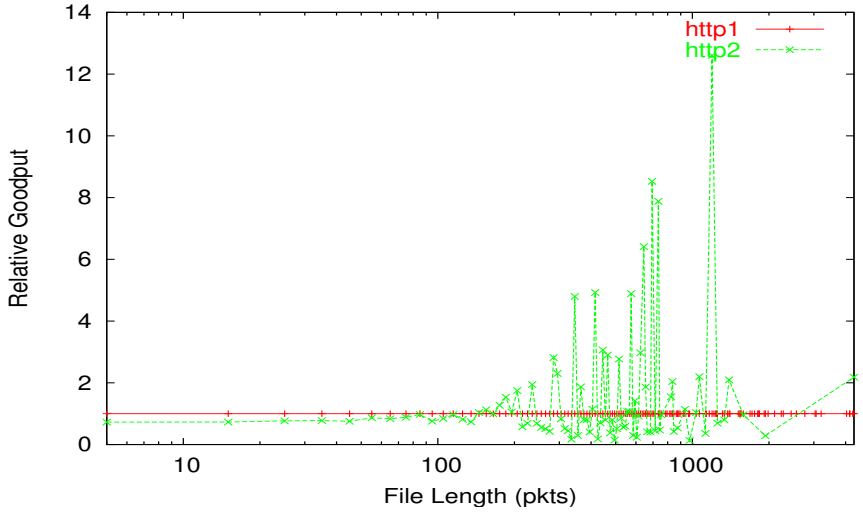


(a) Drop Tail

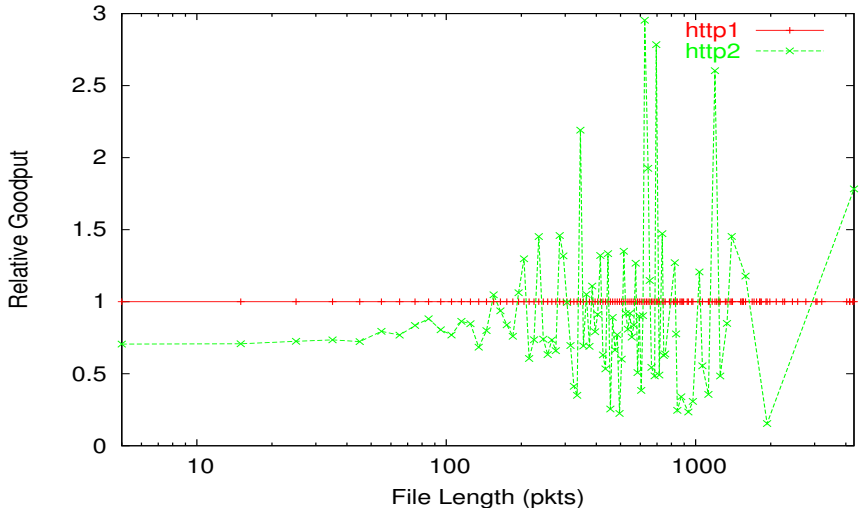


(b) RED

Figure 4.29: TCP goodput relative to traffic http1 with 90% traffic load



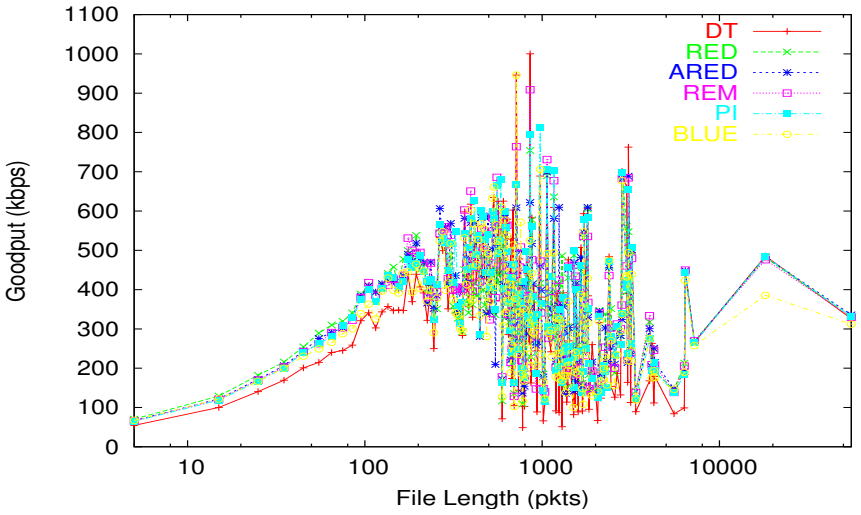
(a) Drop Tail



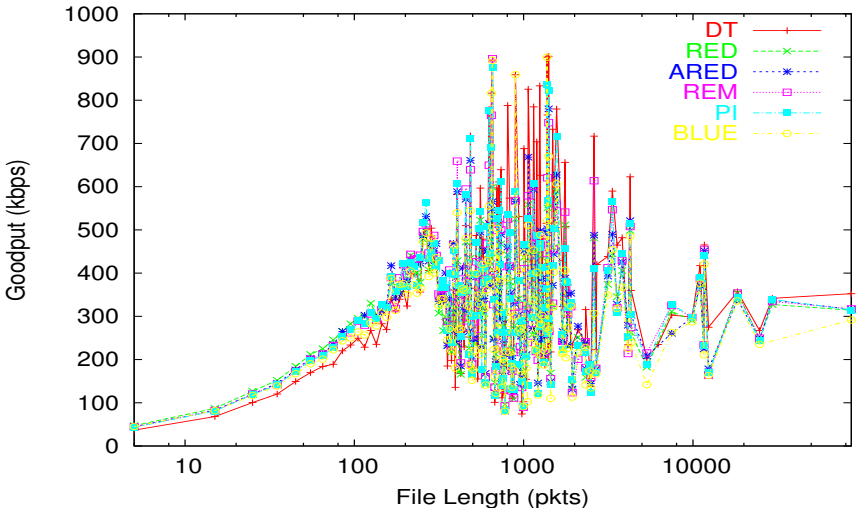
(b) RED

Figure 4.30: TCP goodput relative to traffic http1 with 100% traffic load

- Mice flows are still vulnerable, although AQM has improved their goodput performance as shown in Figures 4.31 and 4.32.



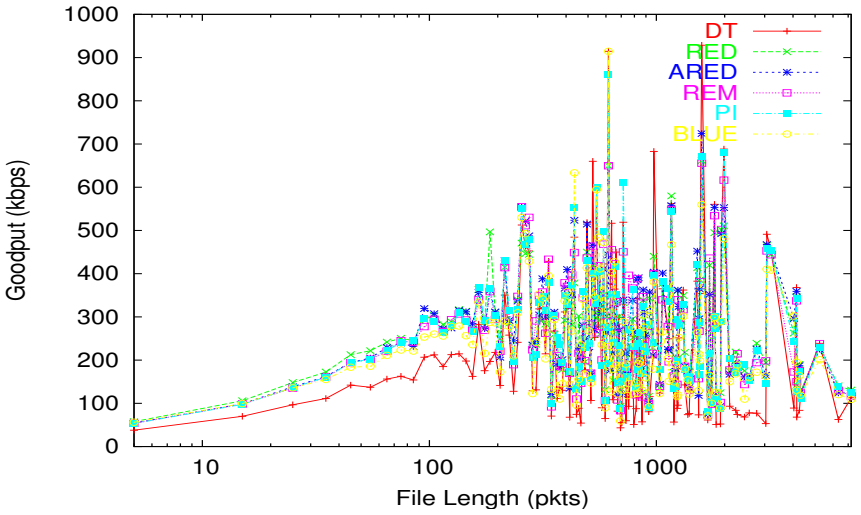
(a) Traffic http1



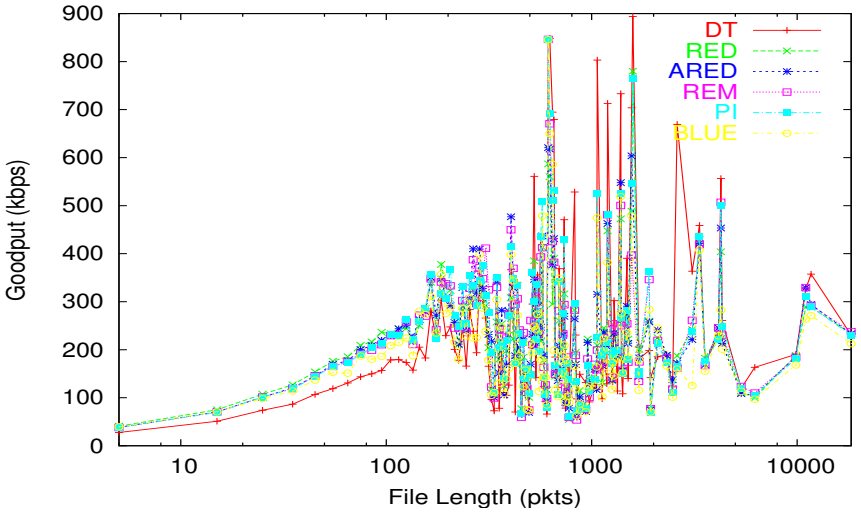
(b) Traffic http2

Figure 4.31: TCP goodput with 90% traffic load

- Network loss rate is decreased by AQM, while AQM and Drop Tail have similar performance on network throughput and link utilization as shown in Figures 4.17, 4.18, and 4.19.



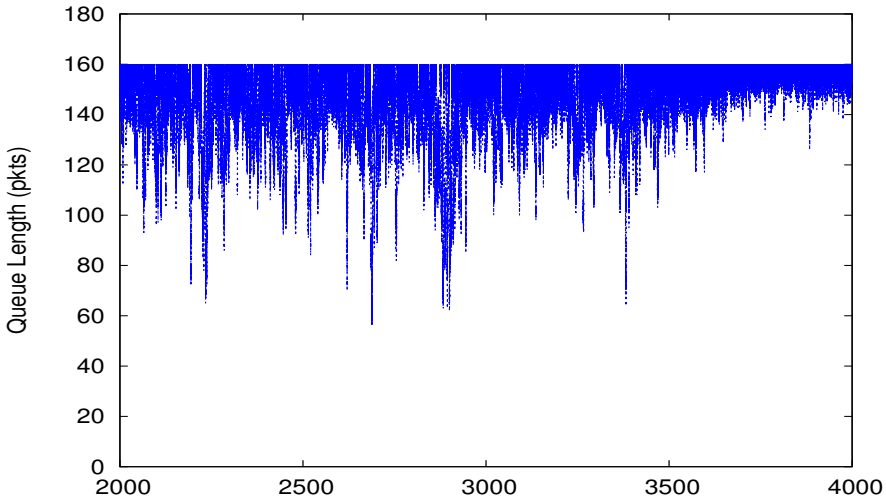
(a) Traffic http1



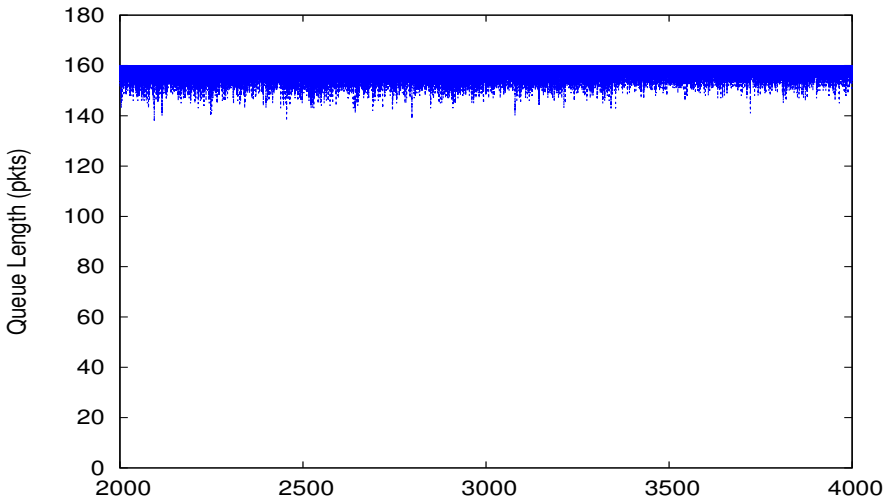
(b) Traffic http2

Figure 4.32: TCP goodput with 100% traffic load

4. **Observations on very heavy traffic loads.** When the traffic load is very heavy, the observations in the previous case of heavy-loaded traffic condition hold true. We emphasize that in this case an AQM scheme is specially needed to relieve congestion that causes a full buffer at the bottleneck as shown in Figure 4.33.



(a) 110% traffic load



(b) 120% traffic load

Figure 4.33: Queue dynamics of Drop Tail with very heavy-loaded traffic

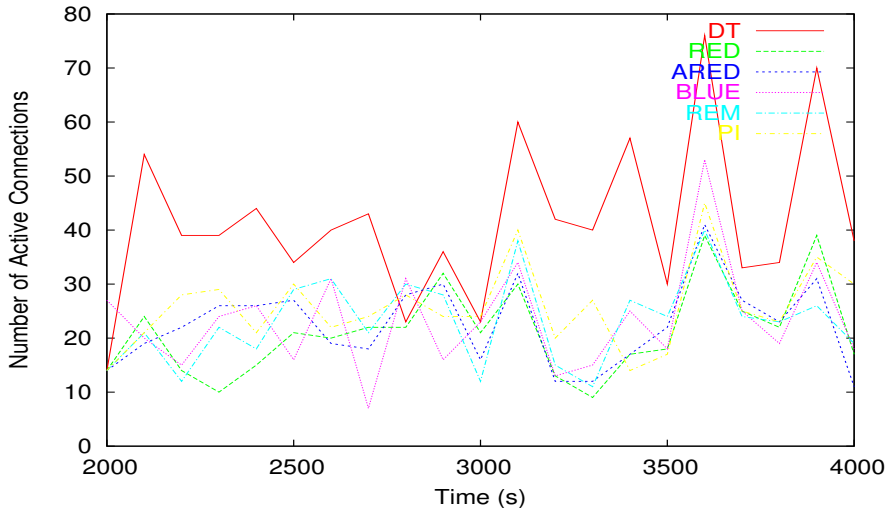
4.2.3 Result Analysis

Based on the simulation results, the following conclusions can be drawn.

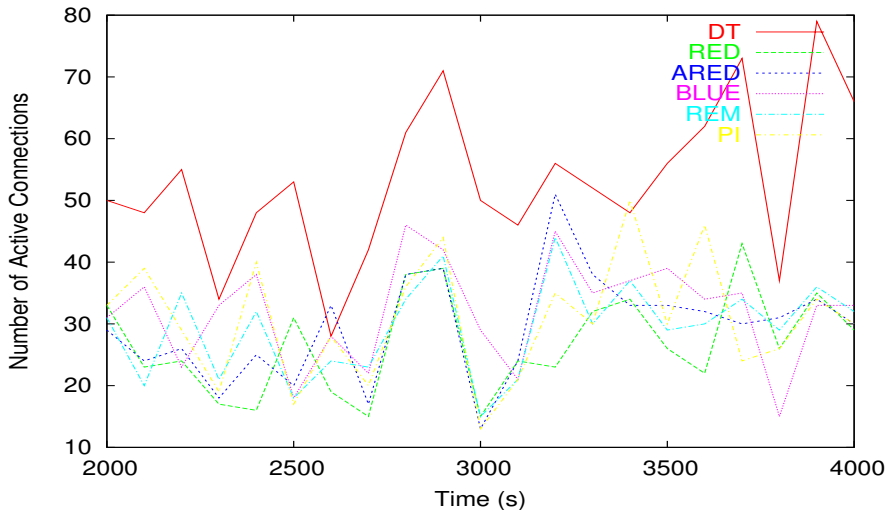
1. *First of all, it is becoming obvious that AQM outperforms Drop Tail on user TCP goodput and response time with increasing traffic loads.* AQM also improves the performance of mice. Despite these advantages for end users, the relative performance of the selected existing AQM schemes on the overall network throughput to the traditional Drop Tail strategy varies merely within a range of [-0.006, +0.013]. The reason is that these AQM mechanisms accommodate more mouse packets by mainly punishing elephant packets, whereas Drop Tail drops packets from both mice and elephants when the network is congested. Since mice take up a small fraction in bytes of the traffic but are large in numbers, the active connection number of these AQM schemes is lower than that of Drop Tail as shown in Figures 4.34 and 4.35 with no less than 90% traffic loads.

2. *Secondly, the poor performance of Drop Tail with heavy-loaded traffic, especially long queuing delay in the bottleneck buffer, has been demonstrated, although its performance is acceptable with light and medium traffic loads.* When the network is heavily loaded, the bottleneck buffer remains full (or almost full) which has considerable impact on packet queuing delay and TCP goodput. Although we have not been able to see global synchronization of Drop Tail and significant network throughput enhancement from the selected existing AQM scheme, an alternative queuing strategy other than Drop Tail is necessarily located at congested points in the Internet.

3. *Finally, the performance of the selected existing AQM schemes is sensitive to traffic load fluctuations existing in a traffic mix of short and long-lived traffic.* Although most of simulation studies have shown the efficiency of these proposed AQM mechanisms, these simulations assume long-lasting connections and a limited number of connections. With realistic traffic pattern made of much short-lived traffic and little long-lived traffic, traffic load fluctuates due to not only the TCP flow control and congestion control algorithms but also the emergence and disappearance of short-lived traffic, despite the average traffic load in a large

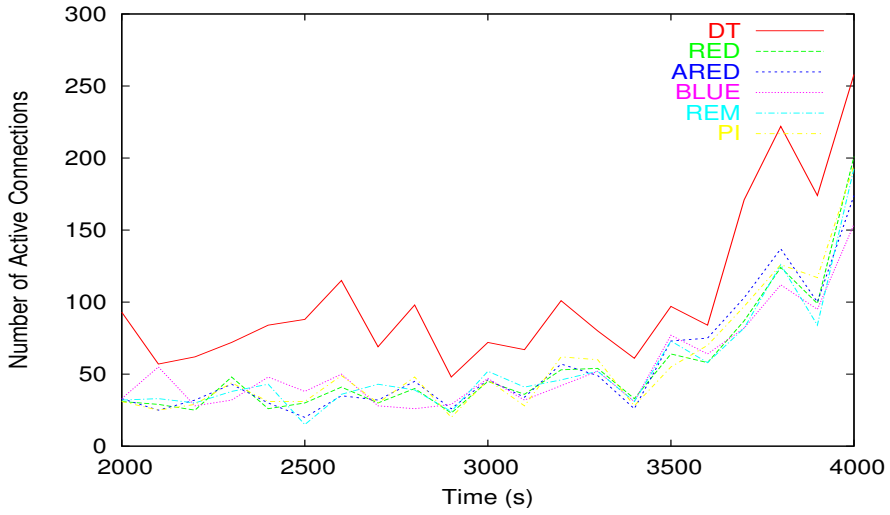


(a) 90% traffic load

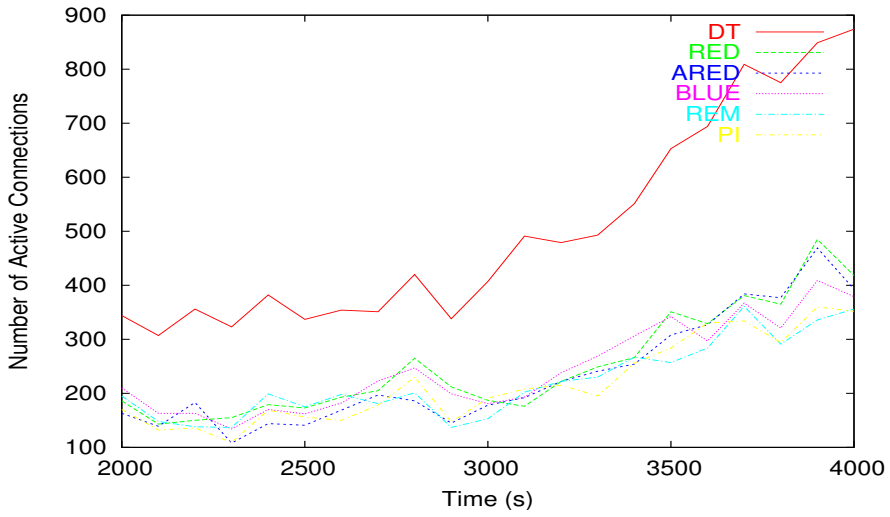


(b) 100% traffic load

Figure 4.34: Active connection number with different queuing strategies under the heavy-loaded traffic condition



(a) 110% traffic load



(b) 120% traffic load

Figure 4.35: Active connection number with different queuing strategies under the very heavy-loaded traffic condition

time scale. With our selected traffic pattern, these AQM schemes lack control stability in the traffic mix environment in the presence of large oscillations in queue dynamics as shown in Figures 4.36, 4.37, 4.38, 4.39, and 4.40 with 100% traffic load as examples.

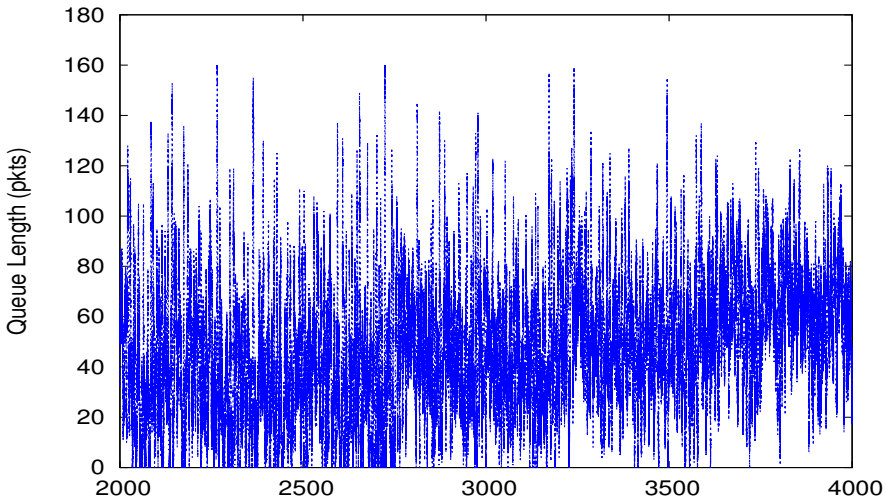


Figure 4.36: Queue dynamics of RED with 100% traffic load

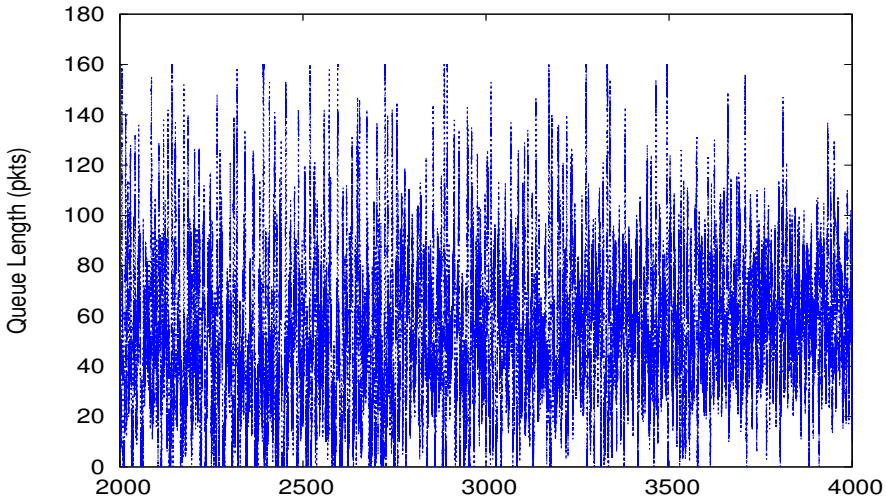


Figure 4.37: Queue dynamics of ARED with 100% traffic load

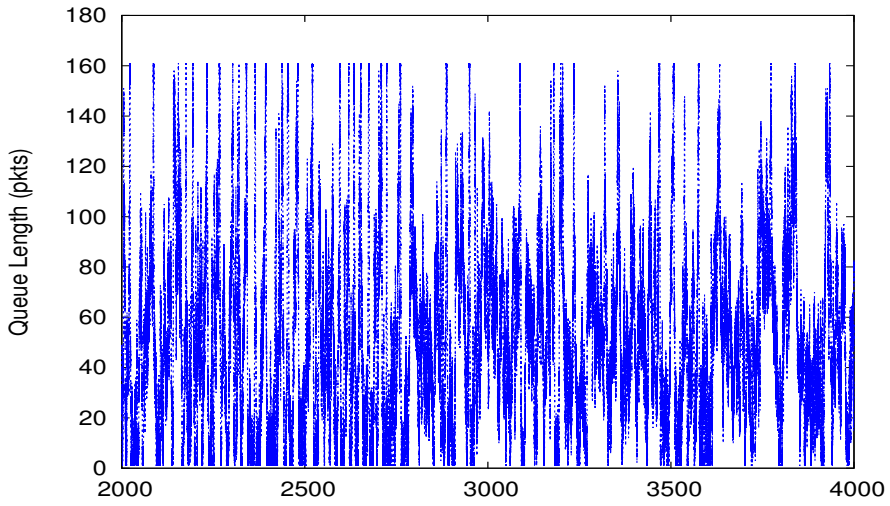


Figure 4.38: Queue dynamics of BLUE with 100% traffic load

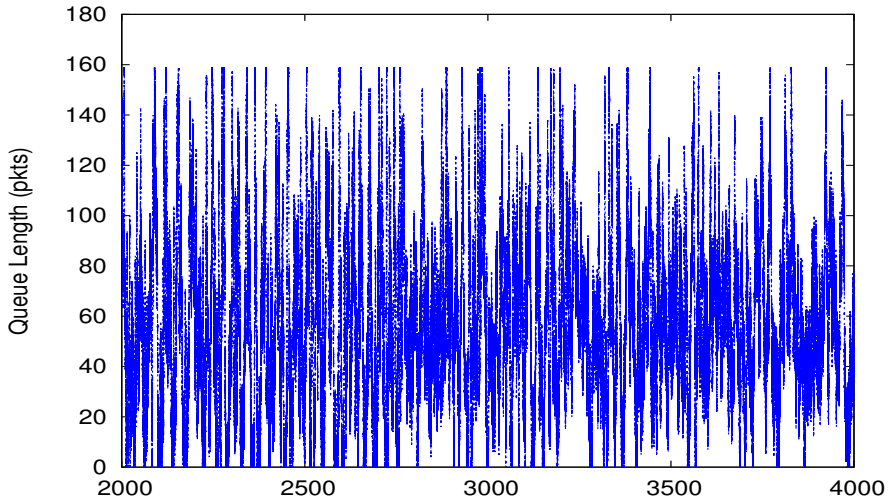


Figure 4.39: Queue dynamics of REM with 100% traffic load

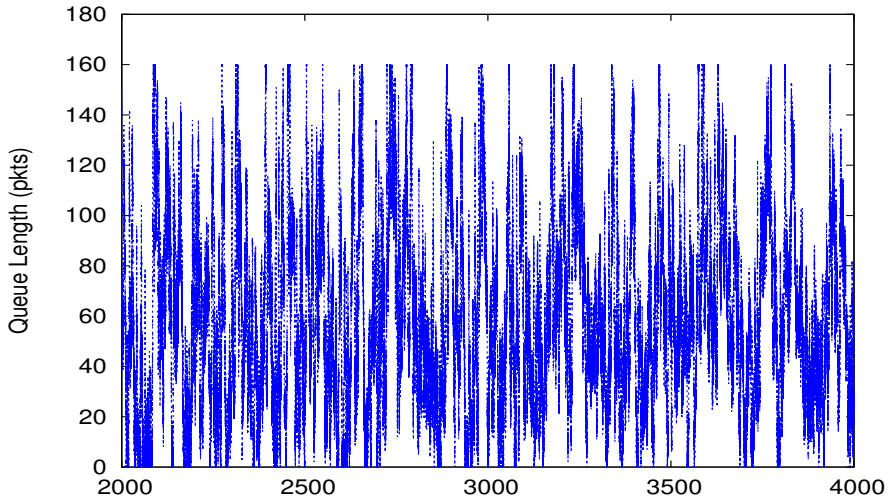


Figure 4.40: Queue dynamics of PI with 100% traffic load

4.3 Summary

We quantitatively investigate whether it is necessary to apply AQM in an Internet gateway. We have compared the performance of some existing AQM schemes against that of Drop Tail via extensive simulations with different traffic load conditions. The simulation results have been presented for light, medium, heavy, and very heavy traffic loads, respectively. The results shows that it is not necessarily beneficial to replace Drop Tail with AQM in light/medium traffic loads, but is beneficial when traffic load is getting heavier.

The status of light/medium traffic loads is in most cases true in the core network of the Internet [2], since it is well equipped and benefits from aggregation of traffic. However, congestion does frequently occur in the link between a private domestic or commercial premise and the Internet in reality. It is likely that such links will remain among the most cost-sensitive and bandwidth constrained components in the future Internet. Therefore, based on the simulation results, it is necessary to replace Drop Tail with an appropriate AQM scheme in such gateways to avoid or relieve congestion. One thing worth mentioning here is that the occurrence of traffic load more than 100% is rare, and an ISP needs to do re-engineering of the network capacity if the condition of very heavy traffic loads happens frenquently. We have used the very heavy traffic load condition in the simulations and just wanted to show the performance comparison of a number of AQM schemes and Drop Tail.

Despite the success of the existing AQM schemes, there is room to reach the AQM's potential of the simultaneous achievement of low delay and high network throughput, especially in a realistic traffic environment. The simulation results have shown the sensitivity of these methods to traffic load fluctuations in such a scenario. Therefore, congestion indicators that are able to reflect traffic load fluctuation and predict its trend are essential to an AQM design so that output queues remain short and absorb bursty traffic. Note that ECN marking is a solution for enhancing network throughput, but the approach is unable to obtain fundamental improvement in performance.

Another observation from the simulation results is that fairness remains unsolved,

[2]Being aware that some core networks, especially at the country boundaries, are heavily loaded. The possible solution for this kind of congestion is more complicated and is beyond the scope of this thesis.

particularly for short-lived flows or mice to obtain their bandwidth share in the condition of heavy-loaded traffic. Giving preference to mice might be a solution to the problem. These two topics, the need for more sophisticated design and implementation in AQM and the need for preferential treatment for mice flows, trigger the subsequent study in this thesis. Note that although the unfairness between short and long-RTT traffic has also been observed, the issue is out of the scope of this dissertation.

Chapter 5

Novel AQMs using Fuzzy Logic

Due to the dynamic nature and complexity of TCP congestion control, the AQMs reviewed in Chapter 3 leave some opportunity for improvement. Fuzzy logic (FL) has been used in various areas and achieved many successes. Research has revealed that there are a numbers of areas in traffic control where we can explore the use of artificial intelligence (AI) technologies such as FL. The objective of this chapter is to design novel AQM schemes which achieve efficiency and robustness by using AI technologies, in particular FL. In this chapter, we elaborate on the approach of developing AQM using FL. First, we present our AQM design and innovations in terms of the traffic load factor and the application of FL for AQM. After describing the structure of a generic FL controller (FLC) which directs an FLC design, the two proposed FL-based AQM (FLAQM) algorithms are then presented to realize proactive queuing in turn.

The analysis of the efficiency and feasibility of our proposed FLAQM algorithms is given in Section 5.4. The performance evaluation of the two FLAQM algorithms in comparison to that of Drop Tail and some widely accepted AQM methods mentioned in the previous chapter 3, is conducted via extensive simulations. Finally, the discussion of the parameter configuration of FLAQM is given.

5.1 Design Rationale

A common principle used by most existing AQM schemes is match-rate-clear-buffer, which implies the simultaneous achievement of low queuing delay and high link utilization. With this control principle in mind, we have also considered the other two issues

in AQM design: selection of congestion indicators and calculation of packet dropping probability.

5.1.1 Traffic Load Factor

Most existing AQM schemes have deployed queue length or input rate, or both, for congestion indication. Our approach is to design a new concept called *traffic load factor*. The traffic load factor, denoted as z, is the ratio of input rate to target capacity, where target capacity or expected traffic input rate is the leftover link capacity after draining the remaining packets in an output buffer. If input rate is measured at fixed intervals, the mathematical definition of z is given as follows.

$$z = \frac{input\_pkts}{target\_capacity} \tag{5.1}$$

$$target\_capacity = fraction \times link\_capacity \tag{5.2}$$

$$fraction = \begin{cases} \max(\gamma, 1 - \frac{Q - Q_0}{link\_capacity}) & Q > Q_0 \\ 1 & 0 \leq Q \leq Q_0 \end{cases} \tag{5.3}$$

$$link\_capacity = bdw \times dur \tag{5.4}$$

where $input\_pkts$ is the total input packets during the measurement period dur, γ is the maximum percentage of link capacity for draining the existing queue, bdw is the link capacity, Q is the instantaneous queue length at the end of the last measurement period, and Q_0 is a predefined target queue length. Note that Q_0 can be gained from a specified queuing delay by a network operator.

The traffic load indicator is a function of queue length and input rate. Using such a variable enables detection of impending congestion and reflects congestion severity. The adoption of the load factor as a congestion indicator is inspired by a successful rate-based scheme for controlling ABR flows in ATM networks in [30].

There are some advantages to using z as a congestion indicator. First, z is dimensionless so that a control algorithm based on such a measure is robust against link capacity changes [21]. Second, to gain z, target capacity is first computed by the capacity used for clearing the buffer subtracted from the link capacity, i.e. target_capacity

= link_capacity - capacity_for_clearing_buffer. The calculation of target capacity is easily extended to deal with the scenario where best-effort traffic coexists with other QoS traffic with reserved bandwidth, and available capacity for best-effort traffic is ever-changing.

5.1.2 The Application of Fuzzy Control for AQM

A TCP/IP network can be regarded as a feedback control system with a QM controller and the TCP/QM traffic plant as mentioned in Chapter 3. Control theory thus can be applied for the design of AQM. Some approaches based on classical control theory have been proposed such as PI [19] and VS [96]. It is pivotal for the schemes in this category to establish a mathematical model for TCP dynamics. However, there are some inherent limitations in the modeling process. Simplified models, such as through linearization, are required by certain control technologies. Some assumptions, such as exclusive long-lived connections and a delay-free control system, apply. Additionally, some parts of TCP dynamics are ignored, for example the slow-start phase and timeout of TCP. Therefore, any classical control-theory-based approaches potentially fail to achieve good performance and system stability. The weakness of classical control theory in the design of AQM is due to the complex and nonlinear nature of the control system and the difficulty in mimicking the dynamic behavior of the TCP/AQM plant in the form of any sound mathematic models.

A TCP/IP network is undoubtedly a complex nonlinear system. The complexity and nonlinear features result from many factors including the nonlinear property of TCP dynamics, dynamic traffic mix, and network heterogeneity ranging from the RTTs and the life-time a connection is experiencing, to protocols at each network layer, to different TCP versions. Given such a complex nonlinear control, FL is a better alternative control solution. In fuzzy control theory, nonlinearity is handled by rules, membership functions, and the inference process (these concepts will be given later), which results in improved performance and system stability with simpler implementation and reduced design costs. The idea behind FL is to emulate the cognitive inference process that human beings deploy in their decision-making and problem-solving in a way that input signals stimulate logic inference utilizing human heuristic knowledge and certain actions are taken accordingly. FL thus is able to apply expert knowledge

to solve complex nonlinear problems without the need for precise and comprehensive information and the mathematical model of controlled objects.

Many applications have been found for the use of FL for traffic/congestion control in computer networks. For instance, in ATM networks, FL approaches range from ABR flow control [58], to frame discard mechanisms [71], to policing [32, 15, 70]. However, to our best knowledge, there are only a few studies in AQM using FL [17, 38] (refer to Chapter 3 for detail).

Combining the use of the traffic load factor for congestion notification and FL to yield packet dropping probability, two FLAQM algorithms are derived. The first proposed FLAQM, FLAQM(I), uses z as an input directly, while the second FLAQM(II) uses its reciprocal z' instead. In addition, the changes of z and z' are used to capture traffic load trend in FLAQM(I) and FLAQM(II), respectively. The intention of using the reciprocal of z in FLAQM(II) is to implicitly realize the input normalization (see the following section for reference) to achieve system stability and robustness.

5.2 A Generic FLC

The idea behind FLCs is to provide a means of converting a linguistic control strategy based on domain expert knowledge into an automatic control strategy in environments where either the processes are too complex for analysis by conventional quantitative techniques, or the available sources of information are interpreted qualitatively, inexactly, or uncertainly [68, 69]. A generic FLC comprises four building blocks: (1) a fuzzification interface, (2) an inference engine, (3) a knowledge base, and (4) a defuzzification interface as shown in Figure 5.1. Note that in practice the values of the process states inputting to an FLC are crisp and the control outputs also require a crisp value. The functionalities of each block are given as follows. For more details refer to [14, 62, 68, 69].

- The fuzzification interface performs two functions including scaling and fuzzification.
  - It performs a scale transformation or an input normalization, which maps the physical values of the process state variables or input variables into a nor-

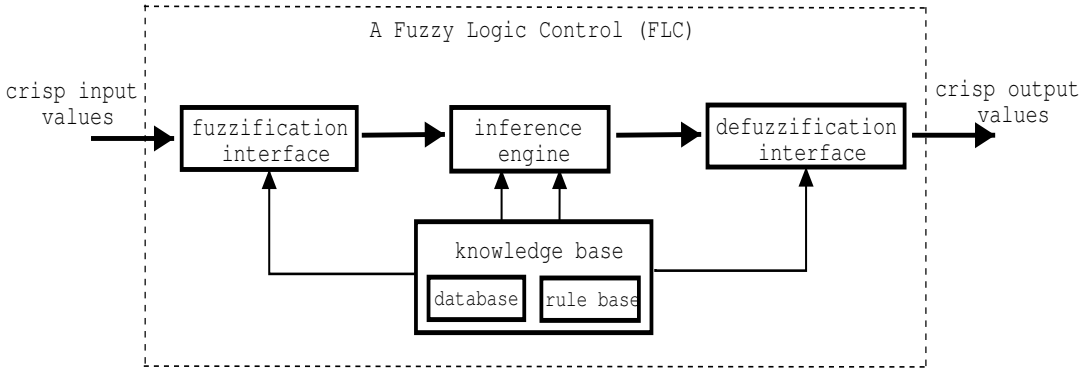


Figure 5.1: The structure of a generic FLC

malized universe of discourse (normalized domain). When a non-normalized domain is used then there is no need for the scaling function.

– It performs the function of fuzzification that converts input data into a fuzzy set F, characterized with a membership function μ_F so that every element u from the universe of discourse U has a membership degree $\mu_F(u) \in [0, 1]$ to F. One widely adopted fuzzification strategy is 'singleton fuzzification', which produces a fuzzy set F for a crisp input u_0 with a membership function $\mu_F(u)$ defined by

$$\mu_F(u) = \begin{cases} 1, & u = u_0 \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

- The knowledge base provides a rule base and a database.

  – A rule base contains a number of fuzzy rules, a linguistic description of domain expert knowledge in the "if-then" form such as *if x is A, then y is B*, where x and y are linguistic variables whose values are words or sentences in a natural or artificial language, and A and B are one of the language values that x and y can take, respectively. For instance, an error between a process state and a desired value or set-point is a linguistic variable if its values are linguistic rather than numerical, i.e., Negative Big (NB), Negative Medium (NM), Negative Small (NS), Zero (ZO), Positive Small (PS), Positive Medium (PM), and Positive Big (PB) etc., and these linguistic values denote a symbol for a particular property of the variable of error that a

domain expert normally describes. Note that a linguistic value corresponds to a fuzzy set, which quantitatively interprets the linguistic value by a membership function. The Triangle and trapezoid shapes are among the popular choices for the membership function of a linguistic value since they usually need less computation.

– A database defines the membership functions of the fuzzy sets used in the fuzzy rules and scaling factors used in normalization and denormalization.

- The inference engine performs the fuzzy inference operations.

  The fuzzy inference operations are based on the selection of a function of a fuzzy implication expressing each individual fuzzy control rule and a compositional rule of fuzzy inference operating between fuzzified inputs and the aggregation of all the fuzzy implications of the rule set. The outcome of the inference engine is a fuzzy set. The minimum operation and the product operation are the two most popular rules of fuzzy implication, while *sup-min* and *sup-product* are two commonly used compositional operators.

- The defuzzification interface performs functions just the opposite to those of the fuzzification interface.

  – It performs the function of defuzzification that converts control output values from a fuzzy set to a script value. There are some defuzzification strategies used in fuzzy control applications. A crisp result y_0 corresponding to fuzzy output B_0 is given by the formula of one commonly used method, Center of Area (COA) in a continuous manner as follows.

$$y_0 = \frac{\int \mu_{B_0}(y) y dy}{\mu_{B_0}(y) dy} \tag{5.6}$$

  – It performs an output denormalization that maps the crisp output value into its physical domain. This function is not needed if non-normalized domains are used.

Note that the combination of 'singleton fuzzification', the minimum operation rule of fuzzy implication, and *sup-min* compositional operators, frequently used in some

successful fuzzy control applications, results in the simple inference procedure of an FLC as shown in the following. The graphic interpretation of the procedure of a two-input-one-output FLC is given in Figure 5.2 with the additional illustration of the COA defuzzification strategy. The same procedure followed by the COA defuzzification strategy is adopted in FLAQM. Note that in Figure 5.2, triangle shape is used for membership functions.

- Match the crisp input values with the membership functions on the premise part of each individual fuzzy rule to obtain the membership degrees of each linguistic value (such as a1 and b1 for R1, a2 and b2 for R2 in Figure 5.2).
- Obtain the minimum among the membership values on the premise part to get the firing strength of each rule (such as b1 for R1 and b2 for R2 in Figure 5.2).
- Clip the fuzzy set describing the meaning of the consequent part to the degree of the firing strength (such as the boundary lines of the region of A1 for R1 and those of the region of A2 for R2 shown in red in Figure 5.2).
- Aggregate the clipped values for the control output of each rule via maximization in the universe of the output linguistic variable, and takes the aggregation as the overall control output (such as the boundary lines for the region of A1 $\cup$ A2 shown in red in Figure 5.2).

5.3 FLAQM

The FLAQM controller is designed to conduct queue management in a bottleneck output queue and control the behaviour of the closed-loop feedback system as shown in Figure 5.3. FLAQM calculates dropping probability pr based on the measurements from feedback signals. Two FLAQM algorithms are proposed in this study, namely FLAQM(I) and FLAQM(II). FLAQM(I) directly uses traffic load factor z and its change Δz as inputs, whereas FLAQM(II) inputs the values of the reciprocal of z, z' and its change $\Delta z'$. Note that valid feedback from the TCP/IP networks is delayed for at least one RTT for each connection. After packets reach their destinations, the corresponding ACK (acknowledgment) packets are received by their sources, and the

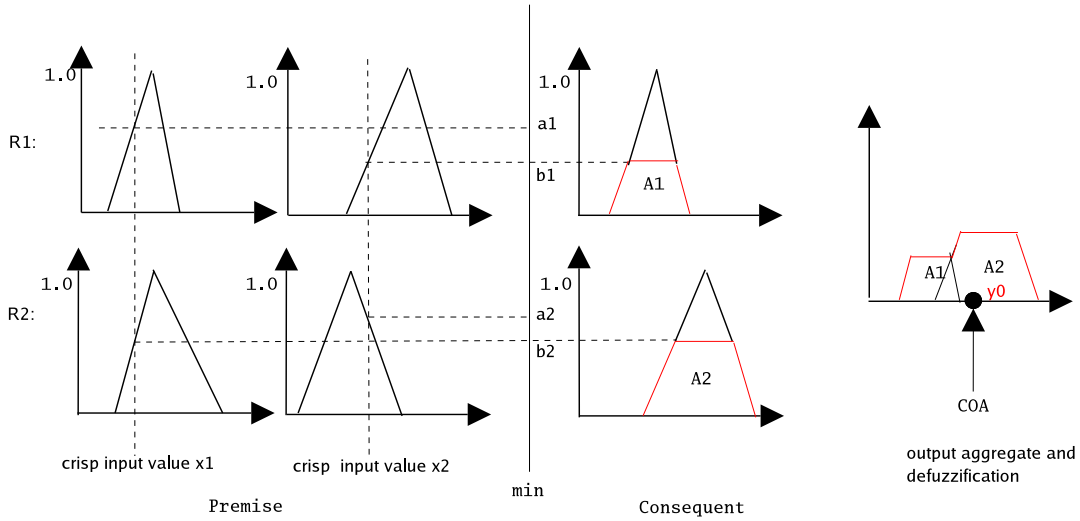


Figure 5.2: The inference procedure of a commonly used FLC

sources respond to the dropping probability *pr* to adjust the amount of packets sent to the network. Therefore, FLAQM is time-based in that measurements of feedbacks or the input values of the FLC and subsequent computation of the dropping probability *pr* are carried out in a fixed time interval. An appropriate value for the interval is carefully chosen to allow for transient traffic conditions with the arrival of bursty data on the one hand and the ability to react quickly to impending congestion on the other hand. Here the interval is set to be at least the maximum RTT of all the active connections to avoid system oscillation. The two proposals of FLAQM are presented in the following.

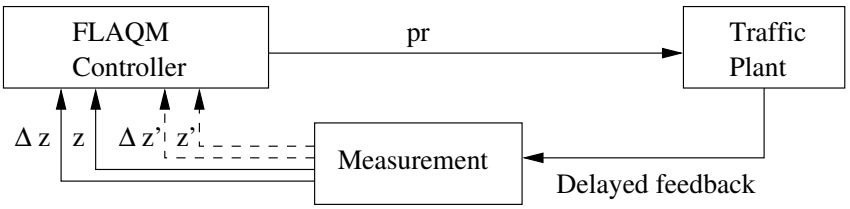


Figure 5.3: The closed-loop feed back system with the FLAQM controller

5.3.1 The FLAQM(I) Controller and Traffic Load Factor

FLAQM(I) aims to determine an appropriate value by which the routers drop the incoming packets based on the feedback information about traffic load and its trend. More specifically, the input variables of the FLAQM controller are traffic load factor z and its change Δz.

It is clear that the set-point for the measured plant output, z, is 1 in that the input rate equals the target link capacity. Thus, the steady-state operating region toward which the FLAQM attempts to drive the closed-loop feedback system is in the neighborhood of $z = 1$. In order to achieve high link utilization of the network, the neighborhood of $z = 1$ is set as the range of $[1, 1 + \delta]$, where δ is a constant.

The FLAQM controller copes with three cases in the network as shown in Figure 5.4. If load factor z is beyond the set-point of the system, multiplicative decrease (MD) action is taken with negative Δz by using the MD_FLAQM controller, while additive increase (AI) is applied with positive Δz by using the AI_FLAQM controller. Otherwise, the traffic in the network is not overloaded and thus it is not necessary to do any extra control, but to add the incoming packets in the output queue. We have tried other designs, such as AIAD. However, the AIMD design appears to be more steady. The pseudo-code of dropping probability calculation in FLAQM(I) is given in Algorithm 5.1.

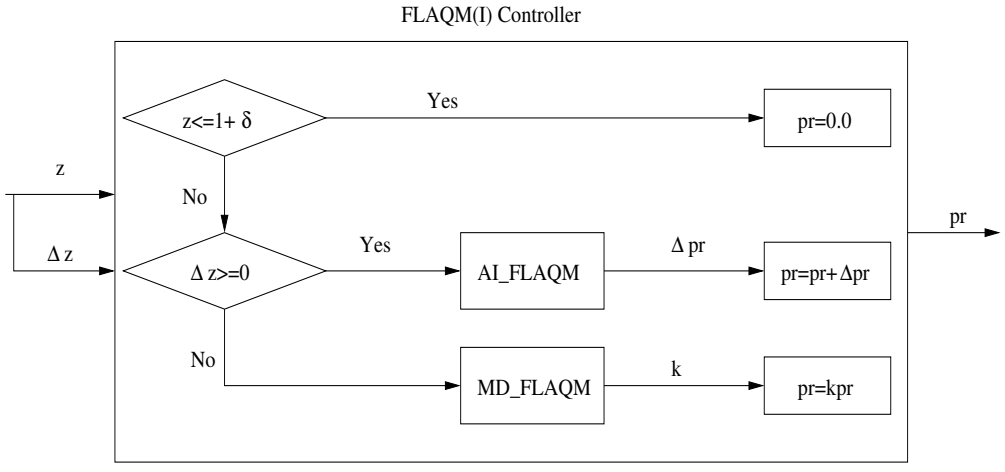


Figure 5.4: The structure of FLAQM(I)

Algorithm 5.1 Dropping probability in FLAQM(I)

```
#pr is dropping probability
if (z <= 1 + δ)
        pr = 0.0
else
        if (Δ z < 0)
                k = MD_FLAQM(z, Δ z)
                pr = k * pr
        else
                Δ pr = AI_FLAQM(z, Δ z)
                pr = pr + Δ pr
        endif
endif
```

In the next two subsections, the membership functions and the fuzzy rules of these two FLCs: MD_FLAQM and AI_FLAQM, will be introduced. Note that the *sup-min* FL inference and the *center of gravity* defuzzification methods are adopted [14], since they are widely used and also proved to be effective in practice.

5.3.2 Design of two FLCs in FLAQM(I)

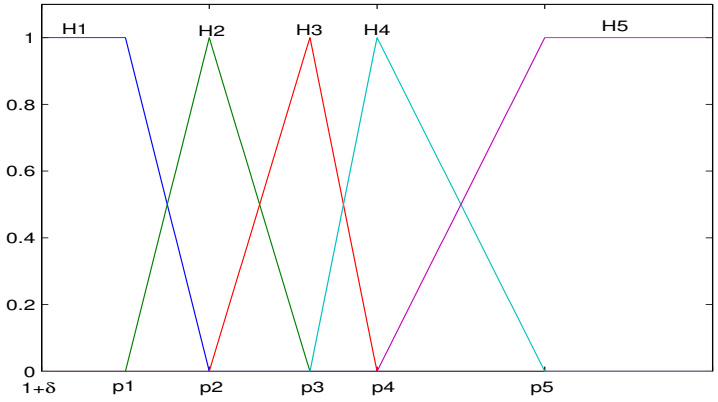
There are a set of membership functions for each FLC. For simplicity and effectiveness, triangular and trapezoidal shapes are chosen for these membership functions. Both FLCs use load factor z and Δz as inputs. For the MD_FLAQM controller, the output variable is coefficient k for a multiplicative decrease of pr, whereas for the AI_FLAQM controller, Δpr is the output variable for an additive increase of pr. Figure 5.5 shows the membership functions used in the MD_FLAQM controller, whereas Figure 5.6 shows the membership functions of the AI_FLAQM controller. The linguistic values of the input z are $Hi, i = 1, 2, \cdots, 5$, and traffic load increases with i. The input Δz is characterized by $Ni, i = 1, 2, \cdots, 5$ where negative N specifies that the current traffic load has decreased when compared with its previous value and its magnitude increases with i, and $Pi, i = 0, 1, \cdots, 5$ where positive P specifies that the traffic load is getting heavier than before and its magnitude increases with i. For the MD_FLAQM controller, the output k is described by $MDi, i = 0, 1, \cdots, 5$ increasing with i, while the linguistic values of the output Δpr in the AI_FLAQM controller are $AIi, i = 0, 1, \cdots, 5$ also increasing with i. All the membership functions are characterized by their own shapes

(triangle or trapezoid) and parameters indistinguishably denoted as $pi, i = 1, \text{or } 2, \cdots$, such as $H1(-\infty, -\infty, p1, p2)$, $H2(p1, p2, p3)$, and $H5(p4, p5, \infty, \infty)$.

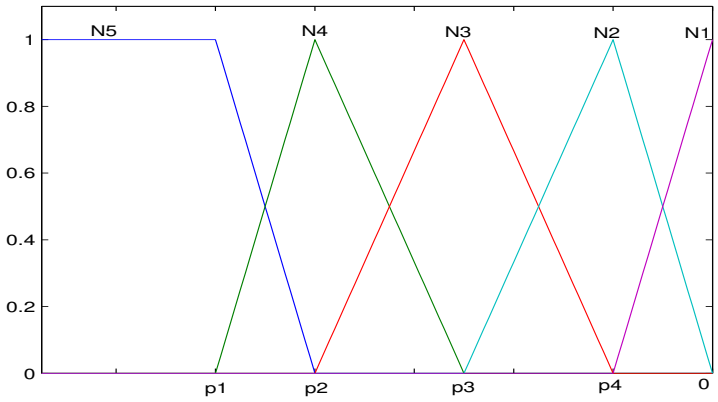
Note that in the MD_FLAQM controller, the effective universe of discourse for the input z is [$p1$, $p5$] and the counterpart for Δz is [$p1$, 0]. Likewise, in the AI_FLAQM controller, a decision has been made for the effective universe of discourse for its inputs. The end points of such an effective universe of discourse specify the "saturation points" at which the outermost membership functions are saturated for input universes of discourse, or beyond which the outputs will not move for the output universe of discourse [79]. The concept of effective universe of discourse makes intuitive sense as at some point the domain expert would just group all extreme values together in the linguistic description so that the membership functions at the outermost edges appropriately characterize "greater than" for the right side or "less than" for the left hand side.

5.3.3 FL Rules in FLAQM(I)

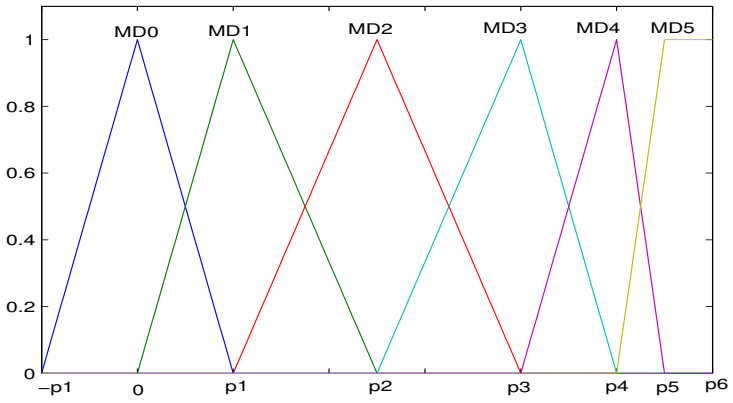
Table 5.1 and Table 5.2 show the fuzzy if-then rules in the MD_FLAQM and AI_FLAQM controllers respectively. FLAQM(I) operates upon a value of z beyond $1 + \delta$, where impending congestion occurs. The principle of selecting fuzzy rules is the larger load factor z is away from the steady-state operating region $[1, 1 + \delta]$ and the more Δz is away from zero, the more strong action is taken, and vice versa. For instance, in the case that z is around $1 + \delta$ and Δz is only slight greater than 0, the slightly increased control action is taken for the dropping probability pr. So, we have *if z is H1 and* Δz *is P0, then AI0.* The other FL rules are derived in a similar way, which are obtained by expertise and the try-and-error method. The rule base of MD_FLAQM consists of 25 fuzzy rules. Although we call the active controller under the situation of $z > 1 + \delta$ and $\Delta z < 0$ a multiplicative decrease (MD) FLC, it does maintain or even increase dropping probability pr sometimes, based on network conditions. MD4 is set as an unchanged control action. The first thing is to judge under which situation of both inputs z and Δz the dropping probability pr would be unchanged. If input z is high and Δz is low, the control value of dropping probability pr has to be increased to cope with sustained high traffic load conditions. Otherwise, if traffic load dramatically reduces to a certain level, the decreased control signal of pr has to been taken. For the AI_FLAQM controller, 30 fuzzy rules are adopted in its rule base. In this FLC, the



(a) Membership functions for z

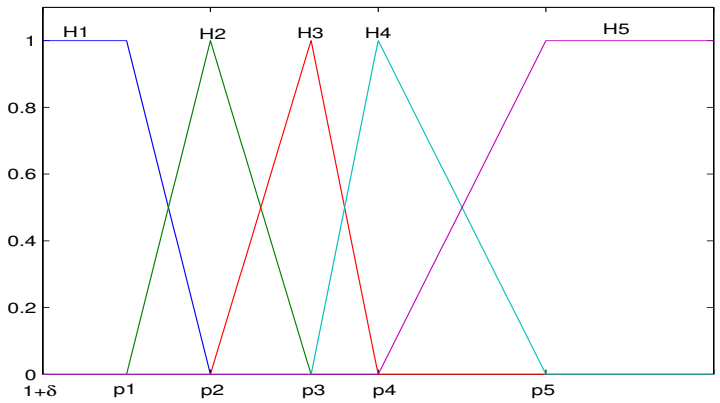


(b) Membership functions for Δz

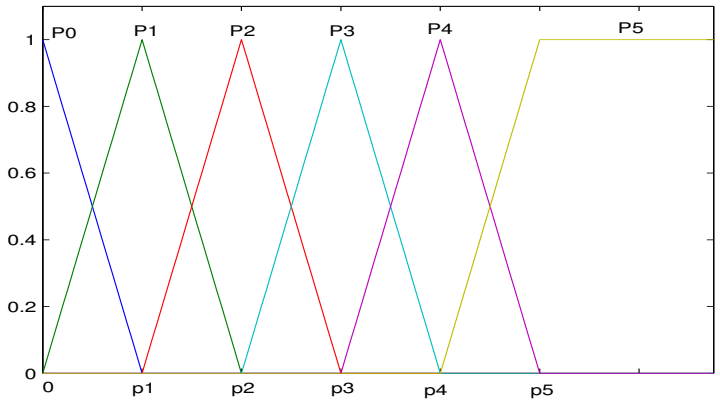


(c) Membership functions for k

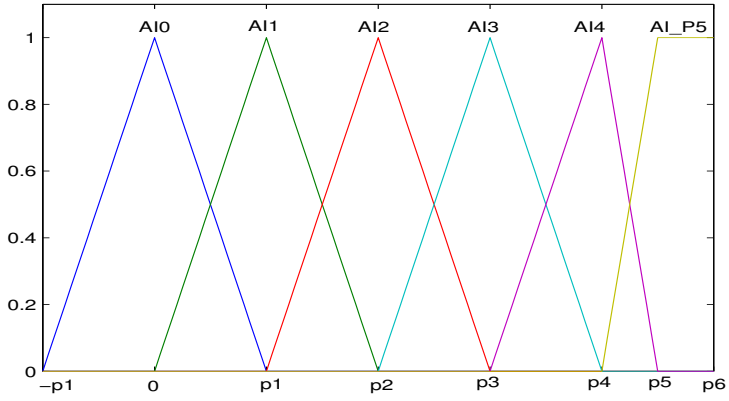
Figure 5.5: MD_FLAQM in FLAQM(I)



(a) Membership functions for z



(b) Membership functions for Δz



(c) Membership functions for Δpr

Figure 5.6: AI_FLAQM in FLAQM(I)

control value of pr is always increased. In the case that z is around $1+\delta$ and Δz is around 0, the slightly increased control action is taken.

The FL rules are obtained by expertise and the try-and-error method. It is worth mentioning that the automation of the control parameters defining the membership functions and even FL rules of the FLC System will be the future study of this project, since the preliminary investigation and application of FL for AQM has been successful.

Table 5.1: FL rules of MD_FLAQM in FLAQM(I)

$\Delta z/z$	H1	H2	H3	H4	H5
N5	MD0	MD1	MD2	MD2	MD3
N4	MD1	MD2	MD3	MD3	MD4
N3	MD2	MD3	MD3	MD4	MD5
N2	MD3	MD3	MD4	MD5	MD5
N1	MD4	MD4	MD5	MD5	MD5

Table 5.2: FL rules of AI_FLAQM in FLAQM(I)

$\Delta z/z$	H1	H2	H3	H4	H5
P0	AI0	AI0	AI1	AI1	AI2
P1	AI0	AI1	AI2	AI2	AI3
P2	AI1	AI2	AI2	AI3	AI4
P3	AI1	AI2	AI3	AI3	AI4
P4	AI2	AI2	AI3	AI4	AI5
P5	AI2	AI3	AI4	AI4	AI5

Remarks on FLAQM(I)

FLAQM(I) has two FLCs with z and Δz as the input variables, and the universe of discourse for the inputs is wide. In precise terms, the range for z in both controller is $[1+\delta, \infty]$, while the range for Δz in the two controller is $[-\infty, 0]$ and $[0, \infty]$, respectively. Also, saturation points are specified for z and Δz respectively to gather the extreme values. The values for the saturation points are heuristically determined with intuition and experience. Be aware that for a wide range of effective universe of discourse, wide membership functions have to be arranged on the right or left to capture the extreme

input values, whereas for a narrow range regularly occurring data will be off the scale and be saturated so that the performance and stability of the controller is affected. One solution to such a problem is to do input normalization using scaling factors so that the effective universe of discourse is [0, 1] or [-1, 1], and tuning the scaling factors with certain criteria. Consequently, input normalization reduces the sensitivity of the controller to the inputs and so reduces the possible oscillation due to any inappropriate selection of saturation points. Alternatively, we propose here to replace z with z' in FLAQM(I), whose effective universe of discourse approximately is $[0, 1]$. This way, input normalization is implicitly realized with the omission of the procedure of tuning the input scaling factor.

5.3.4 Improving FLAQM(I)

An improved version of FLAQM(I) is described in this subsection. FLAQM(II) is applied to the closed-loop feedback queue management system as shown in Figure 5.7. FLAQM(II) uses z' and $\Delta z'$ as process states and the dropping probability pr as control output.

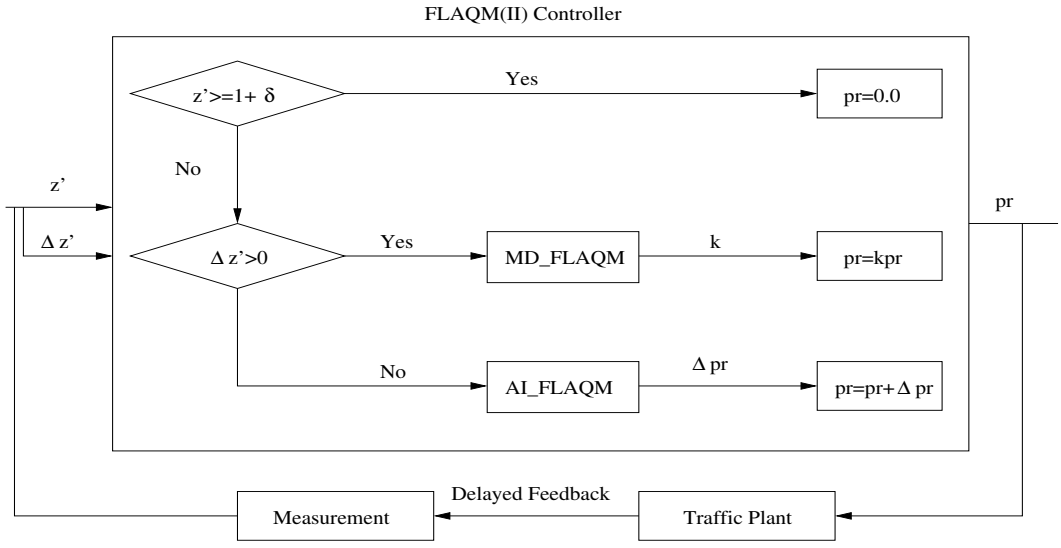


Figure 5.7: A block diagram of the FLAQM(II) control system

The FLAQM(II) controller works also with AIMD policy when input $z' < 1 + \delta$; otherwise, the network traffic load is not too heavy and it is not necessary to early-drop incoming packets. Parameter δ is evaluated with the consideration of control system

stability and link utilization, and we will further discuss the parameter setting issue in section 5.6.

Once a dropping probability is determined by the FLAQM(II) controller, instead of using the RED uniformly drop mechanism, a throttling scheme introduced in the previous chapter is adopted to further decide which incoming packets should be dropped.

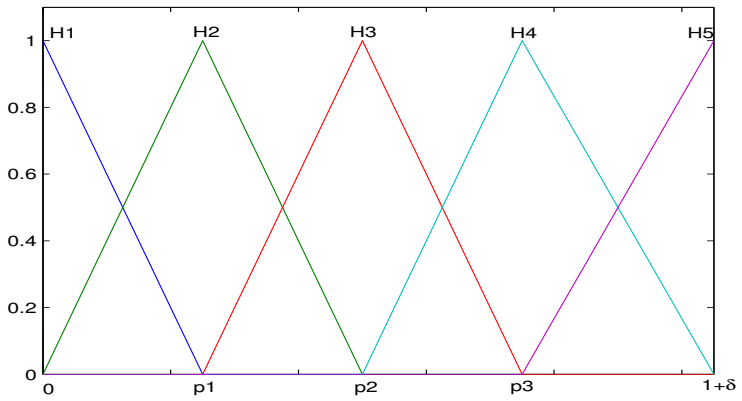
5.3.5 FL Membership Functions of FLAQM(II)

As shown in Figure 5.3, the FLAQM(II) controller also consists of two FLCs, MD_FLAQM and AI_FLAQM. The shape of each membership function is either triangular or trapezoidal. Figure 5.8 shows the membership functions of the MD_FLAQM controller, whereas Figure 5.9 shows the membership functions of the AI_FLAQM controller, where $pi, i = 1, 2, \cdots$, are parameters.

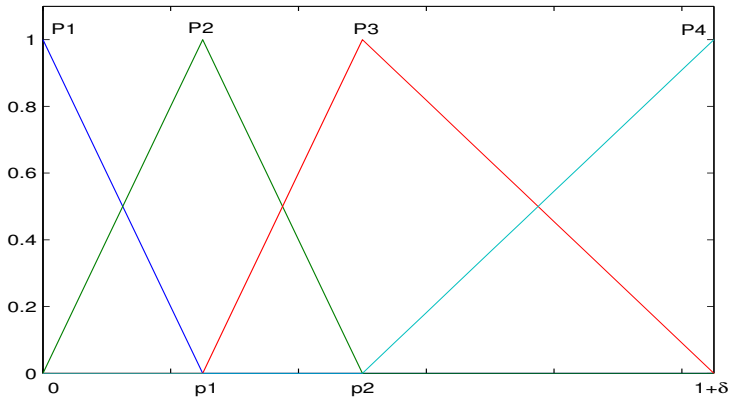
For the MD_FLAQM controller, scaling or normalization of input variables has been implicitly achieved. The inputs z' and $\Delta z'$ are limited in the range of $[0, 1 + \delta]$. For the AI_FLAQM controller, however, the input $\Delta z'$ could vary from negative infinite to zero despite the effective universe of discourse of z' in the range of $[0, 1 + \delta]$. We argue that $\Delta z' < -(1 + \delta)$ is a rather rare event as proved in the simulations. In addition, although it really happens, the previous z' must be larger than $1 + \delta$ and the previous control value of dropping probability pr is zero. In this case, caution is needed to additively increase pr to accommodate bursty traffic and maintain high link utilization. Therefore, in the controller, the same action is taken for any value of $\Delta z'$ less than $-(1 + \delta)$ with $\Delta z' = -(1 + \delta)$.

5.3.6 FL Rules in FLAQM(II)

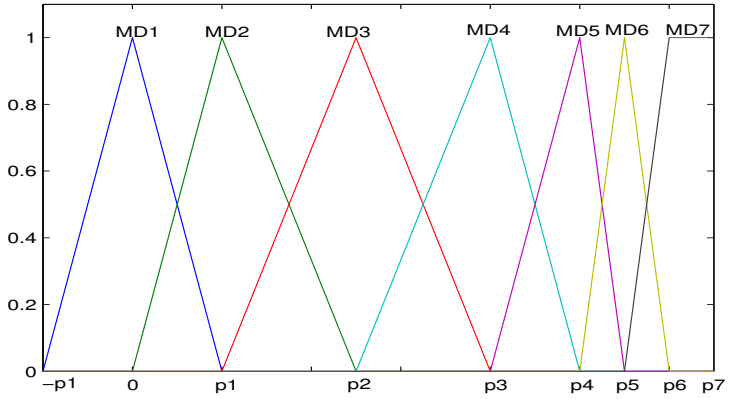
Some fuzzy "if-then" rules are employed to capture the imprecise modes of reasoning that play an essential role in the human ability to make decisions in uncertain and imprecise environments. Table 5.3 and Table 5.4 shows the fuzzy rules in the MD_FLAQM and AI_FLAQM controllers, respectively. The rule base of MD_FLAQM consists of 20 fuzzy rules. Although we call the active controller under the situation of $z' < 1 + \delta$ and $\Delta z' > 0$ a multiplicative decrease FLC, it does maintain or even increase dropping probability pr sometimes, based on network conditions. MD4 is set as an unchanged



(a) Membership functions for z'

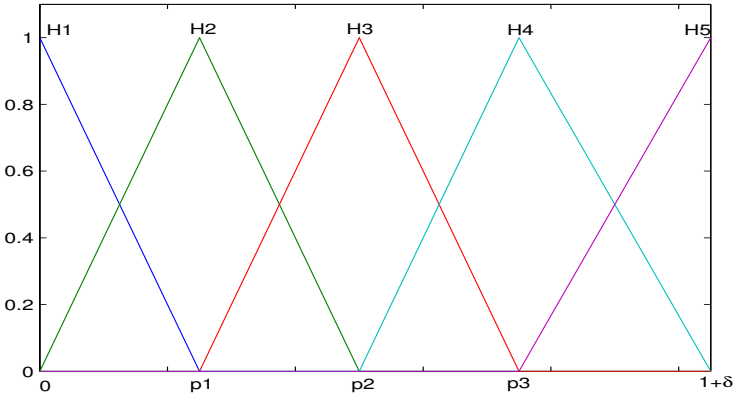


(b) Membership functions for $\Delta z'$

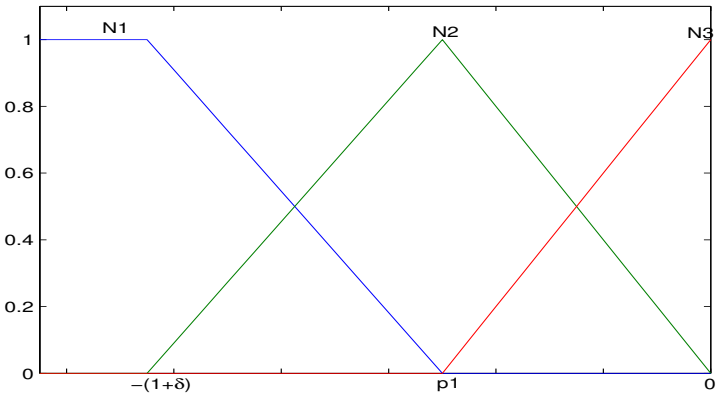


(c) Membership functions for k

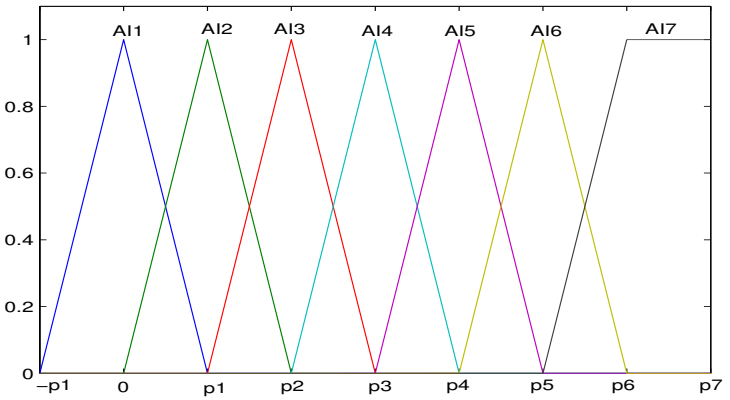
Figure 5.8: MD_FLAQM in FLAQM(II)



(a) Membership functions for z'



(b) Membership functions for $\Delta z'$



(c) Membership functions for Δpr

Figure 5.9: AI_FLAQM in FLAQM(II)

control action. The first thing is to judge under which situation of both inputs z' and $\Delta z'$ the dropping probability pr would be unchanged. Afterwards, if both inputs z' and $\Delta z'$ are low, the control value of dropping probability pr has to be increased to cope with sustained high traffic load conditions. Otherwise, if traffic load dramatically reduces to a certain level, the decreased control signal of pr has to been taken. For the AI_FLAQM controller, 15 fuzzy rules are adopted in its rule base. In this FLC, the control value of pr is almost always increased. The exception is that in the case of $z' = 1 + \delta$ and $\Delta z' = 0$, the unchanged control action is taken with plausible intuition. The principle of selecting these rules in Table 5.4 is that the more load factor z' is away from the steady-state operating region $[1, 1 + \delta]$ and the more $\Delta z'$ is away from zero, the more strong action is taken, and vice versa.

Table 5.3: FL rules of MD_FLAQM in FLAQM(II)

$\Delta z'/z'$	H1	H2	H3	H4	H5
P1	MD7	MD7	MD6	MD5	MD4
P2	MD7	MD6	MD5	MD4	MD3
P3	MD6	MD5	MD4	MD3	MD2
P4	MD5	MD4	MD3	MD2	MD1

Table 5.4: FL rules of AI_FLAQM in FLAQM(II)

$\Delta z'/z'$	H1	H2	H3	H4	H5
N1	AI7	AI6	AI5	AI4	AI3
N2	AI6	AI5	AI4	AI3	AI2
N3	AI5	AI4	AI3	AI2	AI1

5.4 Implementation Complexity of FLAQM

The computational complexity of the proposed FLAQM algorithms is constant with respect to the number of competing flows in a gateway router like most of the existing AQM schemes. AQM is realized by FLAQM in such a way that current traffic load conditions including its level and its change are periodically measured, the FLC system accordingly reasons an appropriate packet marking probability, and arrival packets are

then marked approximately proportional to connection's share of bandwidth. Therefore we believe FLAQM could be implemented in very high speed congestion-prone routers.

The FLAQM algorithms use two-inputs-one-output controllers with the maximum of 30 FL rules. Meanwhile, triangle and trapezoid shapes have been used to construct membership functions to improve on the computational time needed. The simulations have demonstrated that the computation time and memory requirement of FLAQM are comparable to those of other typical AQM schemes. The last point we would like to make here for analyzing the efficiency and feasibility of FLAQM, is that implementation prospects of FLAQM could improve by using hardware such as a better microprocessor or signal processing chip instead of software [79].

5.5 Performance of FLAQM

In this section, we investigate the performance of the proposed FLAQM algorithms compared with that of the traditional Drop Tail (or DT), RED [51], and ARED [50]. The reason to choose RED to do performance comparison here is that RED is not only a benchmark and the IETF default mechanism for buffer management, but also widely studied by the network research community and even Cisco System, a leading networking equipment supplier, has specified its RED implementation. Since the existing AQM schemes do not provide any significant advantage over Drop Tail for realistic traffic load models as discussed in Chapter 3, ARED with the feature of auto-configuration is selected here as another representative AQM. Two simulation experiments have been carried out. In the simulations, the traffic pattern is composed of extremely long FTP connections which last the whole simulations, and Web traffic. By varying the number of active extremely long flows, performance comparison is conducted with different traffic loads in the first experiment, while changed traffic load conditions are simulated in the second experiment. In addition, all the traffic experiences the same propagation delay on the links from the server side to the client side with 40ms from the serves to $R1$, and 1ms from $R3$ to the clients.

The same traffic pattern has been input to the network with different queue management strategies in the bottleneck output buffer including Drop Tail, RED, ARED, and our proposed FLAQM algorithms. The parameter settings in the FLAQM(I) and

FLAQM(II) controllers used in the simulations are shown in Table 5.5 and Table 5.6, respectively. Additionally, Figure 5.10 and 5.11, and Figure 5.12 and 5.13 illustrate the decision surfaces for AI_FLAQM and MD_FLAQM with these parameter choices in FLAQM(I) and FLAQM(II), respectively.

Table 5.5: Parameter settings of the FLAQM(I) controller

target queue length	60 packets
update interval	0.5s
δ	0.05
Parameters of z	$p1 = 1.1, p2 = 1.5, p3 = 2.0, p4 = 2.5, p5 = 3.0$
Parameters of Δz in MD_FLAQM	$p1 = -2.0, p2 = -1.0, p3 = -0.5, p4 = -0.2$
Parameters of Δz in AI_FLAQM	$p1 = 0.2, p2 = 0.5, p3 = 1.0, p4 = 1.5, p5 = 2.0$
Parameters of k in MD_FLAQM	$p1 = 0.8, p2 = 0.85, p3 = 0.9, p4 = 1.0, p5 = 1.1, p6 = 1.15$
Parameters of Δpr in AI_FLAQM	$p1 = 0.01, p2 = 0.02, p3 = 0.03, p4 = 0.04, p5 = 0.05, p6 = 0.06$

Table 5.6: Parameter settings of the FLAQM(II) controller

target queue length	60 packets
update interval	0.5s
δ	0.05
Parameters of z'	$p1 = 0.25, p2 = 0.5, p3 = 0.75, p4 = 1.05$
Parameters of $\Delta z'$ in MD_FLAQM	$p1 = 0.25, p2 = 0.5$
Parameters of Δz in AI_FLAQM	$p1 = -0.5$
Parameters of k in MD_FLAQM	$p1 = 0.5, p2 = 0.95, p3 = 1.0, p4 = 1.05, p5 = 1.1, p6 = 1.15, p7 = 1.2$
Parameters of Δpr in AI_FLAQM	$p1 = 0.01, p2 = 0.02, p3 = 0.03, p4 = 0.04, p5 = 0.05, p6 = 0.06, p7 = 0.07$

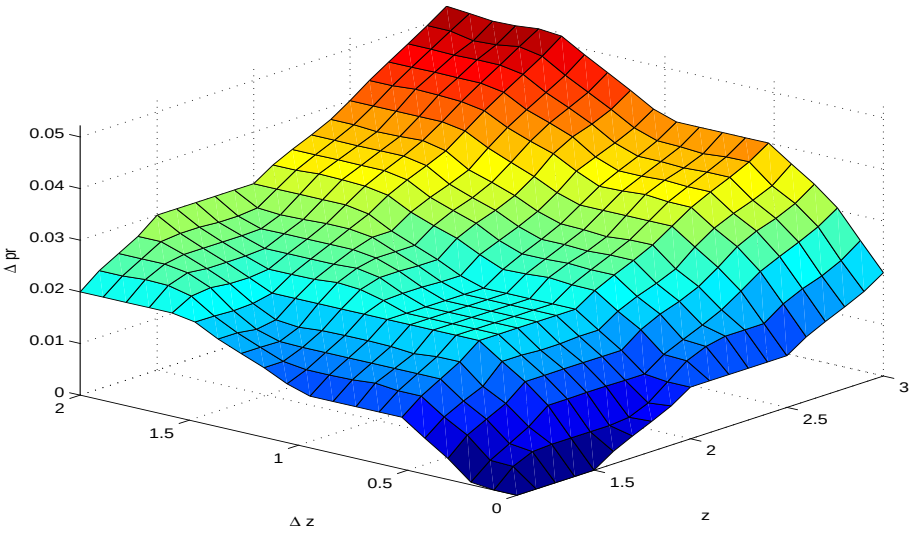


Figure 5.10: Decision surface of AI_FLAQM in FLAQM(I)

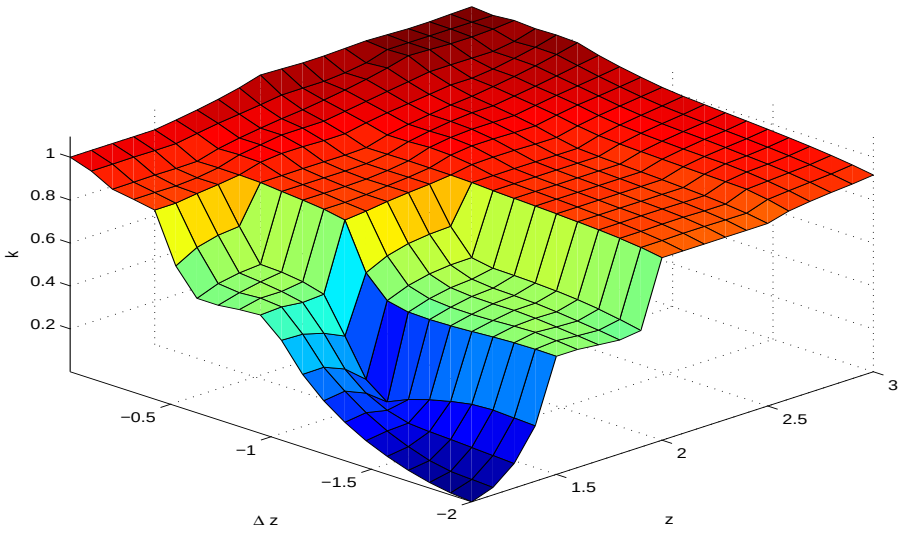


Figure 5.11: Decision surface of MD_FLAQM in FLAQM(I)

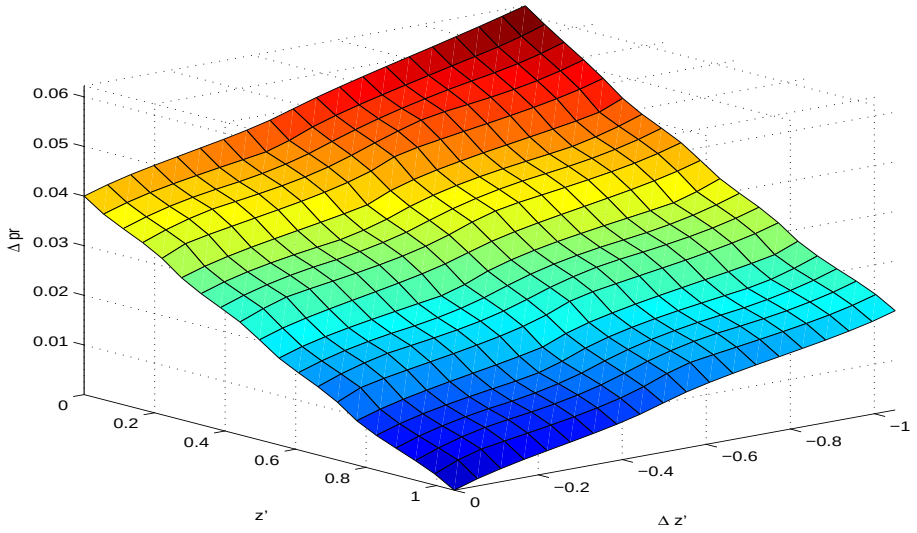


Figure 5.12: Decision surface of AI_FLAQM in FLAQM(II)

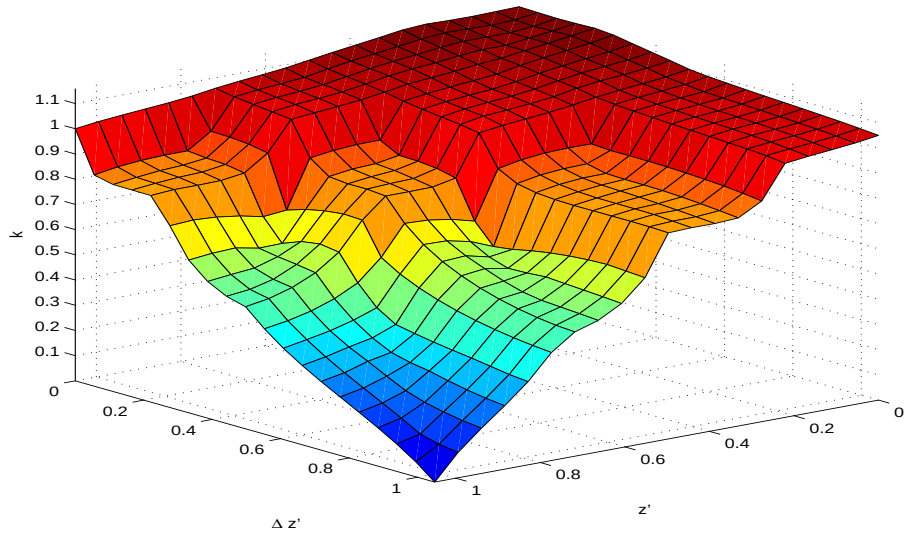


Figure 5.13: Decision surface of MD_FLAQM in FLAQM(II)

5.5.1 Simulations of a Congested Network

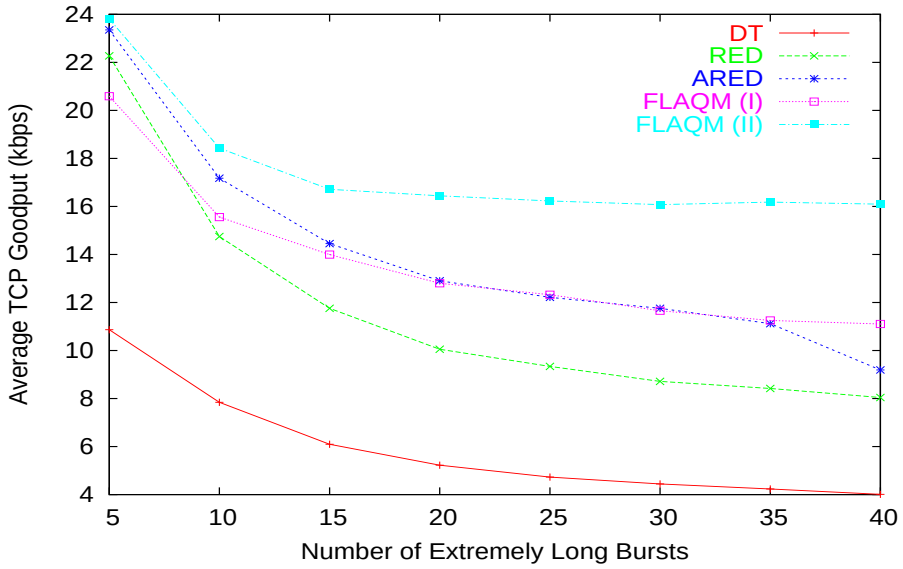
In the simulations, traffic is generated to cause congestion on the bottleneck in the selected network topology described in Chapter 4. There are two kinds of traffic. Traffic from server s1 to client c1 models Web service including long bursts and short bursts. The traffic is generated by the Poisson-Pareto traffic model with the truncated threshold of Pareto distribution of 1000 packets. On the other hand, the traffic from the other servers simulates extremely long bursts which last the whole simulation period, and the number of such servers is varied ranging from 5 to 40 to create different traffic conditions. Note that the simulation duration is 1000s and the first half is set as a warmup period (the simulation code written in Tcl is given in Appendix A.2).

5.5.1.1 Performance Results

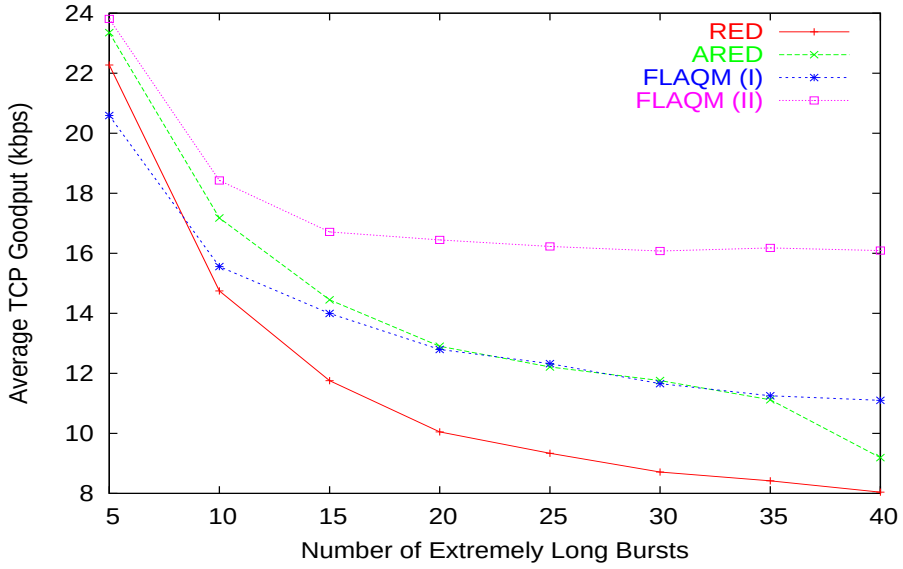
Compared with Drop Tail, RED, and ARED, our proposed FLAQM(I) and FLAQM(II) demonstrate their robustness in a various traffic load conditions as follows.

- The weighted average user TCP goodput comparison plotted in Figure 5.14 shows that FLAQM(II) outperforms the others, while FLAQM(I) achieves goodput performance comparable to ARED, and the goodput performance of Drop Tail is far more behind the others.

- Drop Tail obtains the highest weighted average flow latency in all the investigated traffic conditions and also flow latency increases dramatically with the number of extremely long bursts compared to those of the AQM schemes as depicted in Figure 5.15 (a). Among the AQM schemes, FLAQM(II) still performs the best in terms of flow latency, while FLAQM(I) and ARED are comparable as shown in Figure 5.15 (b).

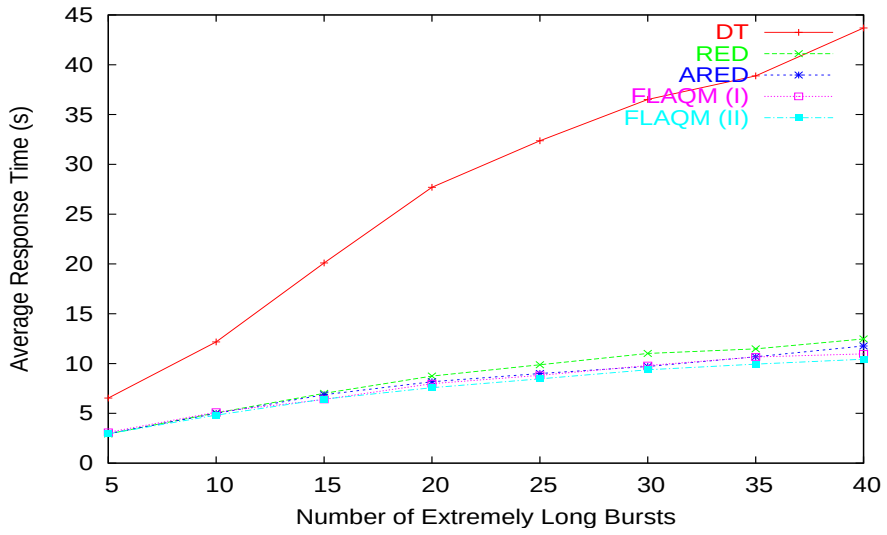


(a) User TCP goodput

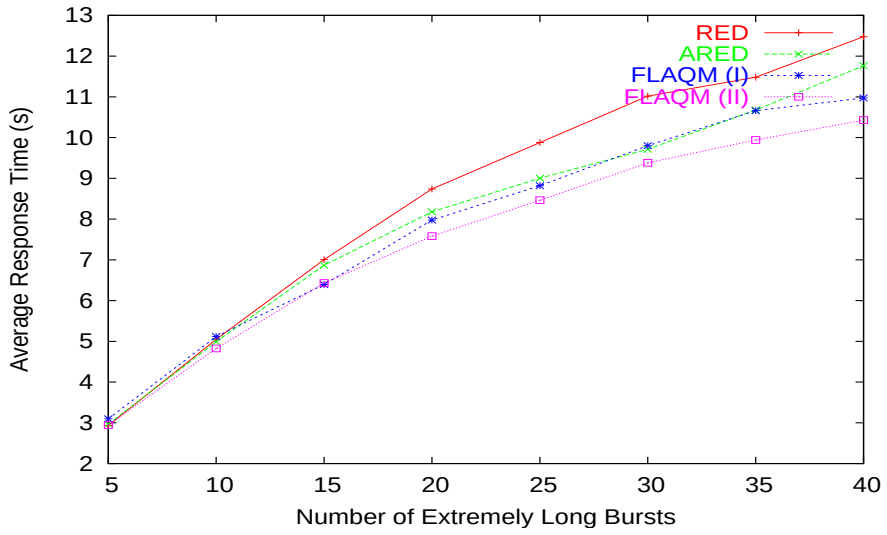


(b) User TCP goodput without Drop Tail

Figure 5.14: Weighted average user TCP goodput comparison



(a) User Response Time



(b) User Response Time without Drop Tail

Figure 5.15: Weighted average user response time comparison

- The network throughputs of FLAQM(I) and FLAQM(II) are the best as plotted in Figure 5.16, with almost 100% link utilization for all the schemes including Drop Tail.

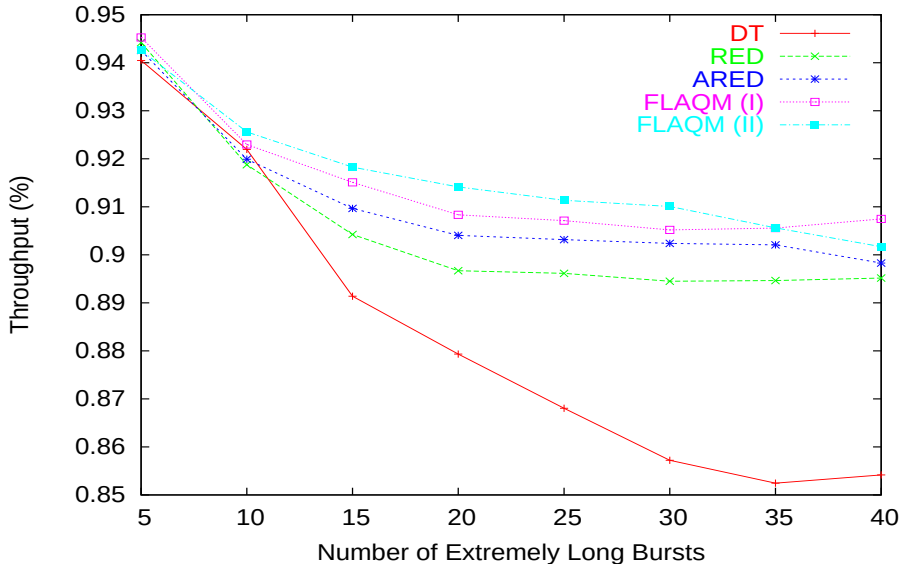
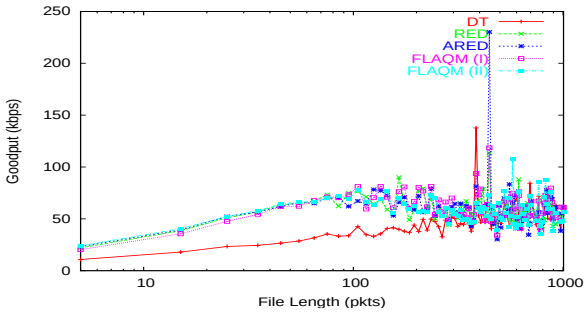
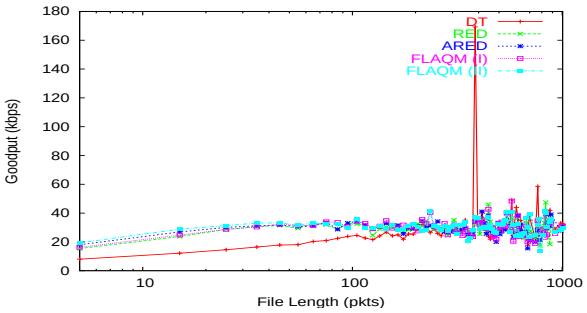


Figure 5.16: Network throughput

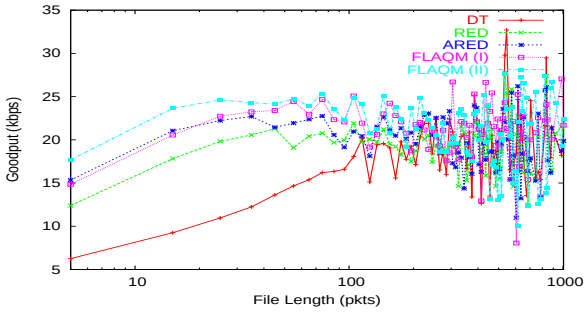
- The AQM schemes largely improve the goodput of mice flows compared with Drop Tail as shown by TCP goodput performance in Figure 5.17. However, mice flows with each AQM scheme under certain circumstances (especially with a small number of background extremely long bursts) still suffer lower TCP goodput than the elephants flows.



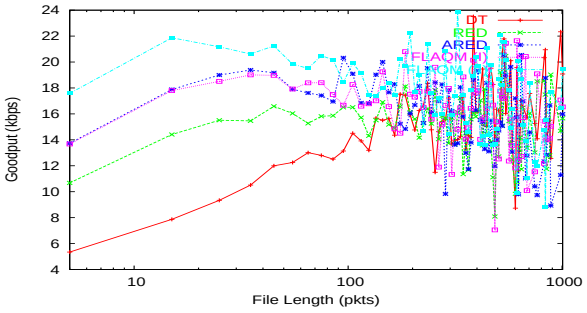
(a) n=5



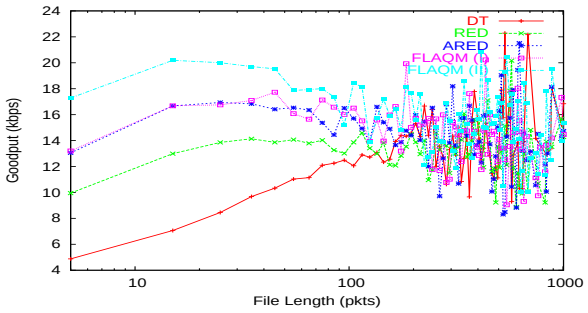
(b) n=10



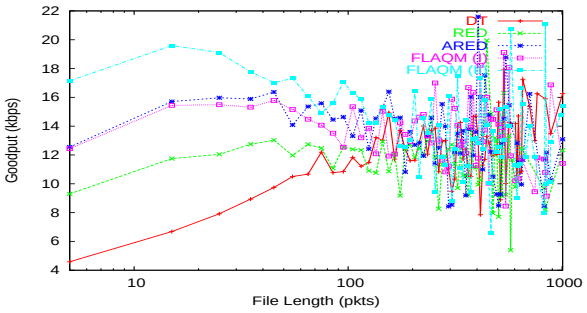
(c) n=15



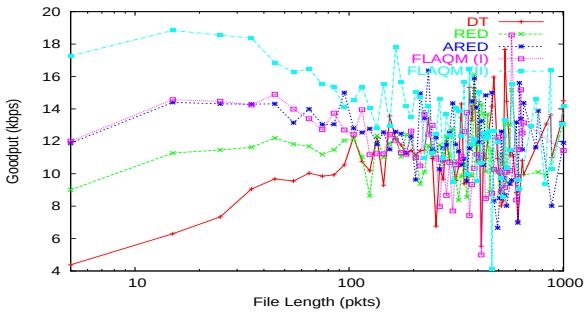
(d) n=20



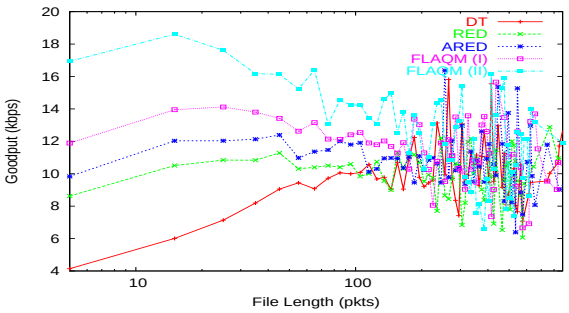
(e) n=25



(f) n=30



(g) n=35



(h) n=40

Figure 5.17: TCP goodput versus file lengths with different number of extremely long bursts

5.5.1.2 Stability Comparison

We evaluate the stability of FLAQM in two aspects including comparing FLAQM to other AQM schemes and the two algorithms within FLAQM.

Firstly, in Chapter 3 we have revealed the sensitivity of some existing AQM methods to traffic load fluctuation in the presence of mice flows. Large oscillation in queue dynamics was observed. To alleviate the problem, we deploy not only the traffic load factor to reflect congestion levels, but also the change of traffic load factor to predict the trend of traffic load. Moreover, at least one RTT time duration is used as the dropping probability adjustment period.

Figures 5.20 and 5.21 illustrate relative steady queue dynamics for the two FLAQM algorithms compared to RED and ARED in Figures 5.18 and 5.19, using 20 of FTP extremely long bursts as example. RED also exhibits a longer queuing delay, whereas FLAQM(II) obtains a lower queueing delay.

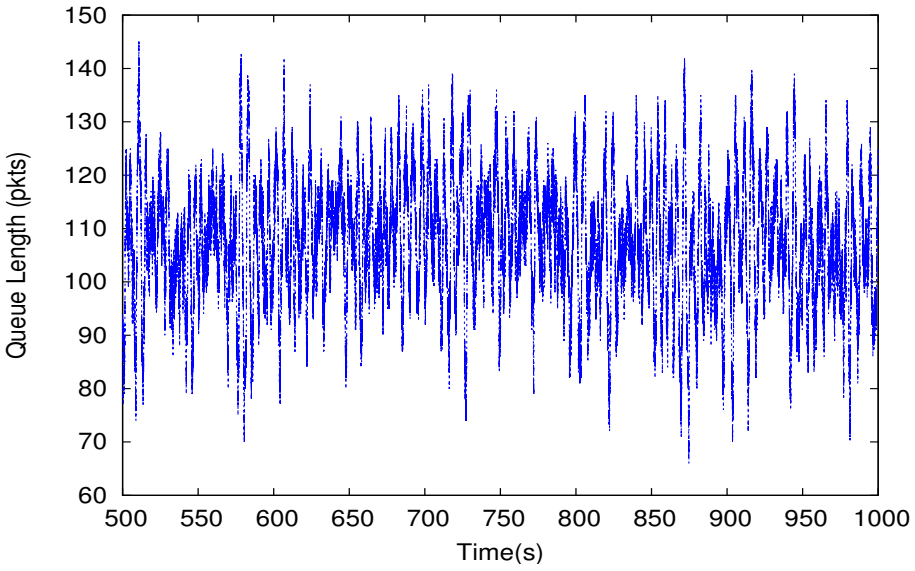


Figure 5.18: Queue dynamics of RED using 20 of FTP extremely long bursts

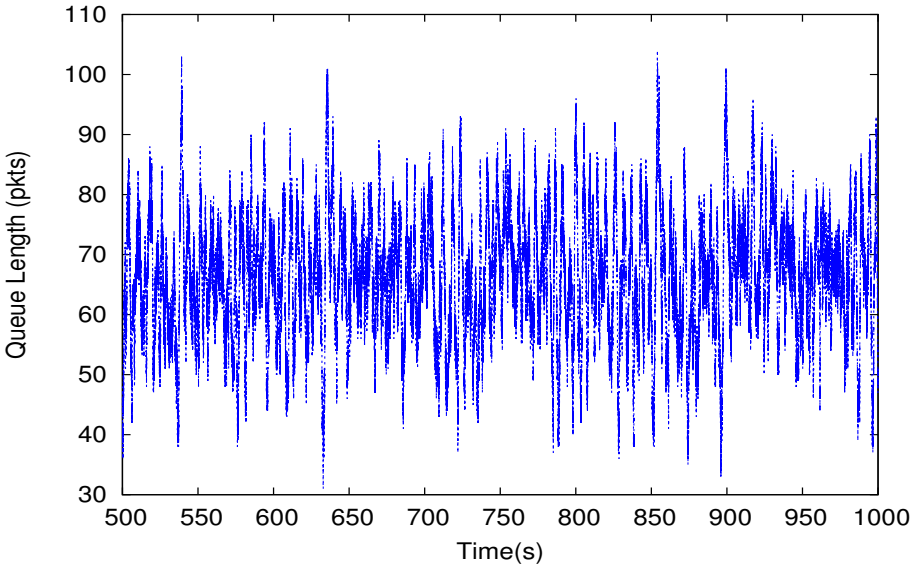


Figure 5.19: Queue dynamics of ARED using 20 of FTP extremely long bursts

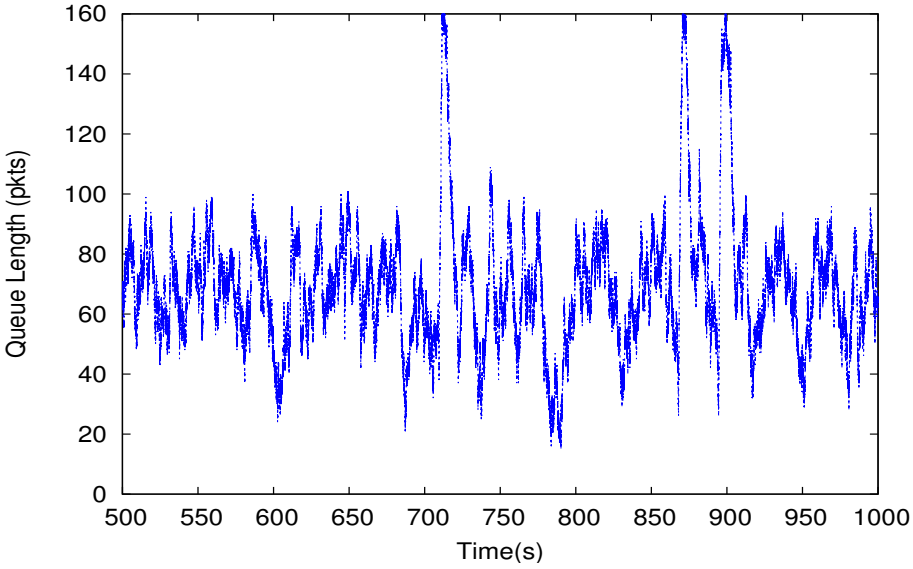


Figure 5.20: Queue dynamics of FLAQM(I) using 20 of FTP extremely long bursts

On the other hand, one major advantage of FLAQM(II) is that it implicitly realizes normalization for input variables z' and $\Delta z'$. Therefore, system stability is expected to be improved over FLAQM(I). Figures 5.22, 5.23, 5.24, and 5.25 show stability comparison between the two proposed schemes by tracing load factor z and observing its dynamics under different traffic load scenarios. We conclude that under overloaded network situations, compared with FLAQM(I), FLAQM(II) is capable of quickly driving the system towards the set point of traffic load factor z without much oscillation.

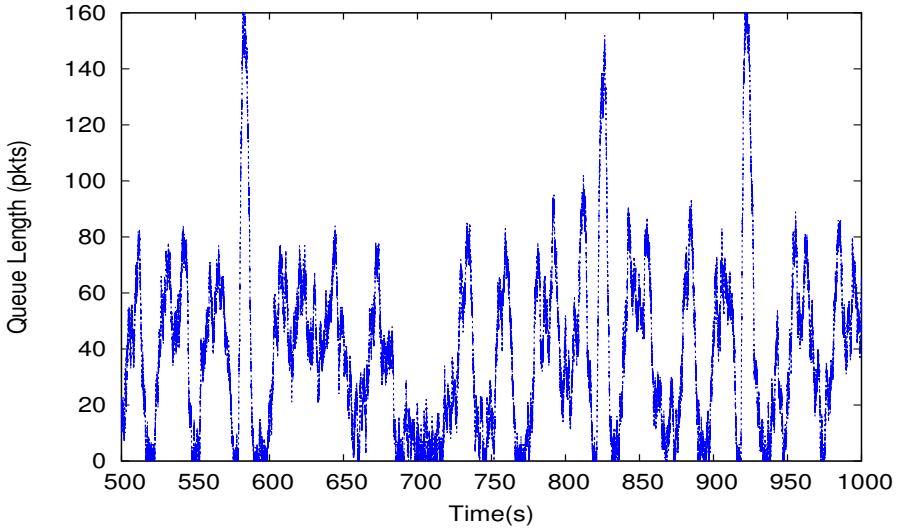


Figure 5.21: Queue dynamics of FLAQM(II) using 20 of FTP extremely long bursts

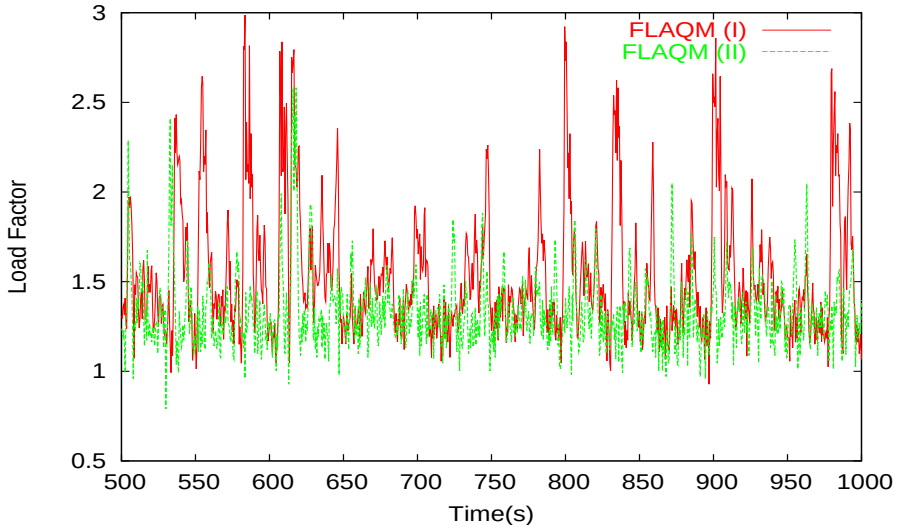


Figure 5.22: Load factor comparison between FLAQM(I) and FLAQM(II) with n=10

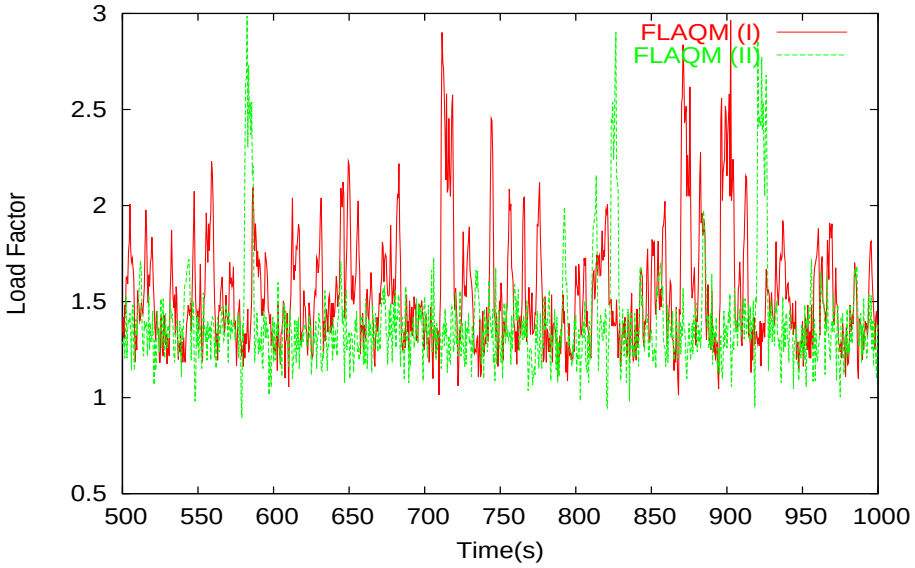


Figure 5.23: Load factor comparison between FLAQM(I) and FLAQM(II) with n=20

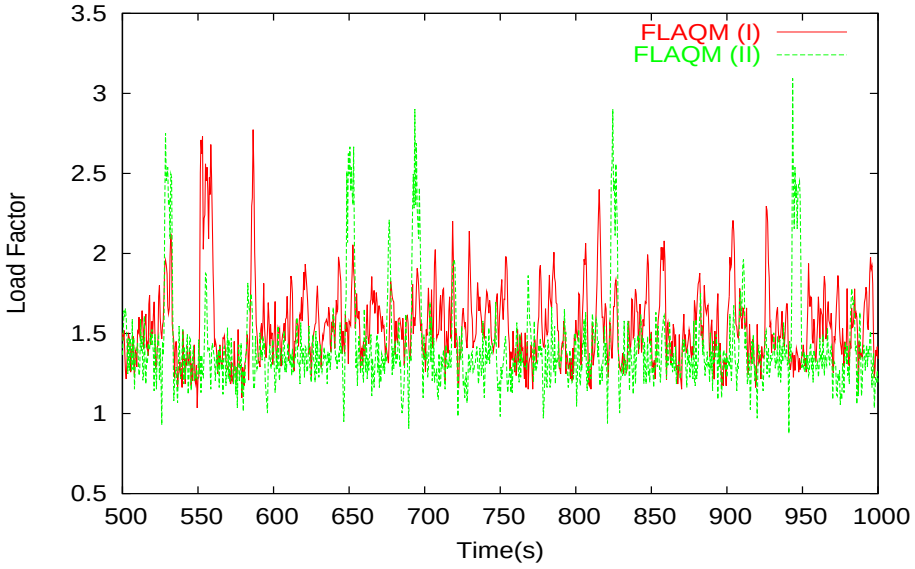


Figure 5.24: Load factor comparison between FLAQM(I) and FLAQM(II) with n=30

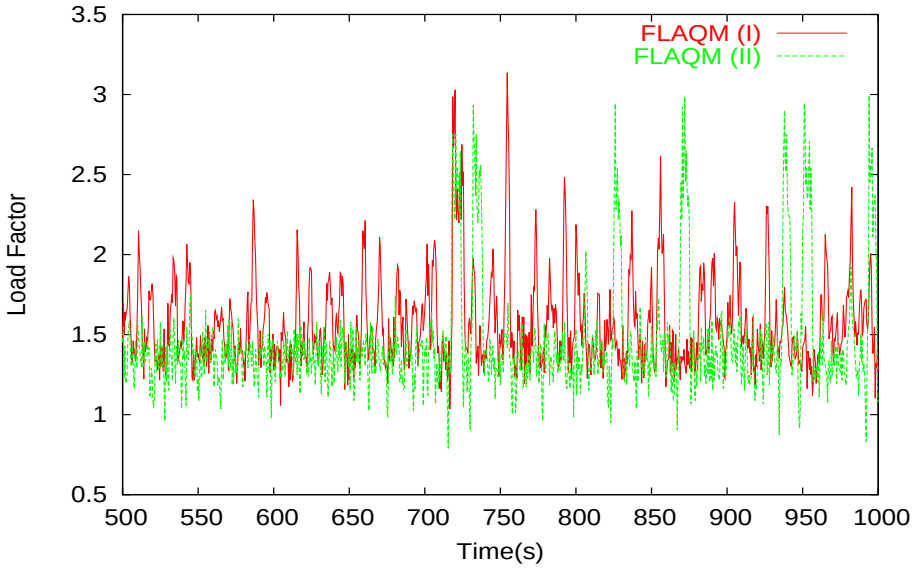
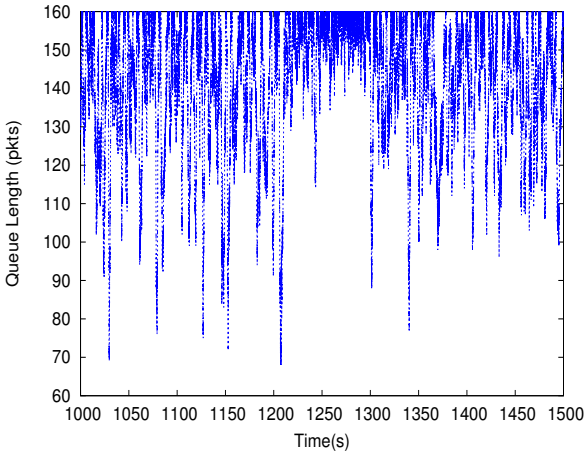


Figure 5.25: Load factor comparison between FLAQM(I) and FLAQM(II) with n=40

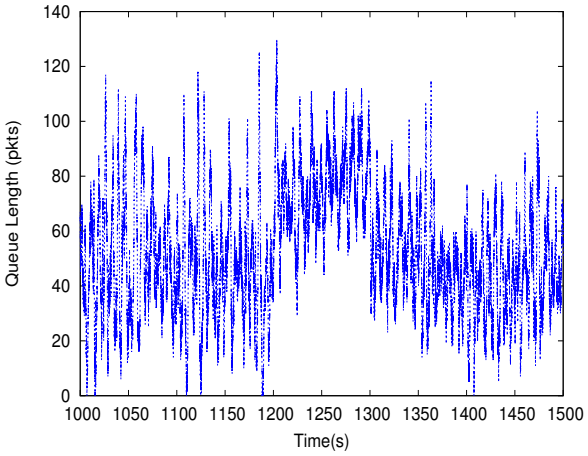
5.5.2 Simulations with Changed Traffic Loads

In this experiment, we evaluate the performance of FLAQM under changed traffic load situations. The truncated Pareto Web traffic is reused but with traffic load chosen as 60% instead of 100%. Also five extremly long FTP flows are imitated as part of the backgroud traffic, besides the Web flows. To obtain changed traffic loads, different numbers of connections are added ranging from 5 to 40, starting from simulation time 1200s with 200ms inter-arrival time between these flows and disappearing at simulation time 1300s. Note that the simulations have been run for 2000s and the first 1000s is taken as warmup duration.

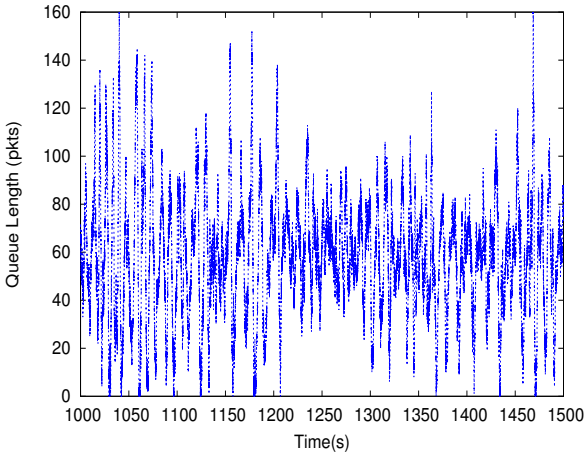
To exhibit the response of each scheme investigated in this experiment to changed traffic load conditions, scenarios of adding 10, 20, 30, and 40 more flows are taken as examples, and Figures 5.26, 5.27, 5.28, and 5.29 illustrate the queue length dynamics of these schemes with adding different numbers of flows. In these figures, the queue dynamics of Drop Tail is typical under overload network conditions, whereas RED is not sensible to the change of traffic loads. ARED, FLAQM(I), and FLAQM(II) have quick response to the traffic load change. ARED and FLAQM(I), however, achieve it with the cost of too much oscillation. Note that with addition of 40 flows, ARED fails to bring queue length down to its predefined value, while FLAQM(II) still performs consistently.



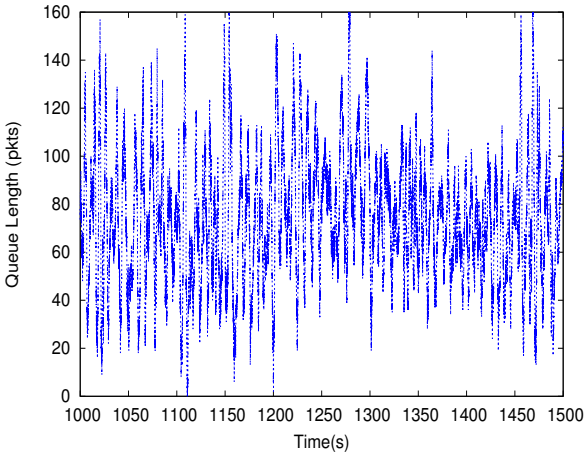
(a) Drop Tail



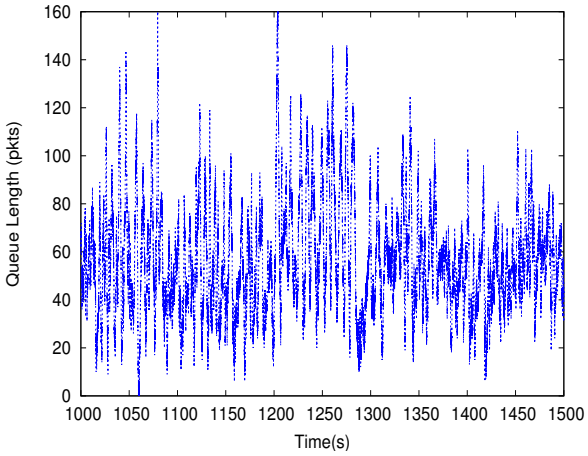
(b) RED



(c) ARED

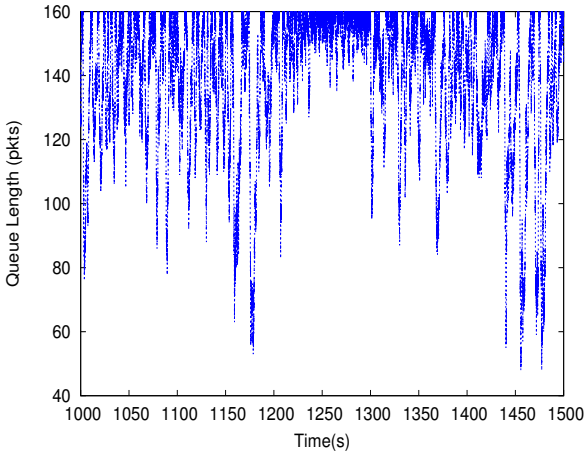


(d) FLAQM(I)

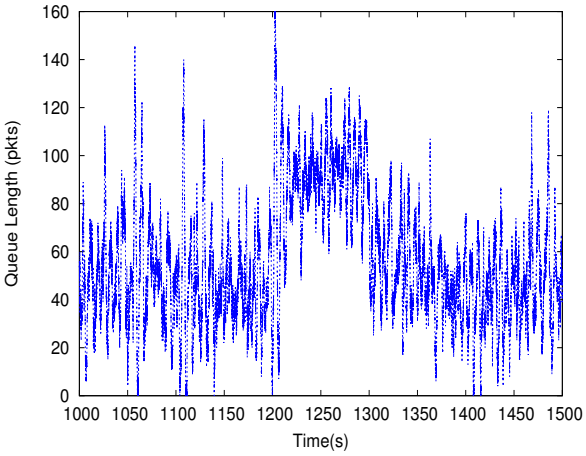


(e) FLAQM(II)

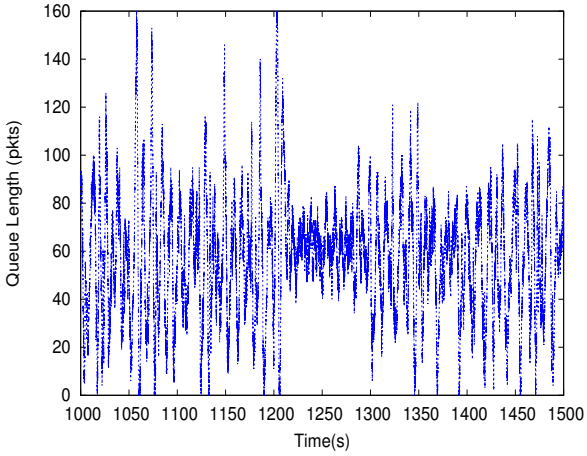
Figure 5.26: Queue length dynamics of different schemes with n=10



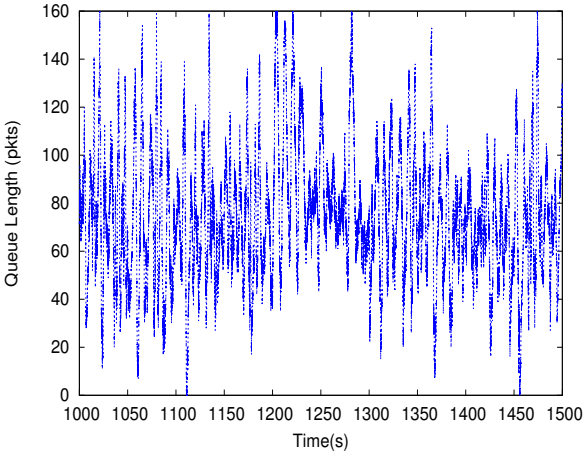
(a) Drop Tail



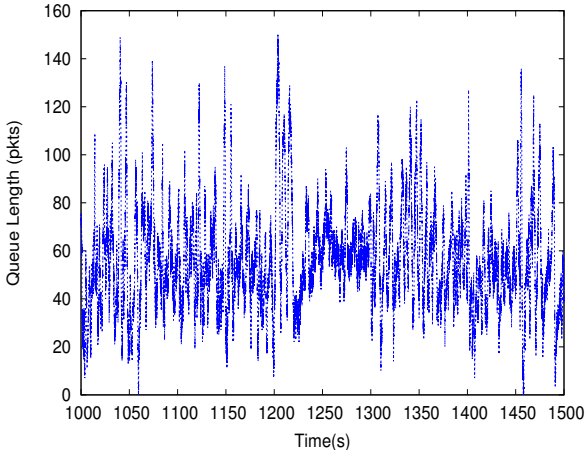
(b) RED



(c) ARED

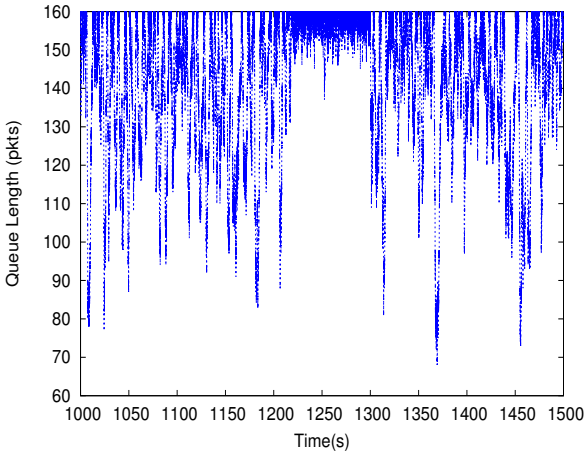


(d) FLAQM(I)

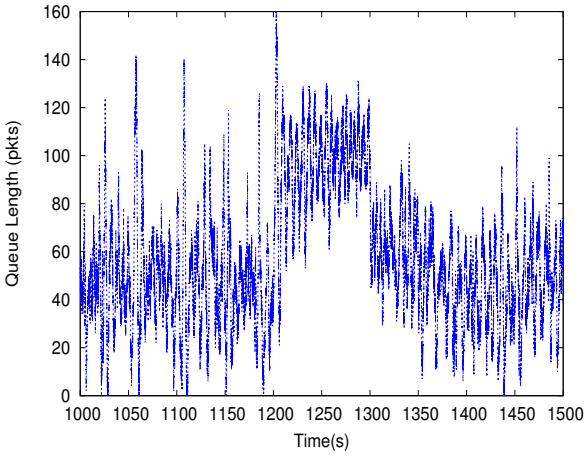


(e) FLAQM(II)

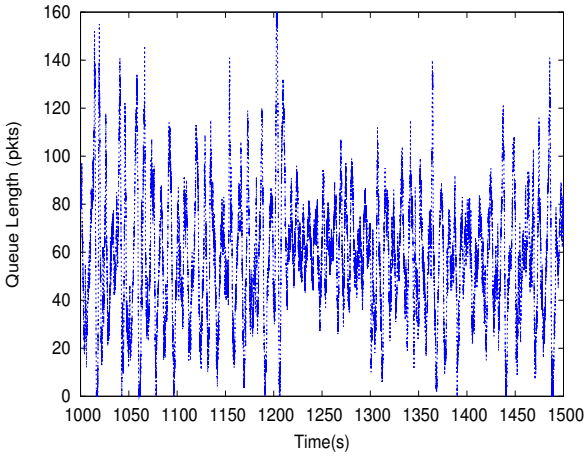
Figure 5.27: Queue length dynamics of different schemes with n=20



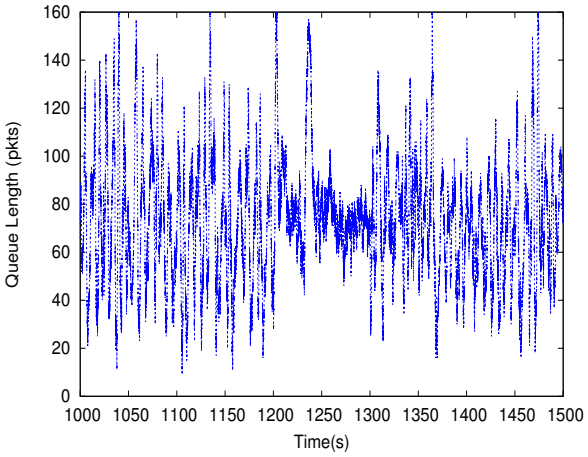
(a) Drop Tail



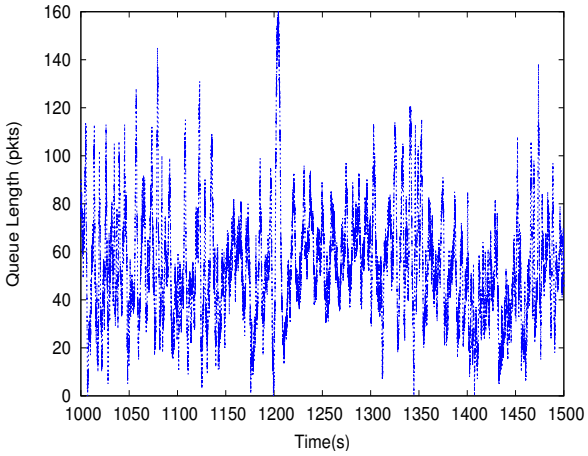
(b) RED



(c) ARED

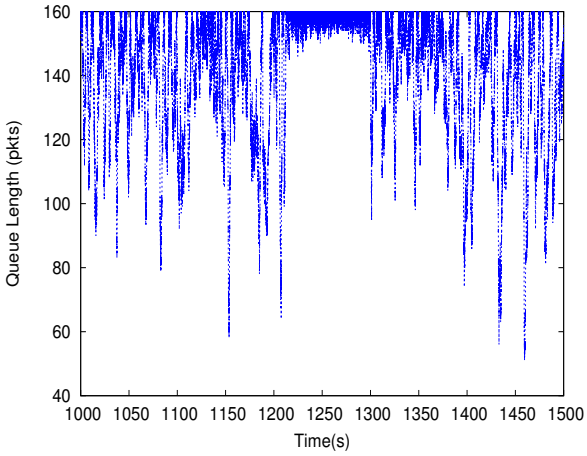


(d) FLAQM(I)

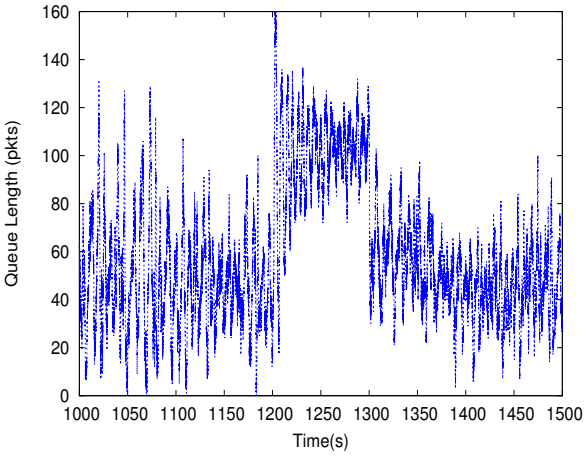


(e) FLAQM(II)

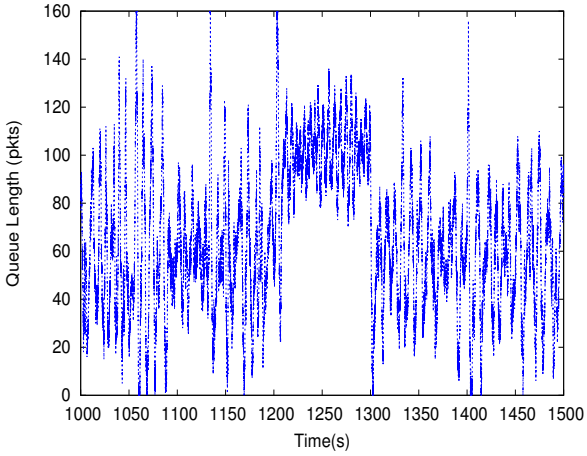
Figure 5.28: Queue length dynamics of different schemes with n=30



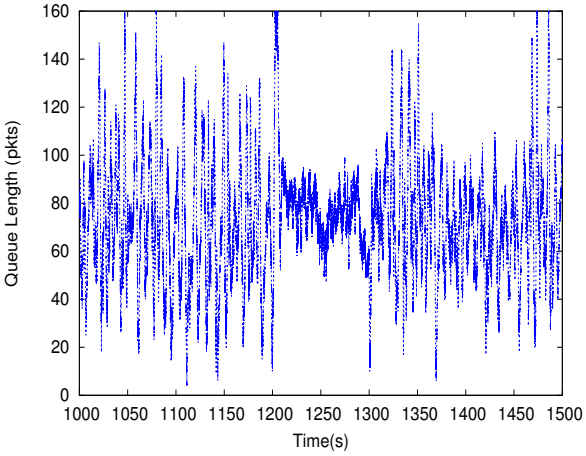
(a) Drop Tail



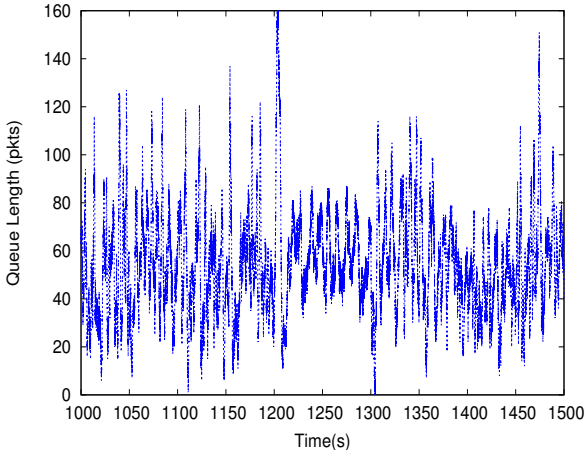
(b) RED



(c) ARED



(d) FLAQM(I)



(e) FLAQM(II)

Figure 5.29: Queue length dynamics of different schemes with n=40

5.6 Configuring the FLAQM Controller

In this section, we consider how to effectively configure the parameters of the FLAQM controller. Due to its achievement of better performance, FLAQM(II) is set as a default option for the FLAQM controller. FLAQM parameters may be classified into (1) fuzzy parameters which include $pi, i = 1, 2, \cdots$ in each membership function of the two FLCs, MD_FLAQM and AI_FLAQM, and (2) system parameters which consist of dropping probability pr updating interval, desired queue length Q_0, and system stability parameter δ. All these parameters have a significant impact on system stability and performance. However, different ways of setting parameters might be necessary in each category. The fuzzy parameters are decided by human experience combined with the trial-and-error method. A good FLC can been designed at the first stage, since it is inherently robust in terms of conducting control in environments with a high level of uncertainty, complexity, or nonlinearity such as exist in the Internet. Nevertheless, these fuzzy parameters might be tuned on-line after the original design to enable a wider class of problems to be solved by reducing the prior uncertainty to the point where satisfactory solutions can be obtained. The on-line adaptive tuning might be achieved by using other artificial intelligent (AI) technologies such as neural networks (NNs) and genetic algorithms (GAs), at a price of high complexity.

In this section, we focus on how to choose those system parameters: dropping probability pr updating interval, desired queue length, and system stability parameter δ, most of which are also the concerns of other AQM schemes. For instance, PI [19] and REM [27, 73, 4] also periodically adjust dropping probability pr, while ARED [50] updates the maximum dropping probability max_p in a certain time interval with a default value of 0.5s. All these three schemes predefine a value for desired queue length. The system stability parameter δ, as one can find out later, is also related to desired link utilization as well as system stability. All AQM schemes aim to achieve high link utilization. Especially, AVQ [67] and GREEN1 [94] explicitly set a desired link utilization value in their algorithms.

5.6.1 Dropping Probability *pr* Updating Interval

The choice of the dropping probability *pr* updating interval is critical to the performance of FLAQM. A shorter updating interval would make the FLAQM take fraud message contributed by transit situations and thus cause system oscillation, whereas with a longer updating interval, it takes time for the controller to realize changes have occurred in network conditions, which might have deteriorated by becoming either heavily congested or idle.

To determine a reliable updating interval, observe that the feedback information of current dropping probability *pr*, which a FLAQM enabled gateway router expects to obtain, is delayed for at least the maximum of one RTT ($maxRTT$) of all the active connections. In addition, in order to accommodate transition of traffic load between updating intervals, a reliable value for the *pr* updating interval is set between $maxRTT$ and $2 \times maxRTT$ in our algorithm.

5.6.2 Desired Queue Length

For a given bottleneck link, a desired queue length Q_0 and the corresponding target queuing delay T_0 can be converted into each other by using formula $T_0 = \frac{Q_0}{link\_capacity}$. The parameter T_0 is affected by several parameters such as router buffer size, bottleneck link capacity, and *pr* updating interval. The selection of the parameter T_0 is a tradeoff between throughput and delay, and is ISP policy based. In FLAQM, in the case where bottleneck link capacity is $1.5Mbps$ and buffer size is 160 packets (1000 bytes for each packet), for instance, 60 packets or 320ms is heuristically chosen.

5.6.3 System Stability Parameter δ

In FLAQM, whenever a packet arrives, the controller in a gateway drops it with a certain probability. The dropping probability *pr* is adjusted periodically based on the measurements of network conditions in the last period. Network conditions are measured by z' and $\Delta z'$. Variable z' reflects congestion occurrence and the level of traffic load, while variable $\Delta z'$ indicates the degree of the change in traffic load. If $z' < 1 + \delta$, the controller regards this signal as the occurrence of congestion and earlier dropping action is taken with the current dropping probability *pr*; otherwise, it drops

the incoming packet with zero probability.

It is no doubt that the desired value of the measured parameter z' in the control system is 1. In order to do control smoothly and respond to increased traffic load quickly, however, the set point of z' chosen in FLAQM is $1 + \delta$, where δ is positive. Note that a negative δ is theoretically acceptable. However, at this point z' is less than 1 and traffic load is pretty heavy so that it might be too late for the FLAQM controller to take any control action. This is also proven by simulation experiments.

On the other hand, parameter δ is also related to link utilization. In a measurement duration dur, link utilization (uti) is measured by the division of the number of sent packets and link capacity during dur, with the formula $uti = \frac{sent\_pkts}{link\_capacity}$. The source of the sent packets is those leftover packets at the end of the last measurement period and incoming packets during the current measurement interval. Be aware that some of the incoming packets might be dropped because of a full buffer or early dropping by FLAQM, and left in the buffer at the end of the current measurement duration. We consider the link utilization in the condition of $Q < Q_0$ for simplicity. Note that this condition is desired for data transmission, and also is dominant as exhibited in the simulations. In this case, $fraction = 1$ in the definition of the traffic load factor z and thus load factor $z = \frac{input\_pkts}{link\_capacity}$. With a bigger δ, the set point of z' is smaller, and also the number of the desired incoming packets is smaller. Consequently, the link utilization might be lower. Therefore, a smaller δ is preferred, and the default value in FLAQM is chosen as 0.05.

5.7 Summary

In this paper, we have proposed two novel AQM schemes: FLAQM(I) and FLAQM(II), by using one artificial intelligent (AI) method, fuzzy logic (FL). Via extensive simulations, both FLAQM schemes outperform the other well-known queue management strategies such as Drop Tail, RED, and ARED. Also, due to the achievement of scaling or normalization of the traffic load factor z' and its change $\Delta z'$, FLAQM(II) has improved the performance of FLAQM(I), in which normalization of its inputs is omitted since it is hard to know the maximum value of the inputs with different traffic load conditions. Although with an estimated approximate value for the maximum the per-

formance of FLAQM(I) is reasonable, FLAQM(II) theoretically is expected to improve stability and performance. Thus, without specification, the term 'FLAQM' is used to stand for FLAQM(II).

One more point we would like to discuss here is the calculation of $fraction$ in target capacity (see Section 5.3 for detail). At the end of each measurement period, when the instantaneous queue length Q is less than or equal to a predefined value Q_0, $fraction$ is set as 1 in the next period. Alternatively, the situation of $Q \leq Q_0$ can be understood as more capacity left for accommodating incoming packets. However, this causes system oscillation in simulations. Thus, $fraction = 1$ is a better choice in the case of $Q \leq Q_0$.

More work still needs to be done to get an even better performance of FLAQM to realize the full potential advantages of FLC in complex nonlinear problem solving. Future work will consider the integration of other AI techniques such as neural networks (NNs) and Genetic Algorithms (GAs) to achieve the self-tuning of the parameters in the two FLCs: MD_FLAQM and AI_FLAQM.

Chapter 6

Coupling FLAQM with Mice and Elephants Strategy

In Chapter 4, we noticed the vulnerability of mice flows in bandwidth share in competition with elephants. In this chapter, we consider an access network by which users are linked to backbone routers. We propose a mice and elephants (ME) strategy and use it at the access network. The objective of this chapter is to look at how the FLAQM designed in Chapter 5 is coupled with the ME strategy to conduct queue management for elephants by making use of the ability of FLAQM to deal with the coexistence of different traffic classes. More specifically, the use of the ME strategy is justified and its network context and operation are depicted in Section 6.1. The router functionalities of the ME strategy are described in Section 6.2. Experiments are conducted to investigate the proposed strategy at gateway routers and to make comparisons with other strategies implemented either at the gateways or at the edge routers. The experiment results are given in Section 6.3, while result analysis is depicted in both quantitative and qualitative aspects in Section 6.4.

6.1 Mice and Elephants Strategy

In this section, we look into the characteristic of today's Internet traffic and the possibility of improving performance in gateway routers in premise networks.

6.1.1 A Phenomenon on Today's Internet Traffic

In today's Internet traffic, there is an obvious phenomenon in which most flows are short in size or lifetime while a small number of long bursts carry large amounts of traffic. Conventionally, short-lived flows are referred to as mice and long-lived ones as elephants. For instance in [37], packet traces have been collected at high speed access links and backbone links. The analysis based on the collected data shows that small flows account for [0.851, 0.967] of flows, but [0.704, 0.975] of bytes belong to elephants, with the assumption that the threshold for mice and elephants is 100KBytes. The authors also argue that although the value for the threshold might be set differently or dynamically-based on network conditions, the results still provide solid evidence for the distributions of short-lived and long-lived flows in terms of bytes and flow number.

The mice and elephants phenomenon has some particular features: (1) mice are vulnerable in obtaining bandwidth share in competition with elephants due to the conservative nature of TCP; (2) it is likely for mice to experience longer latency than that expected by users. Unlike elephants, any packet drops will trigger mice to lapse back to slow-start due to less packets being available for fast retransmit and fast recovery. Therefore, mice mostly stay in the slow-start phase of TCP and rarely get enough knowledge about available capacity in the network. The fragility of mice has been observed in the Internet in [37] and the measurements showed the strong correlation between size and rate. The simulation results in Chapter 4 provide further evidence.

6.1.2 Network Context of the ME Strategy

The idea of the ME strategy is to give mice flows preferential treatment. Access networks are areas where congestion is a problem, and is likely to remain a problem because upgrading capacity is expensive and difficult. Premise networks are increasingly populated by gigabyte Ethernet systems and the Internet is provisioned for shared use by a very large number of users, so the link between the access network and the premise network will often form the critical bottleneck. Furthermore, the premise gateway router is the element chosen for implementing the ME strategy. The reason is that, in this part of the Internet, it is clearly feasible to deploy control strategies which distinguish individual flows. Apparently, the premise gateway is not an ideal location for a traffic

management facility. By the time a packet has reached the premise gateway it has probably already passed the worst bottleneck on the path from the server to the client, namely the gateway link. However, it has the advantage that we can expect some surplus processing resources there. Also, dropping packets after they have managed to transit the chief obstacle on their path can be quite beneficial. This is because long flows are fairly well behaved and quite modest dropping rates may be adequate to control these flows sufficiently well to provide good performance for all. Figure 6.1 gives an illustration of the location of a gateway router.

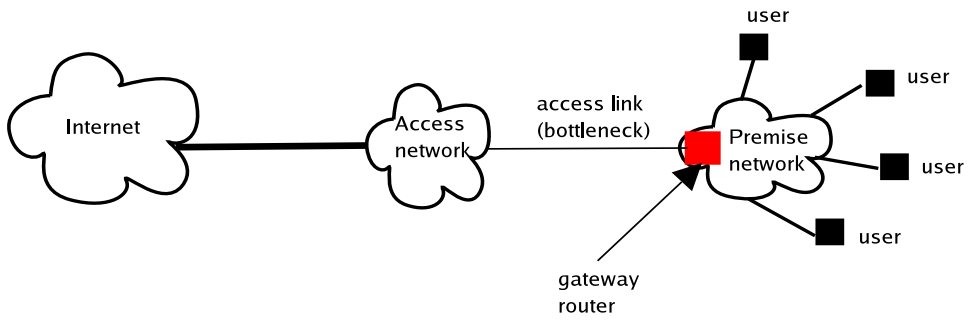


Figure 6.1: Access niches

6.1.3 Operation of the ME Strategy

The ME strategy will be deployed in the premise gateway router. The premise gateway router is equipped with two physical output queues for mice and elephants respectively, and engaged in two functionalities including flow classification and queue management for each queue as shown in Figure 6.2. Individual flows are classified by comparing their flow length with a predefined threshold Th. Any longer flow is marked as an elephant; otherwise, it is a mouse. The mice will continue to be monitored upon each corresponding packet arrival, whereas the elephant packets are sent immediately to the elephant queue. Therefore, there are two databases to keep flow records for mice and elephants, respectively. Also, the two databases are updated every certain time interval T_{db} to delete the records of flows which are no longer active or idle. The default value of T_{db} is 10s. In the queue management module, the chosen scheme for the mice queue is Drop Tail whereas the one for the elephant queue is FLAQM (the detail of

choosing queue management schemes in ME will be given later). The pseudo-code of implementing the ME strategy in a router is given in Algorithm 6.1.

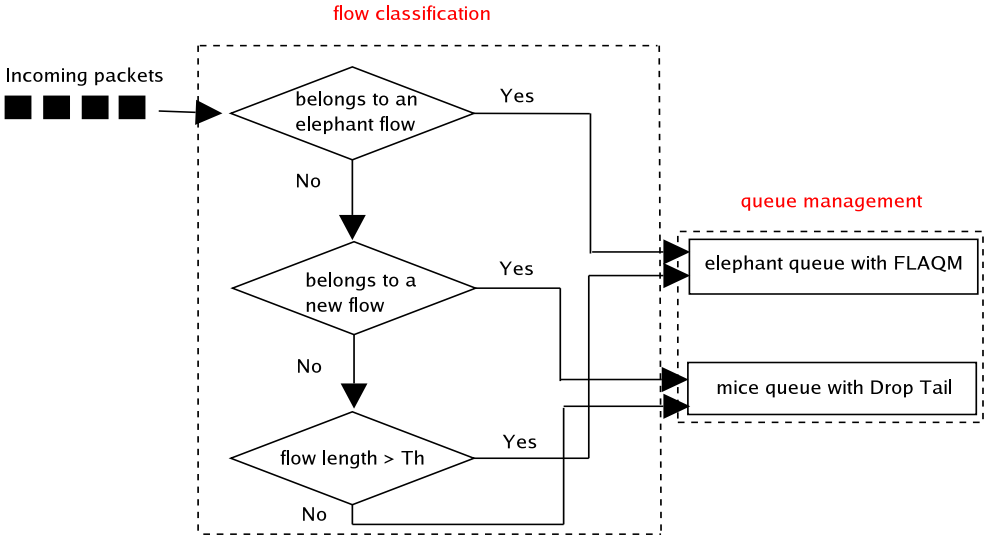


Figure 6.2: The router functions of the ME strategy

The two physical queues use priority scheduling with the mice queue at the higher priority level. To avoid the starvation of elephant packets due to the absolute priority of mice, the file length threshold Th is periodically adjusted to keep the traffic of mice at a certain level of the total bandwidth L_{mice}, such as 40%, based on the network conditions. This way, a minimum capacity is left for elephant packets, while all mice packets have been protected from being dropped. The proposed ME strategy works in the same way as expected by end users in that the longer the flow , the higher possibility of its packets being dropped.

6.2 Router Functions of the ME Strategy

We detail the adjustment of the threshold between mice and elephants in the function module of flow classification and make comments on the selection of the queuing mechanisms for both queues: mice and elephants. Finally, the chosen queuing strategy is compared to RIO.

Algorithm 6.1 The ME algorithm

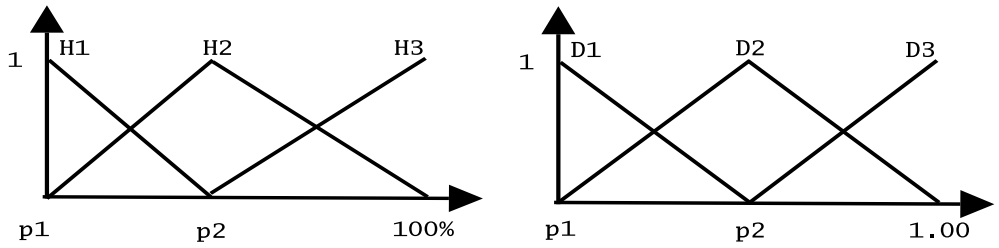
```
if (the incoming packet belongs to an elephant flow)
  put it into the elephant queue with FLAQM queue management
else
  if (the incoming packet belongs to a new flow)
     add this new flow state to the mice database
     update the flow length of this new flow
     put the incoming packet into the mice queue with Drop Tail queue
     management
  else
     if (the total flow length adding the incoming packet is beyond Th)
        delete the flow state from the mice database
        add the flow state to the elephant database
        put the incoming packet into the elephant queue with FLAQM queue
        management
     else
        update the flow length of the corresponding mice flow
        put the incoming packet into the mice queue with Drop Tail queue
        management
     endif
  endif
endif
```

6.2.1 Adjustment of the Threshold between Mice and Elephants using FL

We use FL to adjust Th with an AIMD policy with the fraction of mice traffic load Fr out of the ideal range $[L_{mice}-\delta, L_{mice}+\delta]$, where δ is a constant. Similarly to our design of FLAQM in Chapter 5, there are two FLCs, namely the AI and MD controllers. We qualitatively describe Fr greater than $L_{mice}+\delta$ as $Hi, i = 1, 2, 3$ increasing with i, while lower than $L_{mice}-\delta$ as $Li, i = 1, 2, 3$ also increasing with i. When Fr is greater than $L_{mice}+\delta$, the MD control of Th is used and the multiplicative coefficient k is characterized with three properties of $Di, i = 1, 2, 3$.

On the contrary, the AI control of Th is applied and the increment ΔTh is described with three properties of $Ii, i = 1, 2, 3$. Both Di and Ii increase with i. These membership functions of the AI and MD controllers are illustrated in Figure 6.4 and Figure 6.3, respectively. The corresponding fuzzy rules are given in Table 6.1 and Table 6.1, respectively.

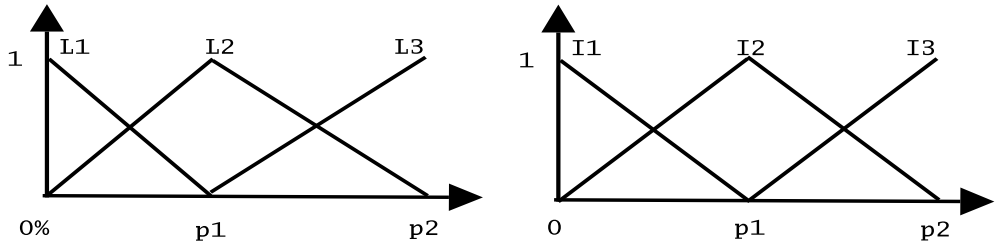
Note that the time interval T_{th} for updating Th is empirically chosen in our simu-



(a) Membership functions for input Fr

(b) Membership functions for output k

Figure 6.3: The MD controller of Th



(a) Membership functions for input Fr

(b) Membership functions for output ΔTh

Figure 6.4: The AI controller of Th

Table 6.1: FL rules of the MD controller of Th

Fr	$H1$	$H2$	$H3$
k	$D3$	$D2$	$D1$

Table 6.2: FL rules of the AI controller of Th

Fr	$L1$	$L2$	$L3$
ΔTh	$I3$	$I2$	$I1$

lations as 5s.

6.2.2 Selection of Queue Management in ME

We use Drop Tail to manage the mice queue since the packets are almost always a small portion of traffic and also should be protected from any packet drop. On the other hand, any effective AQM strategy can be a candidate for conducting queue management for the elephant queue. The selected AQM scheme is expected to be able to take account of the capacity consumption of mice in order to perform precise control on elephants. The proposed FLAQM scheme in Chapter 5, which meets this criterion, is chosen to be integrated into the ME strategy. As a result, the ME strategy only drops or marks packets from elephant flows, which are the main contributor of congestion and are not sensitive to connection latency. Additionally, we would like to stress that packet re-ordering is unlikely to happen with the proposed ME strategy.

6.2.3 Comparison with RIO

Preferential treatment for short flows has also been deployed by two papers [54] and [7] as mentioned in Chapter 2. In both schemes, edge routers first classify individual flows to be either mice or elephants based on a file length threshold and mark the packets in mice as IN and elephants as OUT. Core routers then give higher priority to IN packets via RIO queue management combined with FIFO scheduling. Our proposed ME strategy shares the same idea as these two methods. However, instead of giving implicitly preferential treatment to mice via RIO, priority scheduling is employed. We argue that the configuration of RED is still an open issue in the networking area, and

so is the combination of the two groups of parameter settings of RIO. However, our implementation avoids this controversial issue and any potential associated drawback. On the other hand, one FIFO queue might increase the delay of mice packets and affect their performance, since a timely response is more crucial to mice than elephants.

6.3 Experiments

In this section, we investigate by means of experimental simulations the performance of the proposed ME strategy by comparing with that of queue management strategies at edge routers dealing with traffic accessing customer premises. The network topology described in Chapter 4 is applied for the simulations.

6.3.1 Configuration of the ME Strategy in the Gateway

To implement the ME strategy at the gateway, a virtual link and a virtual buffer are introduced, each of which has less capacity than the real bottleneck, i.e. the link between the Internet edge router and the gateway. Choosing a small value for the virtual link bandwidth wastes network resources, whereas with a big value the ME strategy can not perform effectively due to the failure of transfer congestion from the real bottleneck to the virtual link. A capacity slightly less than the real bottleneck bandwidth is chosen for the virtual link capacity in the ME strategy, along with a small value for the virtual buffer, to maintain high network throughput. Besides, FLAQM is used in the real bottleneck to gain low queuing delay and effective dropping of elephant packets. This way, elephant connections are regulated to maximize the network utilization and at the same time mice flow through the network without delay.

In the simulations, the virtual bottleneck built in premises R3 is realized by literally adding a virtual router VR between R2 and R3. The router VR has link bandwidth of 1.45Mbps, buffer size of 150 packets with 30 packets for the mice queue and 120 packets for the elephant queue, and propagation delay of zero.

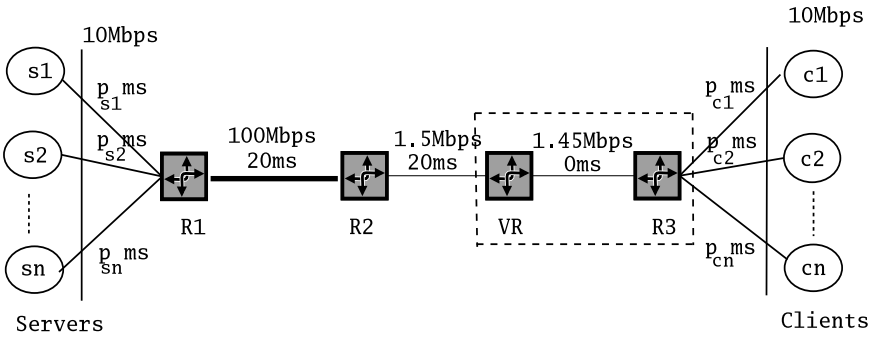


Figure 6.5: Network topology with the built-in virtual bottleneck

6.3.2 Implementation

To implement the ME strategy at the gateway router, mice packets are put into a separate queue that has priority, and used FLAQM to control the queue for the remaining packets (the elephant queue). This combined strategy will be referred to as ME+FLAQM. The parameter settings for FLAQM are the same as that in Chapter 5, with the one exception that the target queue length is set as 40 packets.

The traffic pattern of simulations in Chapter 5 is reused here, in that the combination of truncated Web traffic and extremely long FTP bursts is mimicked with the number of extremely long FTP bursts varied from 5 to 40 to create different congested traffic conditions. The initial threshold between mice and elephants has been set to 20,000 bytes. Note that the simulations have been run for 600s and the first half is regarded as a warmup period. Five independent replications of a simulation are carried out to perform statistical analysis. Two set of simulations have been conducted for both with and without ECN marking.

6.3.3 Results

Performance is evaluated without ECN marking and with ECN marking, respectively.

6.3.3.1 Without ECN

The performance of ME at the premise gateway is compared to that of Drop Tail, ARED, FLAQM, and ME at the edge router without ECN. The simulation results show that:

- The network throughput performance comparison is plotted in Figure 6.6. To deploy the ME strategy at the gateway, the virtual bottleneck capacity is set about 3% lower than the real bottleneck capacity. However, the network throughput performance with ME at the gateway is no less than that of Drop Tail with more than 20 extremely long bursts, with a decrease of at most 3% otherwise. With ME at the Edge, the network throughput reached is similar to that of FLAQM.

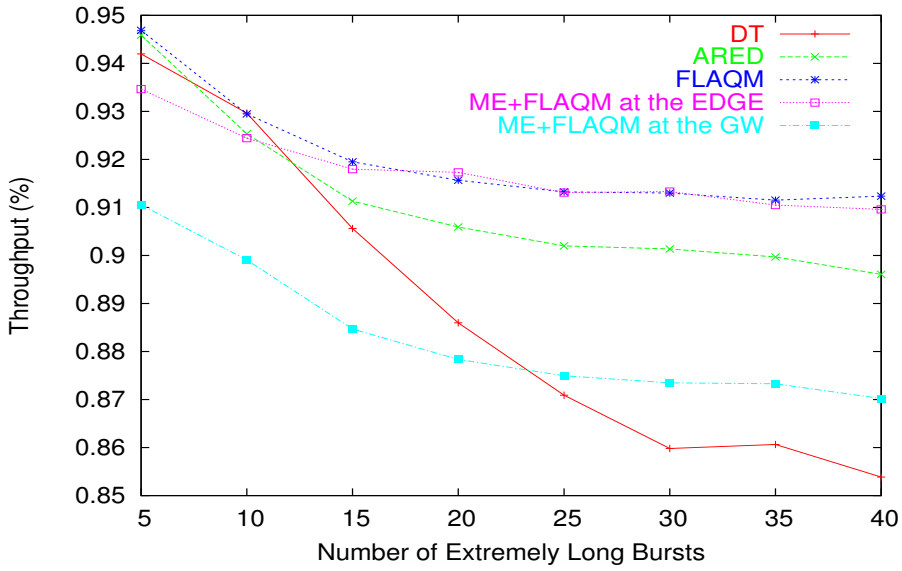


Figure 6.6: Network throughput

- In terms of TCP goodput, ME at the gateway has significantly improved the performance of mice in comparison with Drop Tail, ARED, and FLAQM working at the edge router. This improvement is shown by the weighted average TCP goodput value in Figure 6.7 for TCP goodput of total traffic, Figure 6.8 for TCP goodput of mice, and Figure 6.9 for TCP goodput of elephants [1]. It is also shown by the relative TCP goodput versus file lengths in Figures 6.10, 6.11, 6.12, and 6.13 with different traffic load conditions. More importantly, it has been achieved without sacrificing the TCP goodput of elephants. In fact, the elephant goodput approaches that of FLAQM. Moreover, FLAQM outperforms ARED at the edge router in terms of TCP goodput.

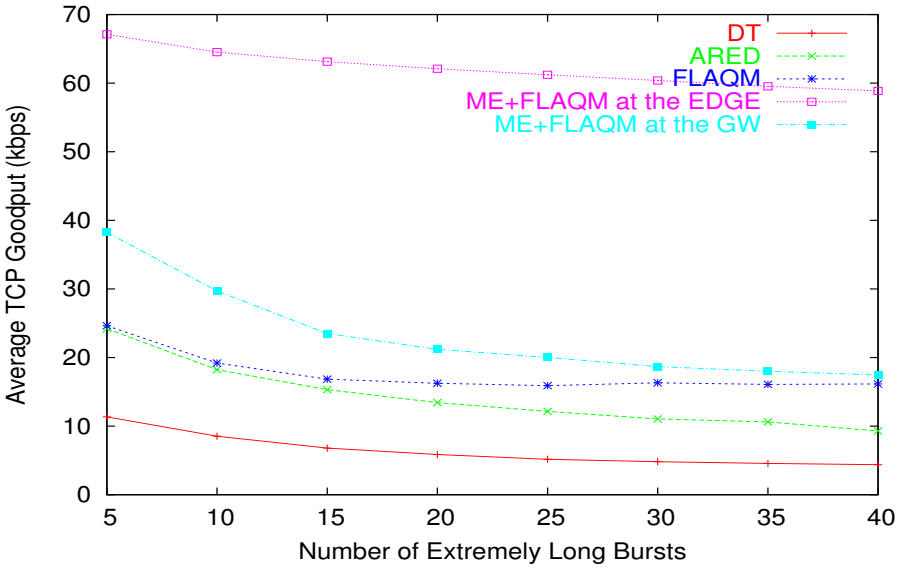


Figure 6.7: Weighted average TCP goodput

[1] The threshold for mice and elephants is 15 packets

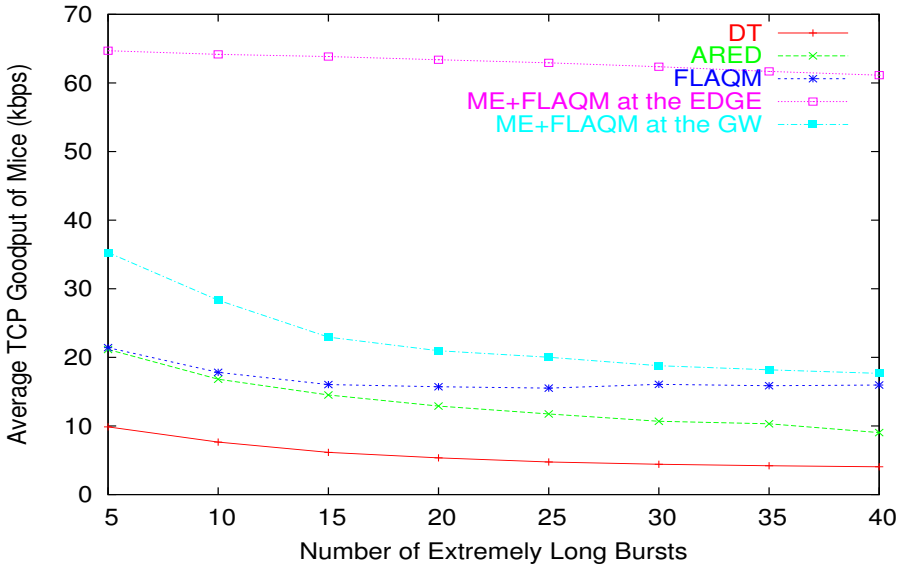


Figure 6.8: Weighted average TCP goodput of mice

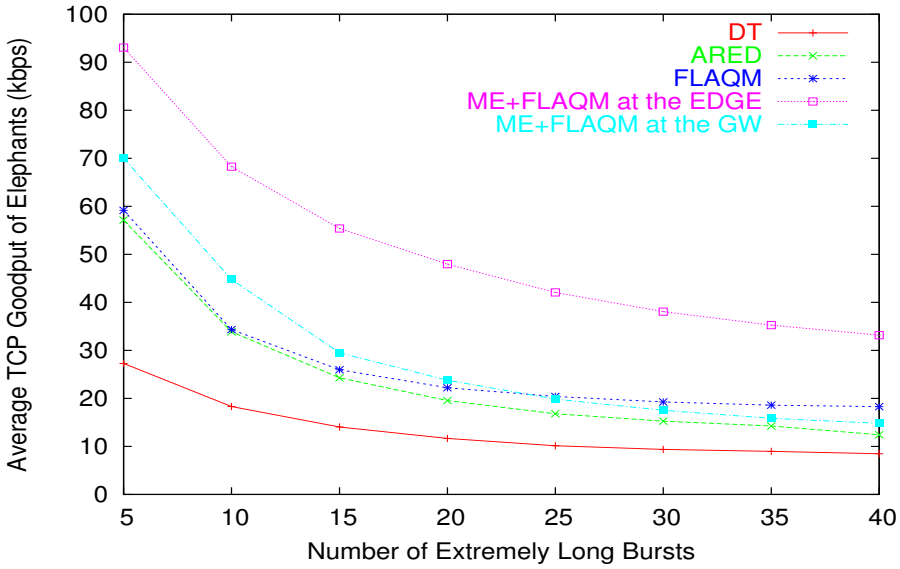


Figure 6.9: Weighted average TCP goodput of elephants

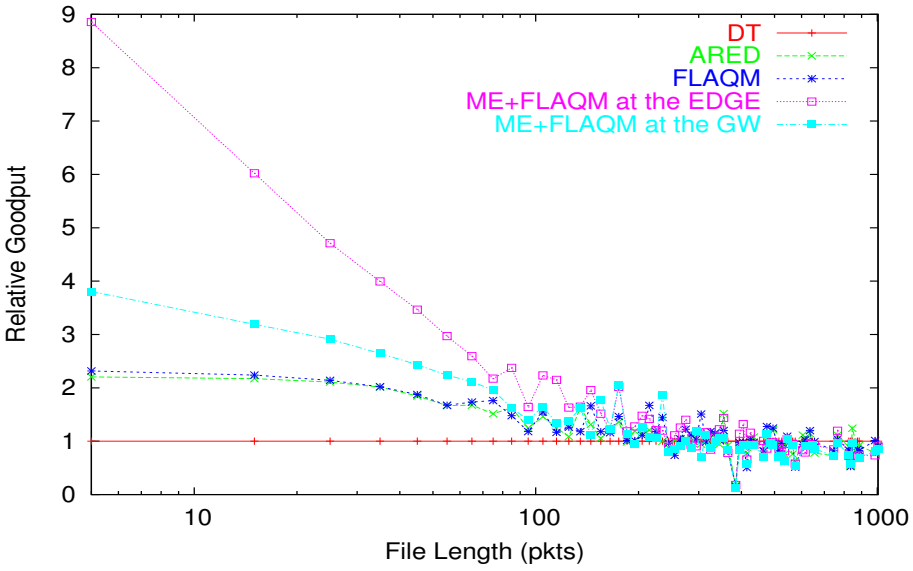


Figure 6.10: Relative user TCP goodput with n=10

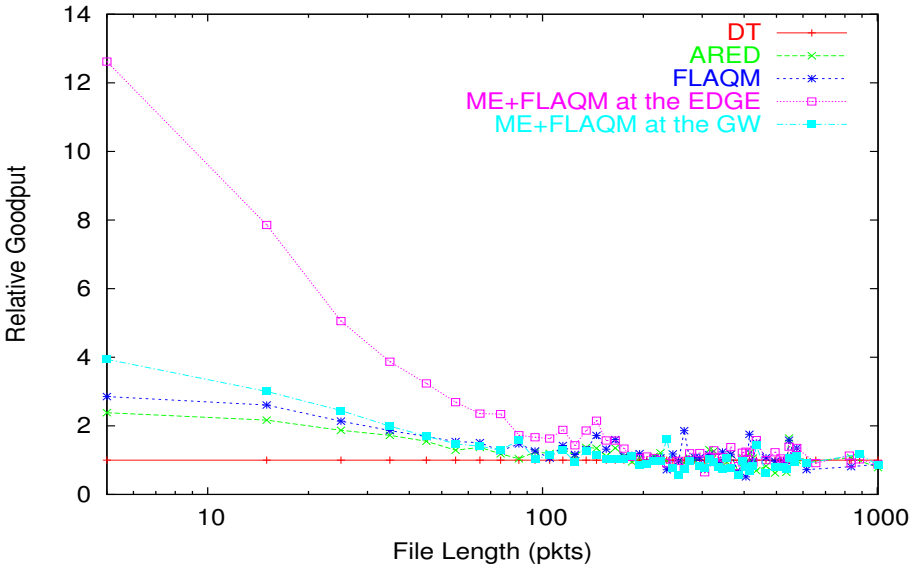


Figure 6.11: Relative user TCP goodput with n=20

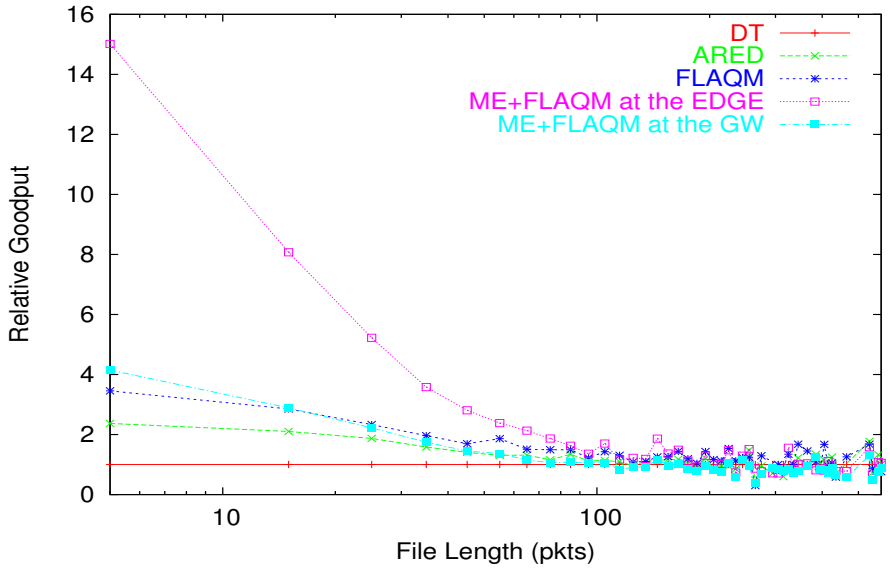


Figure 6.12: Relative user TCP goodput with n=30

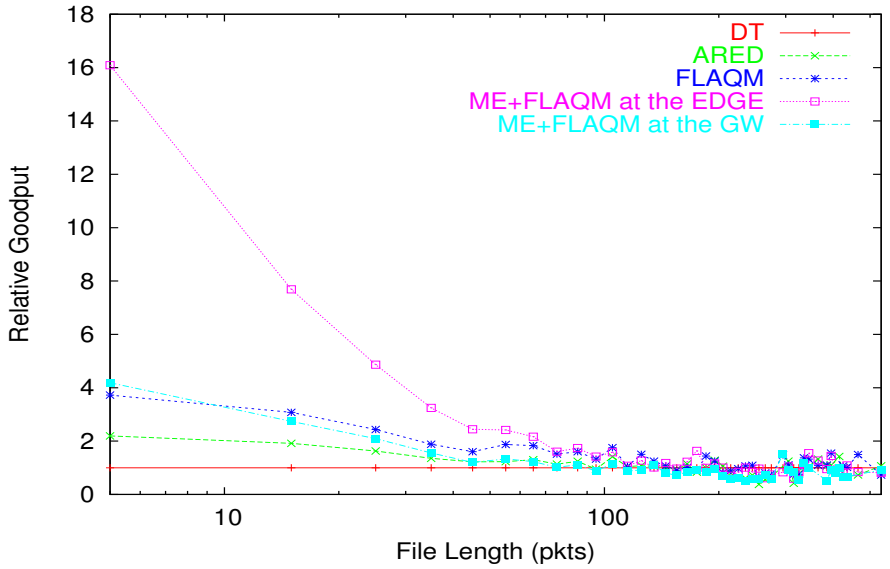


Figure 6.13: Relative user TCP goodput with n=40

- In terms of response time, ME at the gateway has also significantly improved the performance of mice in comparison with Drop Tail, ARED, and FLAQM working at the edge router. This improvement is shown by the weighted average response time value in Figure 6.14 for response time of total traffic, Figure 6.15 for response time of mice, and Figure 6.16 for response time of elephants, and by the relative response time versus file lengths in Figures 6.17, 6.18, 6.19, and 6.20 with different traffic load conditions. Moreover, FLAQM outperforms ARED at the edge router in terms of response time.

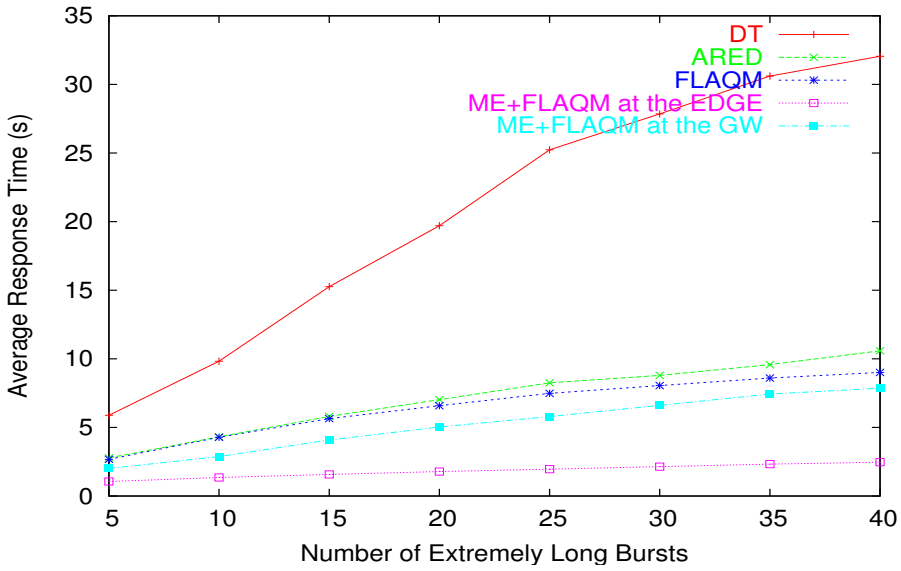


Figure 6.14: Weighted average response time

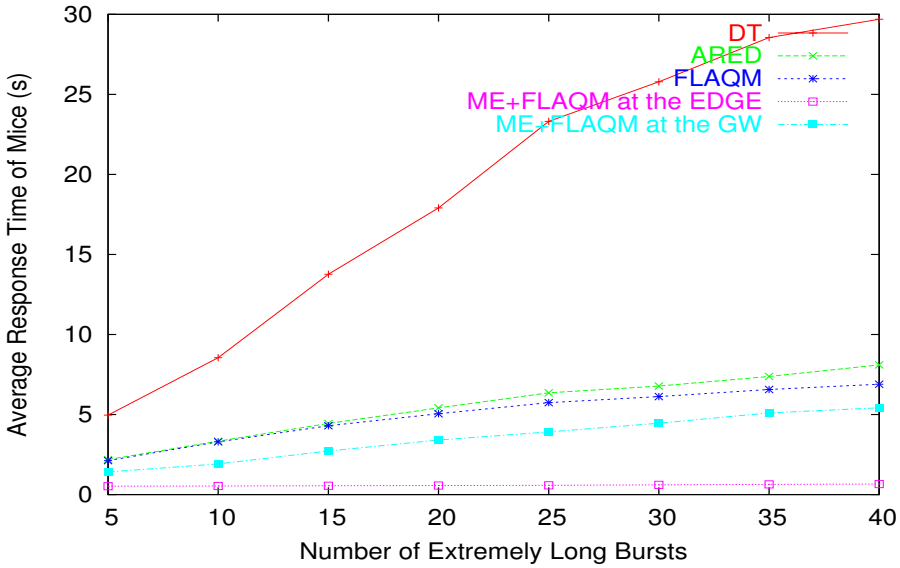


Figure 6.15: Weighted average response time of mice

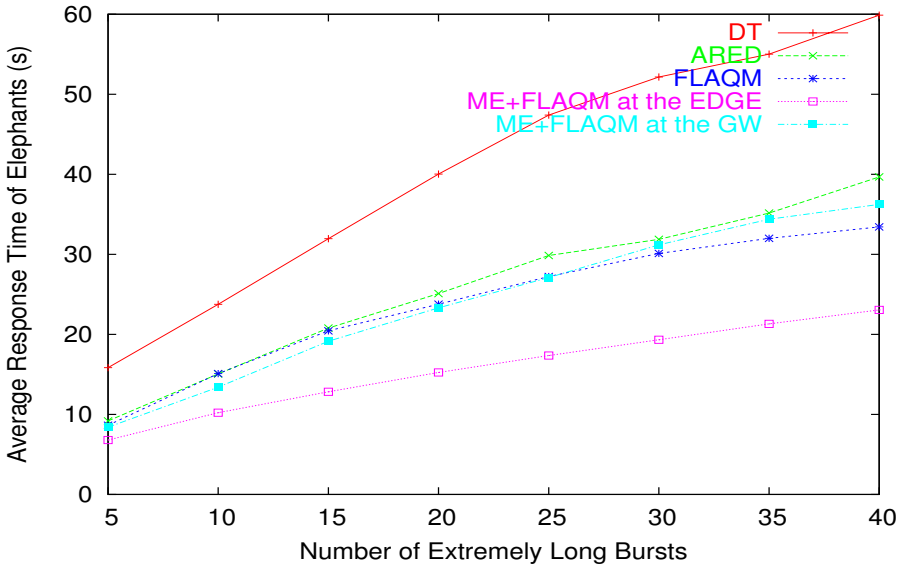


Figure 6.16: Weighted average response time of elephants

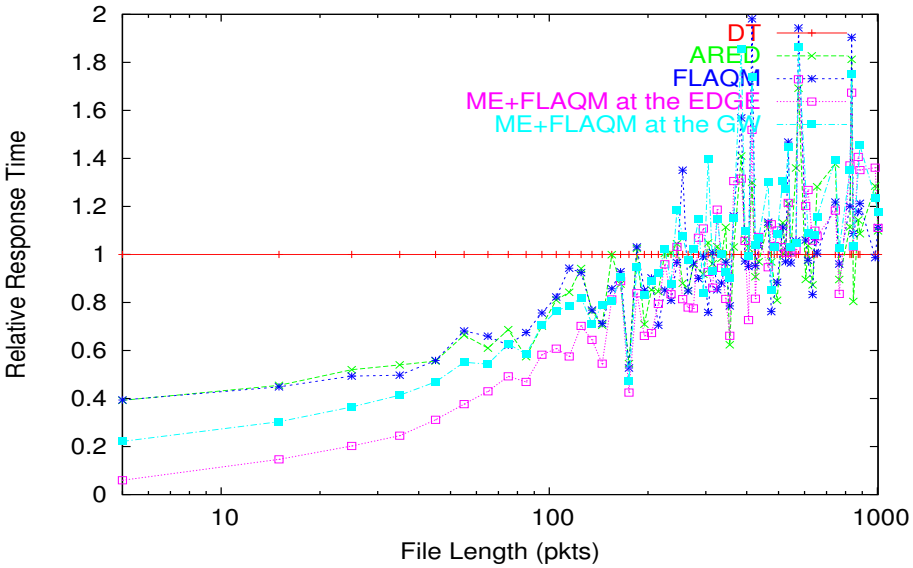


Figure 6.17: Relative use response time with n=10

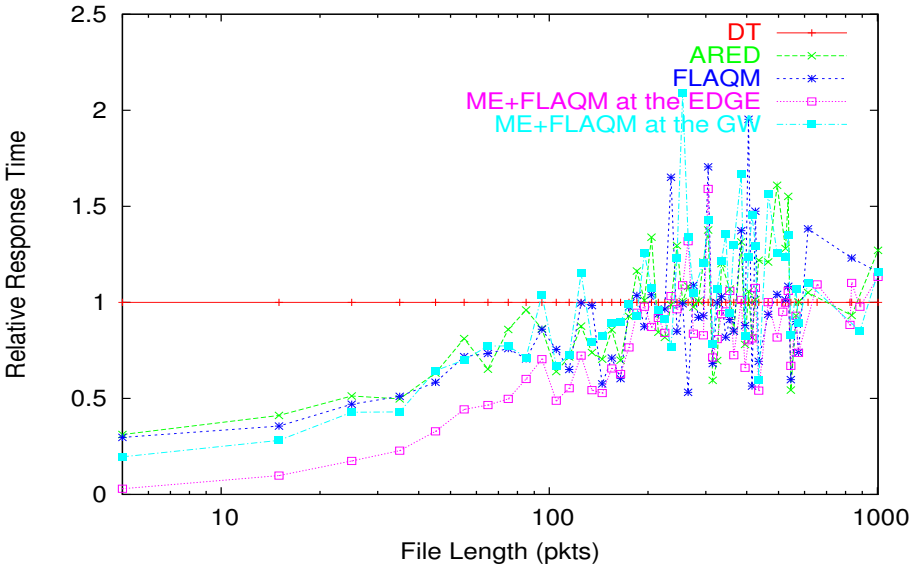


Figure 6.18: Relative user response time with n=20

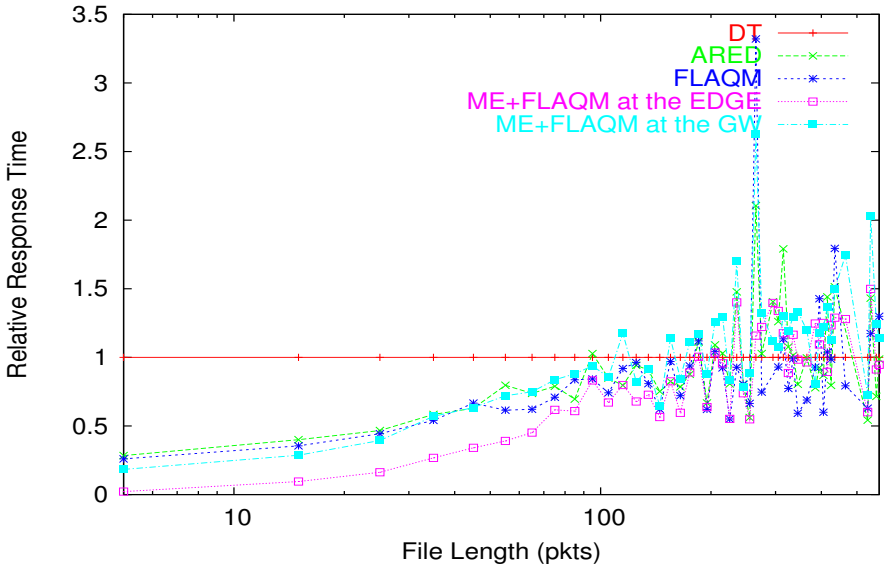


Figure 6.19: Relative user response time with n=30

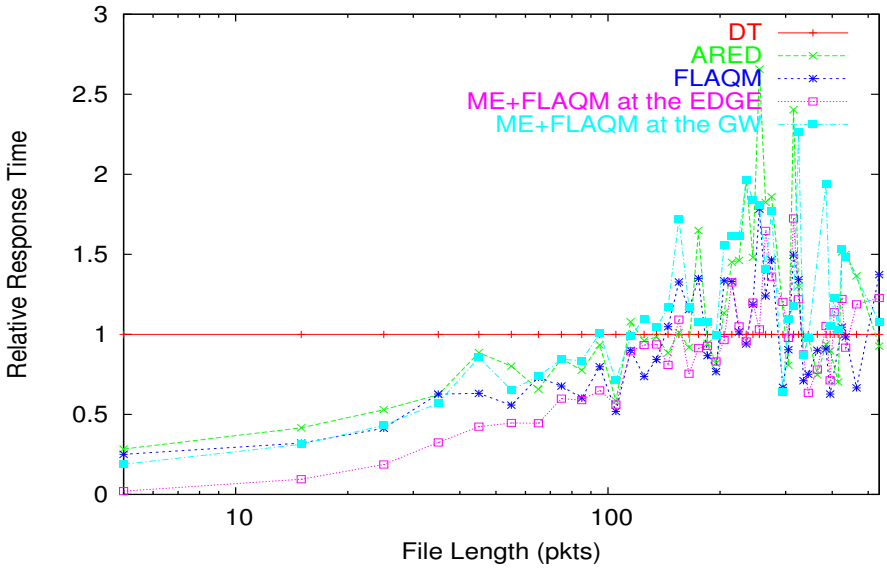
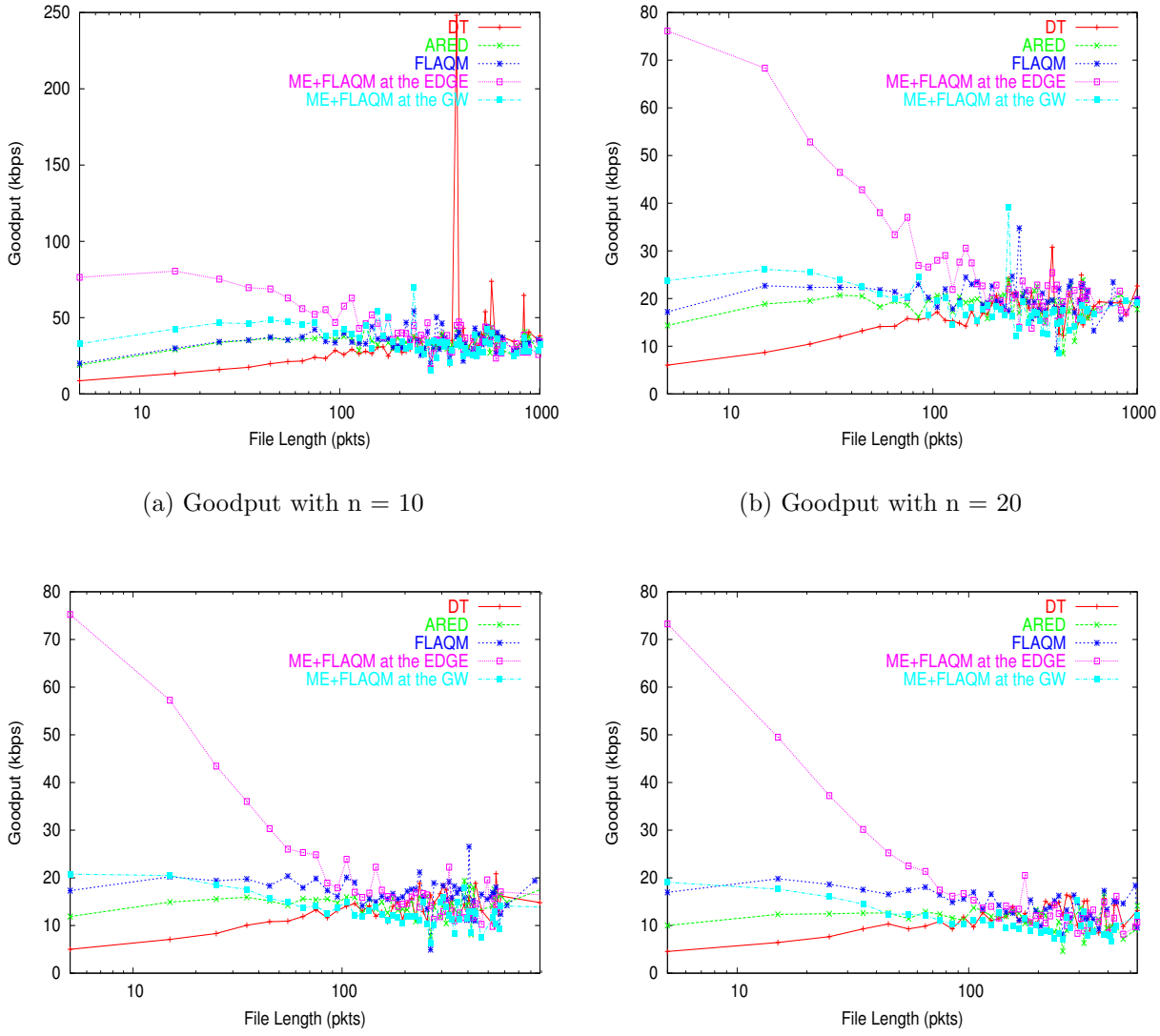


Figure 6.20: Relative user response time with n=40

- The goodput of mice has been improved by ME at the gateway over their longer counterparts compared to Drop Tail, ARED, and FLAQM at the edge router, as shown by the absolute performance of user TCP goodput versus file lengths with different traffic loads in Figure 6.21. It is clear that Drop Tail has a strong bias against mice in terms of TCP goodput.



(a) Goodput with n = 10

(b) Goodput with n = 20

(c) Goodput with n = 30

(d) Goodput with n = 40

Figure 6.21: User TCP goodput performance

- ME enhances the possibility of all flows for data transfer. With heavy-loaded traffic, it is possible that within a considerable period there are some flows that keep still in terms of data transfer, given the loss of SYN or SYN ACK packets.

The still flow percentage has been measured and plotted in Figure 6.22. Especially, ME at the gateway has a lower percentage of still flows compared with Drop Tail, ARED, and FLAQM at the edge while ME at the edge has nil still flows.

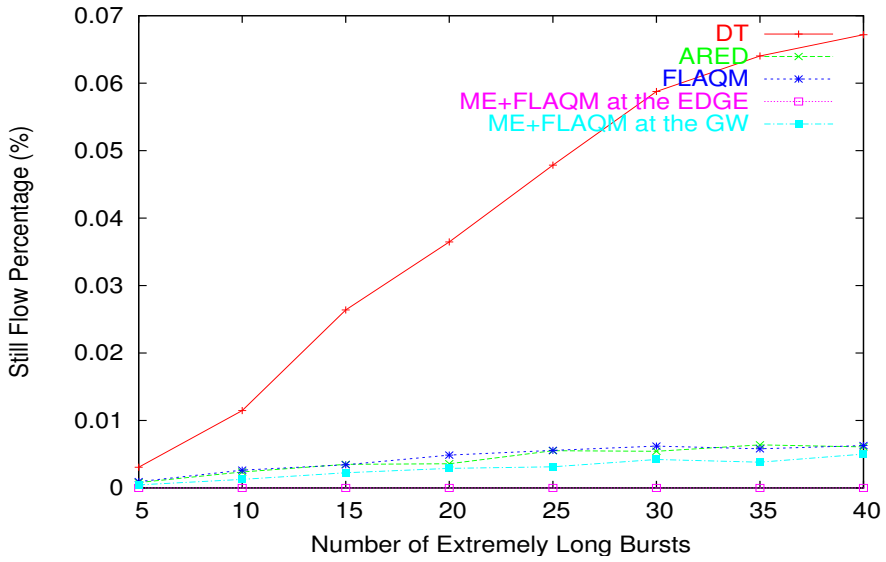


Figure 6.22: Still flow percentage

- As expected, ME at the edge has outperformed ME at the gateway in that it improves the performance of both mice and elephants while also increasing the whole network throughput greatly over Drop Tail in all investigated traffic conditions. However, to fully utilize bandwidth using FLAQM, the real bottleneck queue with the strategy of ME at the gateway is almost always not empty. This inevitably results in longer delays and lower TCP goodput for mice.

6.3.3.2 With ECN

In this section, ECN marking is used instead of the dropping mechanism with ARED, FLAQM, and ME either at the edge or the gateway.

As expected, using ECN marking improves the network throughput, the user performance in terms of TCP goodput and response time, and still flow percentage of each investigated scheme over those of without ECN except Drop Tail. The network throughput performance is plotted in Figure 6.23.

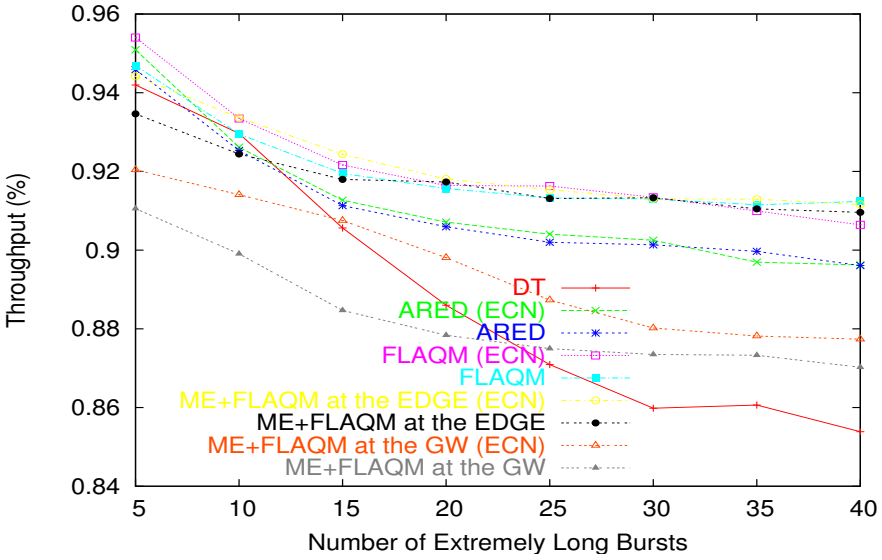


Figure 6.23: Network throughput

The weighted average TCP goodput is illustrated in Figure 6.24 for total traffic, Figure 6.25 for mice, and Figure 6.26 for elephants.

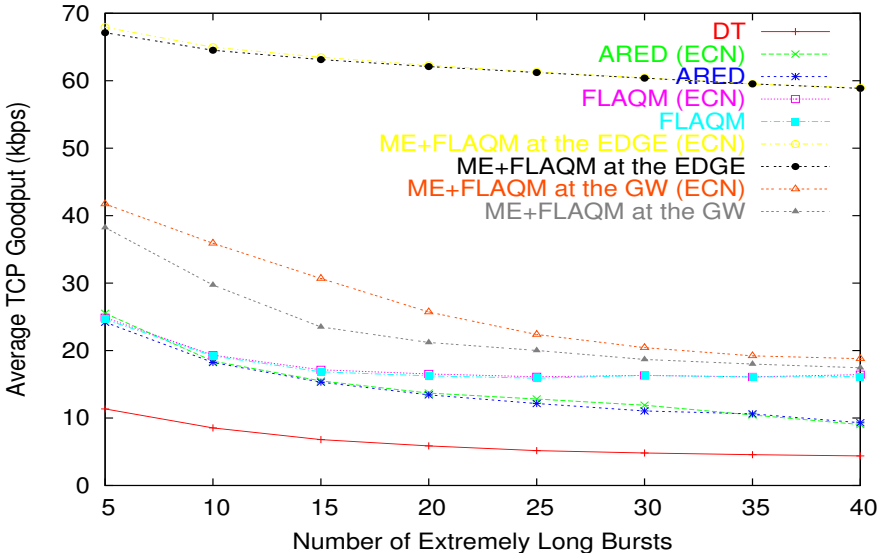


Figure 6.24: Weighted average TCP goodput

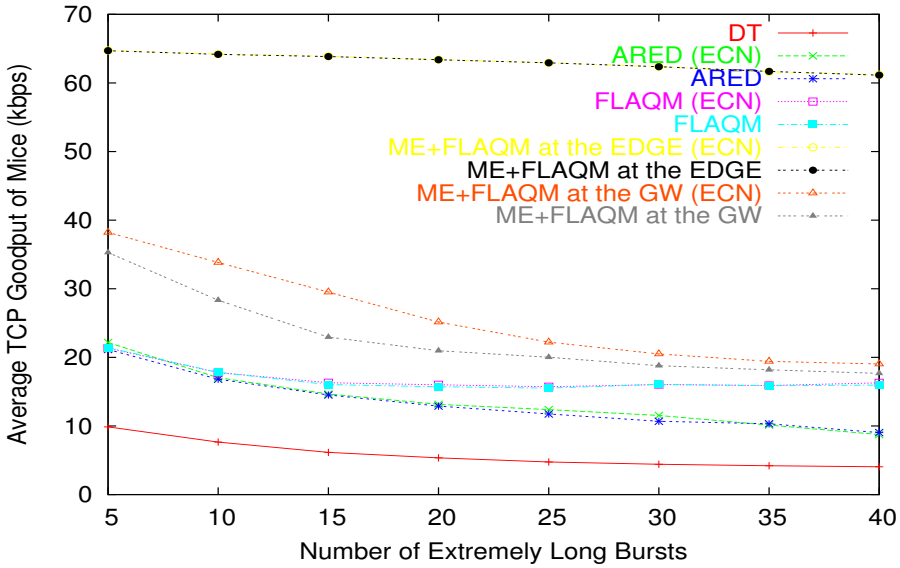


Figure 6.25: Weighted average TCP goodput of mice

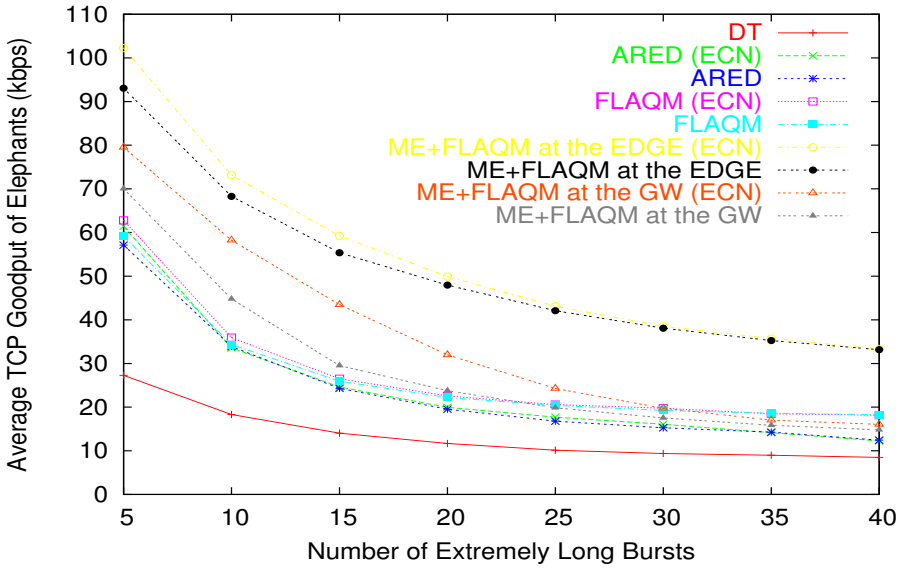


Figure 6.26: Weighted average TCP goodput of elephants

Meanwhile, the performance of response time is shown in Figure 6.27 for total traffic, Figure 6.28 for mice, and 6.29 for elephants. In addition, the performance of still flow percentage is illustrated in Figure 6.30. Compared to other schemes, the enhancement in performance of ME at the gateway with ECN marking is significant over that of the strategy without ECN marking.

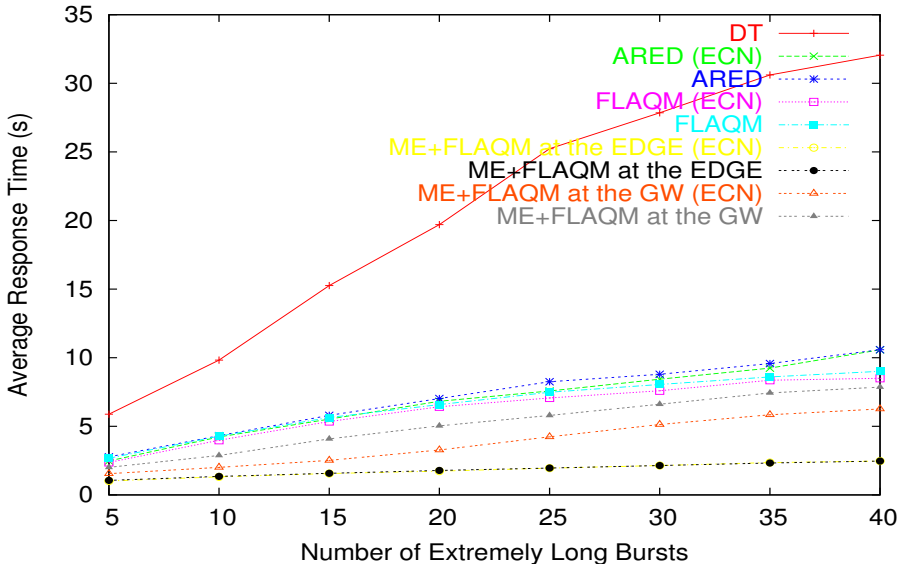


Figure 6.27: Weighted average response time

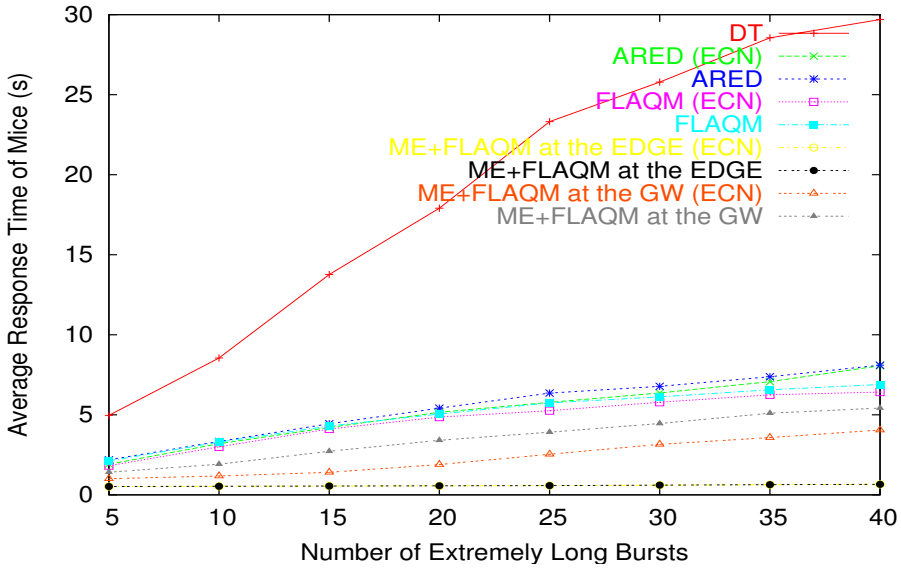


Figure 6.28: Weighted average response time of mice

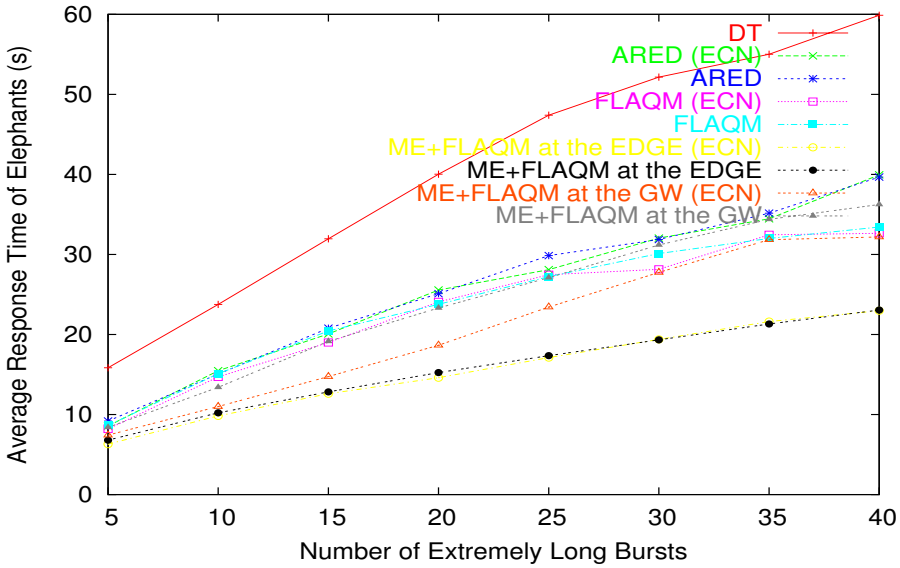


Figure 6.29: Weighted average response time of elephants

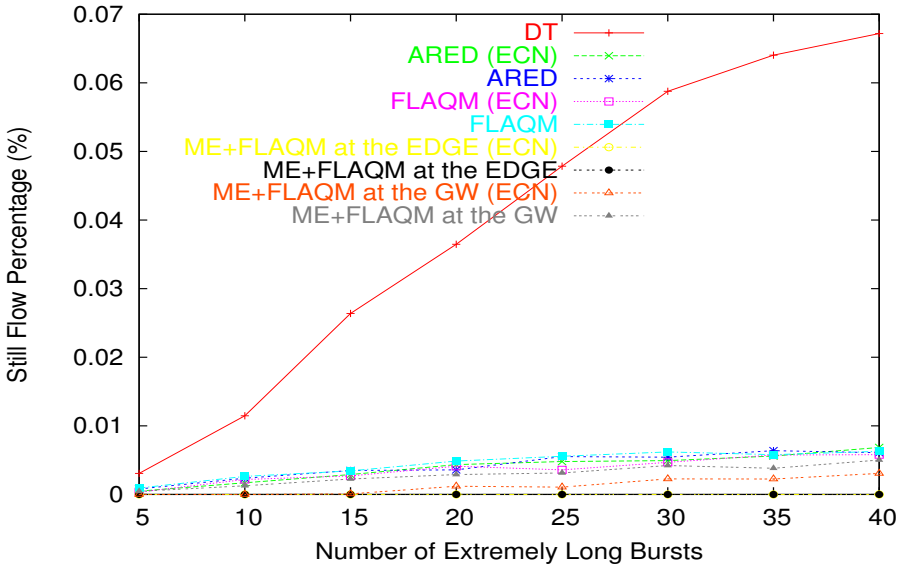


Figure 6.30: Still flow percentage

6.4 Result Analysis

We analyze the results in both quantitative and qualitative aspects.

6.4.1 Quantitative Approach

Based on the simulations, the following conclusions can be drawn about the performance of ME at the gateway:

- In comparison with Drop Tail, ARED, and FLAQM working at the edge, ME at the premise gateway does considerably improve the performance with mice in terms of TCP goodput and response time. The network throughput of ME at the gateway is competitive to that of Drop Tail. Moreover, the ME strategy increases the opportunity for data transfer to avoid flows being still.

- With ECN marking, the advantages of the ME strategy at the gateway become outstanding. The increase in the network throughput of ME at the gateway is dramatic: about 1% compared to that of ARED, FLAQM, and ME at the edge, and thus the performance difference between ME at the gateway and these schemes is reduced.

- As expected, the edge is an ideal place for ME. The network throughput of ME at the edge is higher than at the gateway by about 3%. However, there is a tradeoff between performance and operational feasibility.

6.4.2 Qualitative Approach

In the following, the qualitative explanation is given as to why ME performs better than those queue management schemes with FIFO scheduling including Drop Tail, ARED, and FLAQM.

6.4.2.1 Advantages of the ME Strategy

Firstly, it absolutely gives higher priority to mice and the first few packets of elephants, and thus the SYN and SYN acknowledgment (ACK) packets of all the flows will safely reach their destination. By comparison, Drop Tail and RED could not avoid dropping

these packets. Therefore, the proposed method increases user throughput of all flows in this aspect.

Secondly, mice are vulnerable to any dropped packets. ME protects them from this vulnerability. Any packet drop of mice will make TCPs fall back to slow start, since there are not enough packets for them to have three duplicate ACK packets. Furthermore, the retransmission timeout (RTO) will be doubled. With Drop Tail, RED, and FLAQM, the packet dropping of mice will likely occur when the traffic load is heavy. Therefore, the throughput of mice is influenced by the packet drop and double RTO. Note that in our simulations, the number of flows is fixed during the simulation interval. That is, there are a fixed number of tasks which queue management schemes need to deal with. Based on the above analysis, the proposed method ME is able to finish the tasks by more quickly transmitting mice packets than the others as shown in Figure 6.31, with the number of FTP long bursts set at 20 for example.

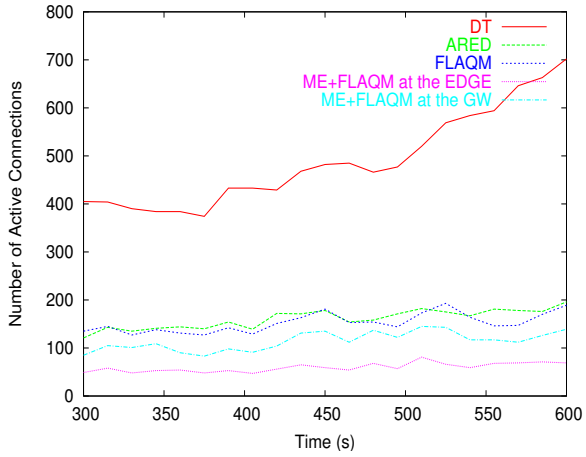
Thirdly, active queue management (AQM) performs better in controlling elephants alone. Because of the conservative nature of TCP, mice almost stay in the slow start phase, and never achieve their fair bandwidth share in competition with elephants. Any drop of mice packets will worsen performance and at the same time elephants might speed up and thus deteriorate network conditions. Without the interference of mice, FLAQM control only elephants, and it does the job more precisely.

6.4.2.2 Obtain Better Control on Elephants by Considering Mice Traffic

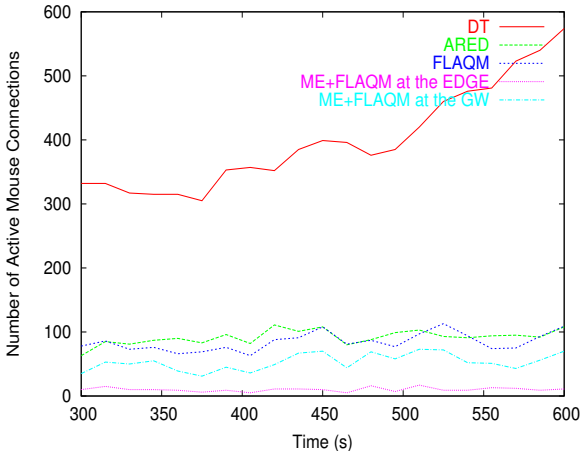
We implement ME strategy combined with FLAQM. It is easy for FLAQM to take account of the current traffic load from not only elephants but also mice and determine dropping probability for elephants in the next time period accordingly. In fact, FLAQM has the same philosophy as RIO in [9].

6.4.2.3 Remaining Issues

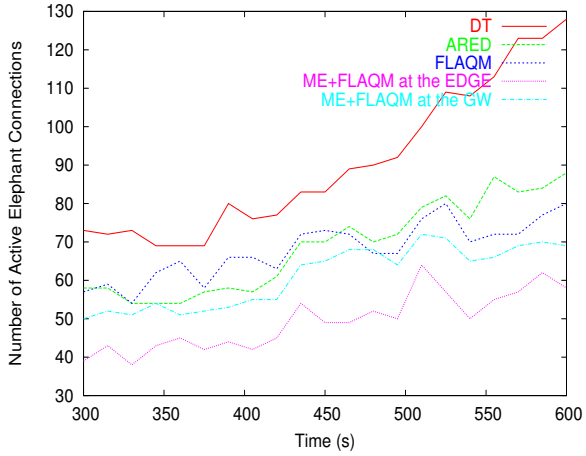
Artificially dropping packets after receiving them from the real bottleneck certainly degrades the performance of ME with regard to network throughput, TCP goodput, and flow latency compared with ME working at the upstream of the bottleneck link. Apparently it is impossible for ME at the gateway to reach the performance of ME at the edge. However, since the premises gateway is a practically plausible choice



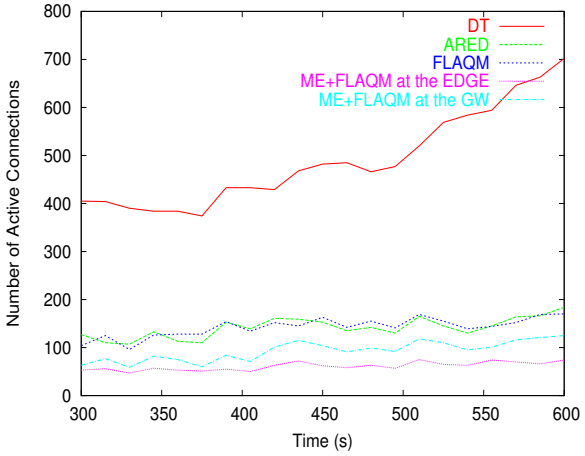
(a) Total



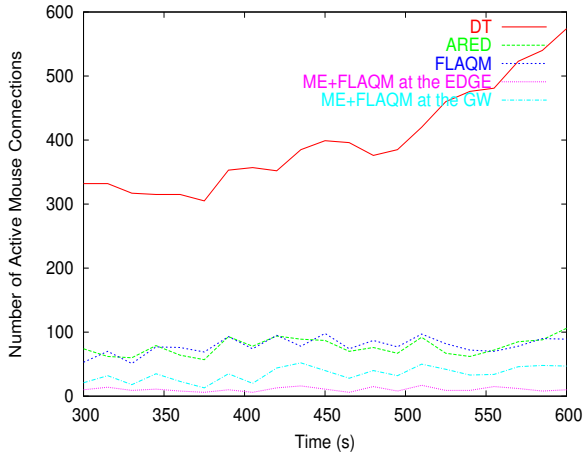
(b) Mice



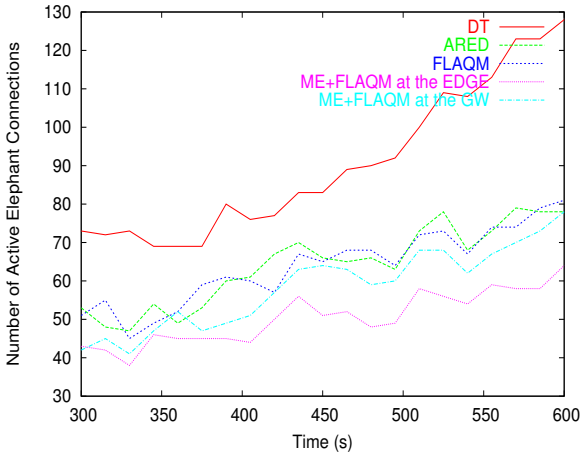
(c) Elephants



(d) Total with ECN



(e) Mice with ECN



(f) Elephants with ECN

Figure 6.31: Number of active connections

for implementing ME, efforts have to be made to approach the ideal performance of ME working at the edge. The key point to this issue is to correctly select the virtual bottleneck parameters.

Virtual bottleneck parameter selection of ME at the gateway is a tradeoff between network throughput and the effectiveness of the ME strategy. A small value of both virtual buffer size and link bandwidth would result in the complete relief of congestion on the real bottleneck link and thereby network traffic is totally controlled by the ME strategy working at the gateway. However, link capacity could not be fully utilized and network resources might be wasted.

6.5 Summary

The goal of a multiservice Internet which can provide statistical guarantees of service has not yet been attained. The direct approach of building protocols into the Internet which support differentiated service is complex and some way off being implemented. In this chapter we have investigated a strategy which differentiated different classes of traffic on the basis of traffic behaviour, thereby, as it turns out, producing better performance for all traffic, but especially for traffic formed of short-lived flows.

We have also shown that it is feasible to implement an effective control strategy for providing better overall performance at the most *convenient* location – the gateway router in an access niche. In order to create a comprehensive architecture for congestion control and performance protection, it will be necessary to consider similar strategies in other routers as well. However the link between a premise and the Internet is of particular importance, and so the fact that we can implement a strategy for controlling congestion over this link in the gateway router is important.

Chapter 7

Conclusions and Future Work

7.1 Introduction

In the current Internet, the sophisticated TCP protocols located at end users are the primary contributor to congestion control and Internet robustness. The TCP congestion control scenario involves TCP probing available network resources by dynamically changing its sending window size in response to packet reception status, and increasing its window size until it detects a packet loss. Packet losses are mostly due to congestion occurrence and subsequent full output buffers in the network. However, the Drop Tail queuing strategy produces low network throughput, high flow latency, and low responsiveness to bringing the network back to steady states. Studies have shown that these problems can be solved by proactively reacting to incipient congestion in the routers via active queue management (AQM).

The design of an AQM scheme includes the selection of appropriate congestion indicators and control principles, and the calculation of dropping probability. Among these, choosing congestion indicators is pivotal for the performance of a given AQM scheme, since congestion indicators are in charge of accurately detecting the levels and trends of congestion. Significant efforts have been made in AQM from different standpoints including experience, statistics, optimization theory, and control theory.

Research also shows that short-lived flows or mice are vulnerable in terms of bandwidth share due to the conservativeness of TCP. Although AQM can improve performance for this majority number of traffic flows to some extent, the performance is still much below the user expectation of the timely delivery of mice.

7.2 Our Approach of Using FL

In this thesis, we started with a discussion of some existing AQM schemes and presented the state of the art in the field of queue management. Following this, we have explored the application of fuzzy logic (FL) for AQM – or FLAQM. Our hypothesis is that FL is able to be applied to AQM on the basis of its capability of integrating domain expert knowledge and conducting robust control of such nonlinear complex systems as the Internet without the needs of precise and comprehensive information and the mathematical model of objects being controlled. Two algorithms, FLAQM(I) and FLAQM(II), have been presented. More importantly, the traffic load factor is elaborately chosen for congestion notification in FLAQM. The traffic load factor takes account of not only input traffic loads but also existing queue lengths, and thus makes accurate calculation of current traffic load. With such a dimensionless measure, control is expected to be robust against link capacity changes. Also, the calculation of traffic load factor is easily extended to deal with the scenario where best-effort traffic coexists with other QoS traffic with reserved bandwidth, and available capacity for best-effort traffic is ever-changing. FLAQM(I) employs the traffic load indicator and its change as its inputs, whereas its reciprocal and the corresponding change are used in FLAQM(II). The use of the reciprocal of the traffic load indicator in FLAQM(II) is to implicitly realize the input normalization to achieve system stability and robustness.

Meanwhile the mice and elephants (ME) strategy has been proposed in this study to provide preferential treatment for vulnerable short-lived flows, or mice, in order to meet the user expectation of timely delivery of this kind of flow. Besides giving priority to mice, the proposed ME strategy also considers issues including operational location, starvation of elephants, storage of flow states, and control of elephants. We have identified that user premise gateways, despite not being an ideal place, is a practical location at which to employ the ME strategy, with expectation of some surplus processing resources there. To address the starvation problem, minimum capacity is ensured for elephants by updating the threshold between mice and elephants via FL technology. Once flows are detected as complete or inactive, the states will be deleted to minimize the size of data storage. Due to its effectiveness and flexibility in being extended to an environment where different traffic classes coexist, the proposed FLAQM is deployed

to control the elephant queue.

7.3 Experiment Results and Analysis

Performance evaluation of the proposed schemes is carried out via extensive experimental simulations in this research. Performance is evaluated under a variety of traffic conditions against a number of performance measures: TCP goodput and flow latency from the user perspective and network throughput, link utilization, and loss rate from the ISP perspective. The simulation results demonstrate that the traditional Drop Tail method performs worse with increasing traffic load and in contrast AQM is able to relieve congestion and tends to improve traffic performance, especially flow latency. It thereby proved the necessity of replacing Drop Tail with an effective AQM scheme. However, there is no outstanding AQM scheme among the typical ones in existence and, in any case, they are sensitive to the fluctuation of traffic load in realistic network conditions. By applying the proposed FLAQM scheme to queue management, the simulation results show that the traffic performance has been improved dramatically compared to that of both Drop Tail and some well-known AQM schemes in all the investigated network circumstances. More importantly, the fuzzy controllers, in particular the FLAQM(II), are robust in the face of traffic fluctuation and different network conditions. It thus demonstrates the feasibility of applying FL to the area of traffic control in the Internet. We also find that the ME strategy with the use of FLAQM to control the elephant queue improves the performance of mice with no harm to elephants, if there is no benefits.

7.4 Confidence Level Analysis

The issue of credibility of stochastic discrete-event simulation studies has arisen in the research area of telecommunication networks, which is a nonterminating system that runs continuously, or at least over a very long period of time[22]. [24] describes the necessity of the use of appropriate analysis of simulation output data, especially for a

steady-state simulation [1]; otherwise simulation results could be misleading. Since some of a TCP/IP network simulation model input variables are random variables such as flow inter-arrival rate and flow size generated from the Poisson-Pareto traffic model in subsection 4.1.3, it follows that the model output variables are random variables. The simulation results thus exhibit random phenomena. A confidence level on an estimate of an actual value can be established. The most frequently used estimate is the mean value, μ. In this case, the sequence of observations or samples of size n, $x_1, x_2, \ldots, x_n$, is used to calculate the average as follows.

$$\bar{X}(n) = \frac{1}{n}\sum_{i=1}^{i=n} x_i \tag{7.1}$$

The accuracy of the estimator $\bar{X}(n)$ can be assessed by the probability

$$P(|\bar{X}(n) - \mu| < \delta) = 1 - \alpha \tag{7.2}$$

r

$$P(\bar{X}(n) - \delta \leq \mu \leq \bar{X}(n) + \delta) = 1 - \alpha \tag{7.3}$$

where δ is the half-width of the confidence interval for the estimator and $(1-\alpha)$ is the confidence level. Thus it is $100(1-\alpha)\%$ confident that the true mean value μ lies in the interval $(\bar{X}(n) - \delta, \bar{X}(n) + \delta)$. It is well known that if observations $x_1, x_2, \ldots, x_n$ can be regarded as realizations of independent and identically distributed (i.i.d.) random variables, then

$$\delta = t_{n-1,1-\alpha/2}\sigma[\bar{X}(n)] \tag{7.4}$$

where

$$\sigma[\bar{X}(n)] = \sum_{i=1}^{i=n} \frac{(x_i - \bar{X}(n))^2}{n(n-1)} \tag{7.5}$$

and $t_{n-1,1-\alpha/2}$ is the $100(1-\alpha/2)\%$ point of a t-distribution with $n-1$ degrees of freedom; that is, $P(t \geq t_{n-1,1-\alpha/2}) = \alpha/2$ and $t \sim t_{n-1}$.

In most cases, a confidence level of 95% ($\alpha = 0.05$) is desired. Once the δ is calculated based on a given α, the relative precison ϵ of the simulation results is obtained

[1] A steady-state simulation is a simulation whose objective is to study long-run, or steady-state, behavior of a nonterminating system [22]

as follows [56].

$$\epsilon = \frac{\delta}{\bar{X}(n)} \tag{7.6}$$

Usually, the relative precision of the estimator is chosen to be 5% of the estimated values.

As a result, confidence level analysis can be conducted for simulations. For instance, for the simulations carried out in Chapter 4, The relative precision of the network throughputs with different AQM schemes at confidence level of 95% is shown in Table 7.1.

Table 7.1: Relative Precision of the network throughputs

Traffic Load	Drop Tail	RED	ARED	PI	REM	BLUE
50	0.153	0.160	0.155	0.153	0.156	0.159
80	0.143	0.135	0.137	0.140	0.135	0.115
90	0.124	0.121	0.123	0.123	0.121	0.108
100	0.068	0.075	0.073	0.072	0.098	0.062
110	0.002	0.005	0.003	0.004	0.003	0.003
120	0.003	0.001	0.003	0.002	0.003	0.001

Some of the simulations have not reached the desired accuracy. The relative precision with higher traffic loads is quite adequate compared with those with lower traffic loads. To achieve the desired confidence level with satisfactory precision, methods are needed to instruct how to carry out simulations. [80] provides a solution called sequential simulation, which adaptively decides about continuing a simulation experiment until the required accuracy of results is reached.

7.5 Future Work

This research has included many interesting research topics and we have achieved significant outcomes, nevertheless, it still leaves a lot for future research. We sincerely believe the following two research directions deserve future exploration.

- Firstly, coupling neural networks (NNs) and FL, namely Neuro-Fuzzy, obtains better control. NNs is derived from the attempt to make use of the known or expected organizing principles of the human brain, and belongs to the same

family of artificial intelligent technologies as FL. The strengths of NNs lie in their learning capabilities and their distributed structure that allows for highly parallel software or hardware implementations [75]. Therefore, it is expected that Neuro-Fuzzy controller will be able to overcome man-made mistakes and conduct autoconfiguration not only of control parameters, but also of fuzzy rules based on network conditions. Implementation of on-line tuning is the next research step towards the design of such Neuro-Fuzzy controllers, particularly for Internet traffic.

- On the other hand, FL or Neuro-Fuzzy can be deployed to carry out robust control in some other network research areas. Diffserv, a new Internet architecture, is becoming of interest to both ISPs and researchers. Before its universal adoption by ISPs, more studies need to be done to achieve end-to-end QoS. One straightforward deployment of current work to Diffserv is to doubly use FLAQM to do differential dropping of IN and OUT packets respectively, for the Assured Forwarding (AF) service model.

Bibliography

[1] The Network Simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[2] R. G. Addie, T. M. Neame, and M. Zukerman. Performance Evaluation of a Queue Fed by a Poisson Pareto Burst Process. *Computer Networks*, 40:377–397, October 2002.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *RFC2581*, April 1999.

[4] S. Athuraliya and S. Low. Optimization Flow Control – II: Implementation. Technical report, University of Melbourne, Australia, 2000. `http://citeseer.nj.nec.com/article/athuraliya00optimization.html`.

[5] T. Bonald, M. May, and J. Bolot. Analytic Evaluation of RED Performance. In *Proceedings of IEEE INFOCOM (3)*, pages 1415–1424, 2000.

[6] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. *RFC1633*, 1994.

[7] X. Chen and J. Heidemann. Preferential Treatment for Short Flows to Reduce Web Latency. *Computer Networks*, 41(6):779–794, April 2003.

[8] K. Claffy, G. Miller, and K. Thompson. the Nature of the Beast: Recent Traffic Measurements from An Internet Backbone. In *Proceedings of Internet Society INET'98*, Geneva, Switzerland, 1998.

[9] D. D. Clark and W. Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, 1998.

[10] The ATM Forum Technical Committee. Traffic Management Specification Version 4.1. *AF-TM-0121.000*, March, 1999.

[11] B. Davie and Y. Rekhter. *MPLS: Technology and Applications*. Morgan Kaufmann, 2000.

[12] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *Proceedings of SIGCOMM*, pages 109–120, 1999.

[13] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services, Part II: Loss Rate Differentiation and Packet Dropping. In *Proceedings of IEEE/IFIP International Workshop on Quality of Service (IWQoS)*, pages 52–61, June, 2000.

[14] D. Driankov, H. Hellendoorn, and M. Reinfrank. *An Introduction to Fuzzy Control (Second Edition)*. Springer, 1996.

[15] A. Kandel et al. ATM Traffic Management and Congestion Control using Fuzzy Logic. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 29(3):474–480, 1999.

[16] B. Braden et al. Recommendations on Queue Management and Congestion Avoidance in the Internet. *RFC2309*, April 1998.

[17] C. Chrysostomou et al. Fuzzy Logic Congestion Control in TCP/IP Best Effort Networks. In *Proceedings of the Australian Telecommunications, Networks and Applications Conference (ATNAC)*, Melbourne, Australia, December 2003.

[18] C. V. Hollot et al. A Control Theoretic Analysis of RED. In *Proceedings of IEEE INFOCOM*, 2001.

[19] C. V. Hollot et al. On Designing Improved Controllers for AQM Routers Supporting TCP Flows. In *Proceedings of IEEE INFOCOM*, pages 1726–1734, 2001.

[20] F. Le Faucheur (Editor) et al. Multiprotocol Label Switching Architecture (MPLS) Support of Differentiated Services. *RFC3270*, 2002.

[21] G. Gopalakrishnan et al. Robust Router Overload Control Using Acceptance and Departure Rate Measures. In *Proceedings of the 18th International Teletraffic Congress (ITC)*, Berlin, Germany, September, 2003.

[22] J. Banks et al. Discrete-Event System Simulation (3rd Edition). In *Prentice-Hall,*, 2001.

[23] J. Sun et al. Adaptive Drop-Tail: A Simple and Efficient Active Queue Management Algorithm for Internet Flow Control. In *Proceedings of the 18th International Teletraffic Congress (ITC)*, Berlin, Germany, September, 2003.

[24] K. Pawlikowski et al. On Credibility of Simulation Studies Of Telecommunication Networks. *IEEE Communications Magazine*, (40(1)):132–139, Jan. 2002.

[25] M. May et al. Reasons Not to Deploy RED. In *Proceedings of 7th. International Workshop on Quality of Service (IWQoS)*, pages 260–262, London, June 1999.

[26] R. Braden Ed. et al. Resource ReserVation Protocol (RSVP) – Version 1 Functional Specification. *RFC2205*, 1997.

[27] S. Athuraliya et al. REM: Active Queue Management. *IEEE Network*, 15(3):48 – 53, May/June 2001.

[28] S. Bajaj et al. Improving Simulation for Network Research. Technical Report 99-702b, University of Southern California, March 1999. `http://www.isi.edu/~johnh/PAPERS/Bajaj99a.html`.

[29] S. Blake et al. An Architecture for Differentiated Services. *RFC2475*, 1998.

[30] S. Kalyanaraman et al. The ERICA Switch Algorithm for ABR Traffic Management in ATM Networks. *IEEE/ACM Transactions on Networking*, 8(1):87–98, 2000.

[31] T. Ziegler et al. Stability of RED with two-way TCP Traffic. In *Proceedings of IEEE ICCCN*, Las Vegas, Oct 2000.

[32] V. Catania et al. A Comparative Analysis of Fuzzy versus Conventional Policing Mechanisms for ATM Networks. *IEEE/ACM Transactions on Networking*, 4(3):449–459, 1996.

[33] W. E. Leland et al. On the Self-similar Nature of Ethernet Traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, 1994.

[34] W. Feng et al. A Self-Configuring RED Gateway. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1320–1328, 1999.

[35] Y. Bernet et al. A Framework for Integrated Services Operation over Diffserv Networks. *RFC2998*, November 2000.

[36] Y. Bernet et al. A Framework for Integrated Services Operation over Diffserv Networks. *RFC2998*, November 2000.

[37] Y. Zhang et al. On the Characteristics and Origins of Internet Flow Rates. In *Proceedings of SIGCOMM*, pages 309–322, Pittsburgh, Pennsylvania, USA, August 2002.

[38] Z. Li et al. Improving the Adaptability of AQM Algorithms to Traffic Load Using Fuzzy Logic. In *Proceedings of the Australian Telecommunications, Networks and Applications Conference (ATNAC)*, Melbourne, Australia, December 2003.

[39] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[40] W. Fang. Differentiated Services: Architecture, Mechanisms and an Evaluation. *PhD Dissertation, Princeton University*, 2000.

[41] W. Feng. BLUE: A New Class of Active Queue Management Algorithms. Technical Report CSE-TR-387-99, 15, 1999. `http://citeseer.nj.nec.com/article/feng99blue.html`.

[42] W. Feng, P. Tinnakornsrisuphap, and I. Philp. On the Burstiness of the TCP Congestion-Control Mechanism in a Distributed Computing System. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 110 – 117. IEEE Computer Society, 2000.

[43] V. Firiou, X. Zhang, and Y. Guo. Best Effort Differentiated Services: Trade-off Service Differentiation for Elastic Applications. In *Proceedings of IEEE ICT*, Jun 2001.

[44] V. Firoiu and M. Borden. A Study of Active Queue Management for Congestion Control. In *Proceedings of IEEE INFOCOM (3)*, pages 1435–1444, 2000.

[45] S. Floyd. Recommendation on using the gentle variant of RED. Technical report. `http://www.icir.org/floyd/red/gentle.html`.

[46] S. Floyd. RED: Discussions of Setting Parameters. Technical report. `http://www.icir.org/floyd/REDparameters.txt`.

[47] S. Floyd. Setting Parameters:. Technical report. `http://www.icir.org/floyd/red.html#notes`.

[48] S. Floyd. Connections with Multiple Congested Gateways in Packet-Switched Networks PartI: One-way Traffic. *Computer Communication Review*, 21(5):30–47, October, 1991.

[49] S. Floyd. TCP and Explicit Congestion Notification. *ACM Computer Communication Review*, 24(5):10–23, October 1994.

[50] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An Algorithm for Increasing the Robustness of RED. Technical report, 2001. `http://citeseer.nj.nec.com/floyd01adaptive.html`.

[51] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[52] S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August, 2001.

[53] D. Grossman. New Terminology and Clarifications for Diffserv. *RFC3260*, April 2002.

[54] L. Guo and I. Matta. The War Between Mice and Elephants. Technical Report 2001-005, 2001. `http://citeseer.nj.nec.com/guo01war.html`.

[55] M. Hassan and M. Atiquzzaman. *Performance of TCP/IP Over ATM Networks*. Artech House Publishers, 2000.

[56] M. Hassan and R. Jain. High Performance TCP/IP Networking: Concepts, Issues, and Solutions. In *Prentice-Hall,*, 2003.

[57] N. Hohn, D. Veitch, and P. Abry. Cluster Processes, a Natural Language for Network Traffic. *IEEE Transactions on Signal Processing, Special Issue on Signal Processing in Networking*, 51(8):2222–2249, 2003.

[58] R. Q. Hu and D. W. Petr. A Predictive Self-tuning Fuzzy-logic Feedback Rate Controller. *IEEE/ACM Transactions on Networking*, 8(6):697–709, 2000.

[59] P. Hurley, J. Y. L. Boudec, and P. Thiran. The Asymmetric Best-Effort Service. In *Proceedings of IEEE GLOBECOM*, Rio de Janeiro, Brazil, December 1999.

[60] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM*, pages 314–329, 1988.

[61] V. Jacobson, K. Nichols, and K. Poduri. RED in a different Light. Technical report, Cisco Systems, 1999.

[62] J. R. Jang. ANFIS: Adaptive-Network-Based Fuzzy Inference System. *IEEE Transactions on Systems, Man, and Cybernetics*, 23:665–684, 1993.

[63] S. Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network.* ADDISON-WESLEY, 1997.

[64] S. Kunniyur and R. Srikant. A Decentralized Adaptive ECN Marking Algorithm. In *Proceedings of GLOBECOM*, San Francisco, Nov. 2000.

[65] S. Kunniyur and R. Srikant. End-to-End Congestion Control Schemes: Utility Functions, Random Losses and ECN Marks. In *Proceedings of IEEE INFOCOM (3)*, pages 1323–1332, 2000.

[66] S. Kunniyur and R. Srikant. Analysis and Design of An Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 123–134. ACM Press, 2001.

[67] S. Kunniyur and R. Srikant. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 123–134. ACM Press, 2001.

[68] C. C. Lee. Fuzzy Logic in Control Systems: Fuzzy Logic Controller – Part I. *IEEE Transactions on Systems, Man and Cybernetics*, 20(2):404–418, 1990.

[69] C. C. Lee. Fuzzy Logic in Control Systems: Fuzzy Logic Controller – Part II. *IEEE Transactions on Systems, Man and Cybernetics*, 20(2):419–435, 1990.

[70] Z. Li and Z. Zhang. An Application of Fuzzy Logic to Usage Parameter control in ATM Networks. In *Proceedings of the 1st International Conference on Fuzzy Systems and Knowledge Discovery: Computational Intelligence for the E-Age*, 2002.

[71] H. H. Lim and B. Qiu. Performance Improvement of TCP using Fuzzy Logic Prediction. In *Proceedings of International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, number 20-21, pages 152–156, Nashville, USA, November 2001.

[72] D. Lin and R. Morris. Dynamics of Random Early Detection. In *Proceedings of SIGCOMM*, pages 127–137, Cannes, France, september 1997.

[73] S. H. Low and D. E. Lapsley. Optimization Flow Control — I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874, 1999.

[74] D. McDysan. *QoS & Traffic Management in IP & ATM Networks*. McGraw-Hill, 2000.

[75] D. Nauck, F. Klawonn, and R. Kruse. *Foundations of Neuro-Fuzzy Systems*. Chichester: Wiley, 1997.

[76] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. *RFC2638*, July 1999.

[77] T. J. Ott, T. V. Lakshman, and L. H. Wong. SRED: Stabilized RED. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1346–1355, 1999.

[78] K. Park, G. Kim, and M. Crovella. On the Effect of Traffic Self-similarity on Network Performance. In *Proceedings of SPIE International Conference on Performance and Control of Network Systems*, pages 296–310, 1997.

[79] K. M. Passino and S. Yurkovich. *Fuzzy Control*. ADDISON-WESLEY, 1998.

[80] K. Pawlikowski. Steady State Simulation of Queueing Processes: A Survey of Problems and Solutions. *ACM Computing Surveys*, (22(2)):123–170, June 1990.

[81] V. Paxson and S. Floyd. Wide-area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[82] V. Paxson and S. Floyd. Wide Area Traffic: the Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[83] K. Pentikousis and H. Badr. An Evaluation of TCP with Explicit Congestion Notification. In *Annals of Telecommunications*, Mid-2003.

[84] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. *RFC2481*, January 1999.

[85] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP. *RFC2481*, January 1999.

[86] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC3168*, September 2001.

[87] P. Ranjan, E. H. Abed, and R. J. La. Nonlinear Instabilities in TCP-RED. In *Proceedings of IEEE INFOCOM*, June 2002.

[88] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. *RFC3031*, 2001.

[89] V. Rosolen, O. Bonaventure, and G. Leduc. A RED Discard Strategy for ATM Networks and Its Performance Evaluation with TCP/IP Traffic. *ACM SIGCOMM Computer Communication Review*, 29(3):23–43, 1999.

[90] W. Stallings. *High-speed Networks: TCP/IP and ATM Design Principles*. Prentice Hall, 1998.

[91] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[92] S. Thulasidasan and W. Feng. Using Steady-State TCP Behavior for Proactive Queue Management. Technical report. `http://citeseer.nj.nec.com/543165.html`.

[93] W. Willinger, V. Paxson, and M. S. Taqqu. Self-similarity and Heavy Tails: Structural Modeling of Network Traffic. *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, 1998.

[94] B. Wydrowski and M. Zukerman. GREEN: An Active Queue Management Algorithm for a Self Managed Internet. Technical report, 2002. `http://citeseer.nj.nec.com/wydrowski02green.html`.

[95] P. Yan, Y. Gao, and H. Ozbay. Variable Structure Control in Active Queue Management for TCP with ECN. In *Proceedings of ISCC, (accepted by IEEE Transactions on Control Systems Technology)*, Kemer-Antalya, Turkey, June 2003.

[96] P. Yan, Y. Gao, and H. Ozbay. Variable Structure Control in Active Queue Management for TCP with ECN. In *Proceedings of ISCC*, Kemer-Antalya, Turkey, June 2003. Accepted by IEEE Transactions on Control Systems Technology.

[97] L. Zhang and D. Clark. Oscillating Behavior of Network Traffic: a Case Study Simulation. *Internetworking: Research and Experience*, 1(2):101 – 112, 1990.

[98] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: the Effects of Two-way Traffic. In *Proceedings of the conference on Communications architecture & protocols*, pages 133–147. ACM Press, 1991.

[99] T. Ziegler, S. Fdida, and C. Brandauer. Stability Criteria of RED with TCP Traffic. In *Proceedings of IFIP ATM&IP Working Conference, Budapest*, June 2001.

Appendix A

Computer Programs for FLAQM(I) and FLAQM(II)

A.1 The *.h* and *.cc* Source Code for Implementing FLAQM(I) and FLAQM(II) Builtin NS2 (2.26)

A.1.1 FLAQM.h

```
 /*                                                                    1
  * Fuzzy logic.AQM: Zhi Li (USQ)
  */

#ifndef ns_fl_h
#define ns_fl_h

#include "queue.h"
#include <stdlib.h>
#include "agent.h"
#include "template.h"

class LinkDelay;

class FLQueue;

/*
 * fl parameters, supplied by user
 */
struct MYflp {
        /*                                                            21
         * User supplied.
         */
        int whichFLAQM;
        // parameters for FLAQM (FLAQM(I))
        double FLAQM_lf_p1;
        double FLAQM_lf_p2;
        double FLAQM_lf_p3;
        double FLAQM_lf_p4;
        double FLAQM_lf_p5;

        double FLAQM_d_lf_N_p1;
        double FLAQM_d_lf_N_p2;
        double FLAQM_d_lf_N_p3;
        double FLAQM_d_lf_N_p4;

        double FLAQM_d_lf_P_p1;
        double FLAQM_d_lf_P_p2;
        double FLAQM_d_lf_P_p3;
        double FLAQM_d_lf_P_p4;
        double FLAQM_d_lf_P_p5;                                       41
```

```
        /* multiply decrease parameters */
        double FLAQM_MD_p1;
        double FLAQM_MD_p2;
        double FLAQM_MD_p3;
        double FLAQM_MD_p4;
        double FLAQM_MD_p5;
        double FLAQM_MD_p6;

        /* additionaly increase parameters */
        double FLAQM_AI_p1;
        double FLAQM_AI_p2;
        double FLAQM_AI_p3;
        double FLAQM_AI_p4;
        double FLAQM_AI_p5;
        double FLAQM_AI_p6;

        // parameters for IFLAQM (FLAQM(II))
        double IFLAQM_lf_p1;                                              61
        double IFLAQM_lf_p2;
        double IFLAQM_lf_p3;
        double IFLAQM_lf_p4;

        double IFLAQM_d_lf_N_p1;

        double IFLAQM_d_lf_P_p1;
        double IFLAQM_d_lf_P_p2;


        /* multiply decrease parameters */
        double IFLAQM_MD_p1;
        double IFLAQM_MD_p2;
        double IFLAQM_MD_p3;
        double IFLAQM_MD_p4;
        double IFLAQM_MD_p5;
        double IFLAQM_MD_p6;
        double IFLAQM_MD_p7;

        /* additionaly increase parameters */                             81
        double IFLAQM_AI_p1;
        double IFLAQM_AI_p2;
        double IFLAQM_AI_p3;
        double IFLAQM_AI_p4;
        double IFLAQM_AI_p5;
        double IFLAQM_AI_p6;
        double IFLAQM_AI_p7;


        double p_delta; // delta: 1 <=load_factor <= 1+delta
        int  aggressive; //qlen<Q0,fraction>1; otherwise fraction = 1
        int p_pktsize;
        double p_updtime;
        double p_bo; // time for example 5ms
        double p_QDLF;
        double p_a;
        double p_b;
        int p_stepnumber;
        double p_inw;
        double q_w;                                                       101
        int wait;       /* true for waiting between dropped packets */
        int p_droplikered;  // uniformly drop incoming packets
        int p_deterministic; // deterministic drop incoming packets
        int input_greenlike; // flag for GREEN like cal for inputrate
        double K; // cal input_rate (GREEN like)

        /*
         * Computed as a function of user supplied parameters.
        */
        double ptc;

};

/*
 * fl variables, maintained by fl
```

```
 */
struct MYflv {
        TracedDouble v_prob;     /* prob. of packet marking. */
        TracedDouble v_z;        /* used in computing the load factor */
        TracedDouble v_deltaz;   /* used in computing the load                121
                                    factor change */
        double v_z_old;          /* for calculating v_deltaz */
        double v_count;
        double v_ave;
        TracedDouble v_qave;
        //double v_qave_old;
        int v_qlen_old;          /* leftover queue at the end of
                                    last period */
        int count;               /* # of packets since last drop */
        int count_bytes;         /* # of bytes since last drop */
        double deterministic_r; /*for deterministic dropping */
        double est; // estimation input rate as GREEN

        MYflv() : v_prob(0.0), v_z(0.0), v_deltaz(0.0), v_z_old(0.0),
                  v_ave(0.0), v_qave(0.0), v_qlen_old(0), count(0),
                  count_bytes(0), deterministic_r(0.0), est(0.0) { }
};

class FLTimer : public TimerHandler {
        public:                                                            141
                FLTimer (FLQueue *a) : TimerHandler() { a_ = a; }
        protected:
                virtual void expire (Event *e);
                FLQueue *a_;
};

class FLQueue : public Queue {
 public:
        FLQueue();
        void set_update_timer();
        void timeout();

 protected:
        int command(int argc, const char*const* argv);
        void initialize_params();
        double estimator(int nqueued, int m, double ave, double q_w);
        double modify_p(double p, int count, int count_bytes, int bytes,
                        int mean_pktsize, int wait, int size);
        double arrival_time_old; // last packet arrive time
                                 // in GREEN for inputrate                  161
        void enque(Packet* pkt);
        Packet* deque();
        void reset();
        void FLAQM_run_updaterule();
        void IFLAQM_run_updaterule();

        LinkDelay* link_;        /* outgoing link */
        PacketQueue *q_;         /* underlying (usually) FIFO queue */

        Tcl_Channel tchan_;      /* Place to write trace records */
        TracedInt curq_;         /* current qlen seen by arrivals */
        void trace(TracedVar*); /* routine to write trace records */

        FLTimer fl_timer_;

        /*
         * Static state.
         */
        double pmark_;    //number of packets being marked
        MYflp flp_;      /* early-drop params */                           181

        /*
         * Dynamic state.
         */
        MYflv flv_;              /* early-drop variables */
        int idle_;
```

```
        double idletime_;
        int markpkts_ ;
        int qib_;

        void print_flp();        // for debugging
        void print_flv();        // for debugging


        double FLAQM_fl_MD_AQM(double lf, double delta_lf,
                               int step_number);
        double FLAQM_fl_AI_AQM(double lf, double delta_lf,
                               int step_number);
        double IFLAQM_fl_MD_AQM(double lf, double delta_lf,
                               int step_number);                    201
        double IFLAQM_fl_AI_AQM(double lf, double delta_lf,
                               int step_number);
        double triangle(double x, double left, double medium,
                        double right) {
              double y;
              if(x < left || x > right)
                   y = 0;
              else if( x >= left && x <= medium) {
                   if (medium > left)
                        y = (1 / (medium-left)) * (x-left);
                   else if (medium == left)
                        y = 1;
                   else
                        fprintf(stderr, "medium should be greater
                                         than or equal to left\n");
              }
              else {
                   if (medium < right)
                        y = (1 / (medium-right)) * (x - right);
                   else if (medium == right)                        221
                        y = 1;
                   else
                        fprintf(stderr, "medium should be less
                                         than or equal to right\n");
              }
              return y;
        }
        double  trapezoid(double x, double left, double mleft,
                        double mright, double right) {
             double y;
             if(x < left || x > right)
                  y = 0;
             else if( x >= left && x <= mleft) {
                  if (mleft > left)
                       y = (1 / (mleft-left)) * (x-left);
                  else if (mleft == left)
                       y = 1;
                  else
                       fprintf(stderr, "mleft should be greater than
                                        or equal to left\n");       241
             }
             else if(x > mleft && x <= mright)
                 y = 1;
             else {
                if (mright < right)
                       y = (1 / (mright-right)) * (x - right);
                else if (mright == right)
                       y = 1;
                else
                       fprintf(stderr, "mright should be less
                                        than or equal to right\n");
             }
             return y;
        }

// this is for FLAQM (FLAQM(I))
        double FLAQM_H1(double x) {
```

```
        double y;
        y = trapezoid(x, 0.8, 0.8, flp_.FLAQM_lf_p1,
                      flp_.FLAQM_lf_p2);
        return y;
}
double FLAQM_H2(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_lf_p1, flp_.FLAQM_lf_p2,
                 flp_.FLAQM_lf_p3);
    return y;
}
double FLAQM_H3(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_lf_p2, flp_.FLAQM_lf_p3,
                 flp_.FLAQM_lf_p4);
    return y;
}
double FLAQM_H4(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_lf_p3, flp_.FLAQM_lf_p4,
                 flp_.FLAQM_lf_p5);
    return y;
}
double FLAQM_H5(double x) {
    double y;
    y = trapezoid(x, flp_.FLAQM_lf_p4, flp_.FLAQM_lf_p5,
                  200, 200);
    return y;
}


double FLAQM_N5(double x) {
    double y;
    y = trapezoid(x, -200, -200, flp_.FLAQM_d_lf_N_p1,
                  flp_.FLAQM_d_lf_N_p2);
    return y;
}
double FLAQM_N4(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_N_p1, flp_.FLAQM_d_lf_N_p2,
                 flp_.FLAQM_d_lf_N_p3);
    return y;
}
double FLAQM_N3(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_N_p2, flp_.FLAQM_d_lf_N_p3,
                 flp_.FLAQM_d_lf_N_p4);
    return y;
}
double FLAQM_N2(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_N_p3, flp_.FLAQM_d_lf_N_p4, 0);
    return y;
}
double FLAQM_N1(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_N_p4, 0, 0);
    return y;
}
double FLAQM_P0(double x) {
    double y;
    y = triangle(x, 0, 0, flp_.FLAQM_d_lf_P_p1);
    return y;
}
double FLAQM_P1(double x) {
    double y;
    y = triangle(x, 0, flp_.FLAQM_d_lf_P_p1,
                 flp_.FLAQM_d_lf_P_p2);
    return y;
}
double FLAQM_P2(double x) {
```

```
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_P_p1, flp_.FLAQM_d_lf_P_p2,
                 flp_.FLAQM_d_lf_P_p3);
    return y;
}
double FLAQM_P3(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_P_p2, flp_.FLAQM_d_lf_P_p3,
                 flp_.FLAQM_d_lf_P_p4);
    return y;
}
double FLAQM_P4(double x) {                                          341
    double y;
    y = triangle(x, flp_.FLAQM_d_lf_P_p3, flp_.FLAQM_d_lf_P_p4,
                 flp_.FLAQM_d_lf_P_p5);
    return y;
}
double FLAQM_P5(double x) {
    double y;
    y = trapezoid(x, flp_.FLAQM_d_lf_P_p4, flp_.FLAQM_d_lf_P_p5,
                  200, 200);
    return y;
}

double FLAQM_MD_Z(double x) {
    double y;
    y = triangle(x, -flp_.FLAQM_MD_p1, 0, flp_.FLAQM_MD_p1);
    return y;
}
double FLAQM_MD_P1(double x) {
    double y;
    y = triangle(x, 0, flp_.FLAQM_MD_p1, flp_.FLAQM_MD_p2);          361
    return y;
}
double FLAQM_MD_P2(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_MD_p1, flp_.FLAQM_MD_p2,
                 flp_.FLAQM_MD_p3);
    return y;
}
double FLAQM_MD_P3(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_MD_p2, flp_.FLAQM_MD_p3,
                 flp_.FLAQM_MD_p4);
    return y;
}
double FLAQM_MD_P4(double x) {
    double y;
    y = triangle(x, flp_.FLAQM_MD_p3, flp_.FLAQM_MD_p4,
                 flp_.FLAQM_MD_p5);
    return y;
}                                                                    381
double FLAQM_MD_P5(double x) {
    double y;
    y = trapezoid(x, flp_.FLAQM_MD_p4, flp_.FLAQM_MD_p5,
                  flp_.FLAQM_MD_p6, flp_.FLAQM_MD_p6);
    return y;
}

double FLAQM_AI_Z(double x)       {
    double y;
    y = triangle(x, -flp_.FLAQM_AI_p1, 0, flp_.FLAQM_AI_p1);
    return y;
}
double FLAQM_AI_P1(double x)      {
    double y;
    y = triangle(x, 0, flp_.FLAQM_AI_p1, flp_.FLAQM_AI_p2);
    return y;
}
double FLAQM_AI_P2(double x)      {
    double y;
```

```
        y = triangle(x, flp_.FLAQM_AI_p1, flp_.FLAQM_AI_p2,
                     flp_.FLAQM_AI_p3);
        return y;
    }
    double FLAQM_AI_P3(double x)      {
        double y;
        y = triangle(x, flp_.FLAQM_AI_p2, flp_.FLAQM_AI_p3,
                     flp_.FLAQM_AI_p4);
        return y;
    }
    double FLAQM_AI_P4(double x)      {
        double y;
        y = triangle(x, flp_.FLAQM_AI_p3, flp_.FLAQM_AI_p4,
                     flp_.FLAQM_AI_p5);
        return y;
    }
    double FLAQM_AI_P5(double x)      {
        double y;
        y = trapezoid(x, flp_.FLAQM_AI_p4, flp_.FLAQM_AI_p5,
                      flp_.FLAQM_AI_p6, flp_.FLAQM_AI_p6);
        return y;
    }


// this is for IFLAQM (FLAQM(II))
    double IFLAQM_H1(double x) {
        double y;
        y = triangle(x, 0, 0, flp_.IFLAQM_lf_p1);
        return y;
    }
    double IFLAQM_H2(double x) {
        double y;
        y = triangle(x, 0, flp_.IFLAQM_lf_p1, flp_.IFLAQM_lf_p2);
        return y;
    }
    double IFLAQM_H3(double x) {
        double y;
        y = triangle(x, flp_.IFLAQM_lf_p1, flp_.IFLAQM_lf_p2,
                     flp_.IFLAQM_lf_p3);
        return y;
    }
    double IFLAQM_H4(double x) {
        double y;
        y = triangle(x, flp_.IFLAQM_lf_p2, flp_.IFLAQM_lf_p3,
                     1+flp_.p_delta);
        return y;
    }
    double IFLAQM_H5(double x) {
        double y;
        y = triangle(x, flp_.IFLAQM_lf_p3, 1+flp_.p_delta,
                     1+flp_.p_delta);
        return y;
    }


    double IFLAQM_N1(double x) {
        double y;
        y = trapezoid(x, -2, -2, -(1+flp_.p_delta),
                      flp_.IFLAQM_d_lf_N_p1);
        return y;
    }
    double IFLAQM_N2(double x) {
        double y;
        y = triangle(x, -(1+flp_.p_delta), flp_.IFLAQM_d_lf_N_p1, 0);
        return y;
    }
    double IFLAQM_N3(double x) {
        double y;
        y = triangle(x, flp_.IFLAQM_d_lf_N_p1, 0, 0);
        return y;
```

```
}

double IFLAQM_P1(double x) {
    double y;
    y = triangle(x, 0, 0, flp_.IFLAQM_d_lf_P_p1);
    return y;
}
double IFLAQM_P2(double x) {
    double y;                                                    481
    y = triangle(x, 0, flp_.IFLAQM_d_lf_P_p1,
                 flp_.IFLAQM_d_lf_P_p2);
    return y;
}
double IFLAQM_P3(double x) {
    double y;
    y = triangle(x, flp_.IFLAQM_d_lf_P_p1,
            flp_.IFLAQM_d_lf_P_p2, 1+flp_.p_delta);
    return y;
}
double IFLAQM_P4(double x) {
    double y;
    y = triangle(x, flp_.IFLAQM_d_lf_P_p2, 1+flp_.p_delta,
                 1+flp_.p_delta);
    return y;
}

double IFLAQM_MD_P1(double x) {
    double y;                                                    501
    y = triangle(x, -flp_.IFLAQM_MD_p1, 0, flp_.IFLAQM_MD_p1);
    return y;
}
double IFLAQM_MD_P2(double x) {
    double y;
    y = triangle(x, 0, flp_.IFLAQM_MD_p1, flp_.IFLAQM_MD_p2);
    return y;
}
double IFLAQM_MD_P3(double x) {
    double y;
    y = triangle(x, flp_.IFLAQM_MD_p1, flp_.IFLAQM_MD_p2,
                 flp_.IFLAQM_MD_p3);
    return y;
}
double IFLAQM_MD_P4(double x) {
    double y;
    y = triangle(x, flp_.IFLAQM_MD_p2, flp_.IFLAQM_MD_p3,
                 flp_.IFLAQM_MD_p4);
    return y;
}                                                                521
double IFLAQM_MD_P5(double x) {
    double y;
    y = triangle(x, flp_.IFLAQM_MD_p3, flp_.IFLAQM_MD_p4,
                 flp_.IFLAQM_MD_p5);
    return y;
}
double IFLAQM_MD_P6(double x) {
    double y;
    y = triangle(x, flp_.IFLAQM_MD_p4, flp_.IFLAQM_MD_p5,
                 flp_.IFLAQM_MD_p6);
    return y;
}
double IFLAQM_MD_P7(double x) {
    double y;
    y = trapezoid(x, flp_.IFLAQM_MD_p5, flp_.IFLAQM_MD_p6,
                 flp_.IFLAQM_MD_p7, flp_.IFLAQM_MD_p7);
    return y;
}

                                                                 541
double IFLAQM_AI_P1(double x)     {
```

```
    double y;
    y = triangle(x, -flp_.IFLAQM_AI_p1, 0, flp_.IFLAQM_AI_p1);
    return y;
}
double IFLAQM_AI_P2(double x)    {
    double y;
    y = triangle(x, 0, flp_.IFLAQM_AI_p1, flp_.IFLAQM_AI_p2);
    return y;
}
double IFLAQM_AI_P3(double x)    {
    double y;
    y = triangle(x, flp_.IFLAQM_AI_p1, flp_.IFLAQM_AI_p2,
                 flp_.IFLAQM_AI_p3);
    return y;
}
double IFLAQM_AI_P4(double x)    {
    double y;
    y = triangle(x, flp_.IFLAQM_AI_p2, flp_.IFLAQM_AI_p3,
                  flp_.IFLAQM_AI_p4);
    return y;
}
double IFLAQM_AI_P5(double x)    {
    double y;
    y = triangle(x, flp_.IFLAQM_AI_p3, flp_.IFLAQM_AI_p4,
                  flp_.IFLAQM_AI_p5);
    return y;
}
double IFLAQM_AI_P6(double x)    {
    double y;
    y = triangle(x, flp_.IFLAQM_AI_p4, flp_.IFLAQM_AI_p5,
                  flp_.IFLAQM_AI_p6);
    return y;
}
double IFLAQM_AI_P7(double x)    {
    double y;
    y = trapezoid(x, flp_.IFLAQM_AI_p5, flp_.IFLAQM_AI_p6,
                  flp_.IFLAQM_AI_p7, flp_.IFLAQM_AI_p7);
    return y;
}

double max2(double x, double y)  {
    if (x >= y) {
        return x;
    }
    return y;
}
double min2(double x, double y)  {
    if (x <= y) {
        return x;
    }
    return y;
}
double max3(double x, double y, double z) {
    double y1 = max2(x, y);
    double y2 = max2(y1, z);
    return y2;
}
double min3(double x, double y, double z) {
    double y1 = min2(x, y);
    double y2 = min2(y1, z);
    return y2;
}
double maxn(double *x, int size){
    double y;
    y = x[0];
    for (int i = 1; i < size; i++) {
        y = max2(y, x[i]);
    }
    return y;
}
double minn(double *x, int size) {
```

```
            double y;
            y = x[0];
            for (int i = 1; i < size; i++) {
                y = min2(y, x[i]);
            }
            return y;
        }
        double integral( double *arr, double step_size, int step_number)
        {
            double sum=0;
            //Simpson algorithm samples the integrand in several
            //point which significantly improves precision.
            for(int i=0; i<step_number; i=i+2)
                // divide the area under f(x) into step_number
                // rectangles and sum their areas
                sum = sum + (arr[i]+4*arr[i+1]+ arr[i+2]) *step_size/3;
            return sum;
        }
};
#endif
```

A.1.2 FLAQM.cc

```
/*
 * Fuzzy Logic.AQM: Zhi Li
 */
#include <math.h>
#include <sys/types.h>
#include "config.h"
#include "template.h"
#include "random.h"
#include "flags.h"
#include "delay.h"
#include "fl.h"
#include <iostream>

static class FLClass : public TclClass {
public:
        FLClass() : TclClass("Queue/FL") {}
        TclObject* create(int, const char*const*) {
                return (new FLQueue);
        }
} class_fl;


void FLQueue :: set_update_timer()
{
        fl_timer_.resched(flp_.p_updtime);
}

void FLQueue :: timeout()
{
        //do drop/mark prob update
        if (flp_.whichFLAQM == 0)
                FLAQM_run_updaterule();
        else if (flp_.whichFLAQM == 1)
                IFLAQM_run_updaterule();
        set_update_timer();
}


void FLTimer :: expire (Event *e) {
         a_->timeout();
}
```

```
                                                                        46
FLQueue::FLQueue() : link_(NULL), tchan_(0), fl_timer_(this), idle_(1)
{
  bind("whichFLAQM", &flp_.whichFLAQM);
  //0--FLAQM (FLAQM(I)) 1--IFLAQM (FLAQM(II))
  // FLAQM parameters
  bind("FLAQM_lf_p1_", &flp_.FLAQM_lf_p1);
  bind("FLAQM_lf_p2_", &flp_.FLAQM_lf_p2);
  bind("FLAQM_lf_p3_", &flp_.FLAQM_lf_p3);
  bind("FLAQM_lf_p4_", &flp_.FLAQM_lf_p4);
  bind("FLAQM_lf_p5_", &flp_.FLAQM_lf_p5);


  bind("FLAQM_d_lf_N_p1_", &flp_.FLAQM_d_lf_N_p1);
  bind("FLAQM_d_lf_N_p2_", &flp_.FLAQM_d_lf_N_p2);
  bind("FLAQM_d_lf_N_p3_", &flp_.FLAQM_d_lf_N_p3);
  bind("FLAQM_d_lf_N_p4_", &flp_.FLAQM_d_lf_N_p4);

  bind("FLAQM_d_lf_P_p1_", &flp_.FLAQM_d_lf_P_p1);
  bind("FLAQM_d_lf_P_p2_", &flp_.FLAQM_d_lf_P_p2);
  bind("FLAQM_d_lf_P_p3_", &flp_.FLAQM_d_lf_P_p3);                      66
  bind("FLAQM_d_lf_P_p4_", &flp_.FLAQM_d_lf_P_p4);
  bind("FLAQM_d_lf_P_p5_", &flp_.FLAQM_d_lf_P_p5);

  bind("FLAQM_MD_p1_", &flp_.FLAQM_MD_p1);
  bind("FLAQM_MD_p2_", &flp_.FLAQM_MD_p2);
  bind("FLAQM_MD_p3_", &flp_.FLAQM_MD_p3);
  bind("FLAQM_MD_p4_", &flp_.FLAQM_MD_p4);
  bind("FLAQM_MD_p5_", &flp_.FLAQM_MD_p5);
  bind("FLAQM_MD_p6_", &flp_.FLAQM_MD_p6);


  bind("FLAQM_AI_p1_", &flp_.FLAQM_AI_p1);
  bind("FLAQM_AI_p2_", &flp_.FLAQM_AI_p2);
  bind("FLAQM_AI_p3_", &flp_.FLAQM_AI_p3);
  bind("FLAQM_AI_p4_", &flp_.FLAQM_AI_p4);
  bind("FLAQM_AI_p5_", &flp_.FLAQM_AI_p5);
  bind("FLAQM_AI_p6_", &flp_.FLAQM_AI_p6);

  // IFLAQM parameters
  bind("IFLAQM_lf_p1_", &flp_.IFLAQM_lf_p1);                            86
  bind("IFLAQM_lf_p2_", &flp_.IFLAQM_lf_p2);
  bind("IFLAQM_lf_p3_", &flp_.IFLAQM_lf_p3);


  bind("IFLAQM_d_lf_N_p1_", &flp_.IFLAQM_d_lf_N_p1);
  //      bind("IFLAQM_d_lf_N_p2_", &flp_.IFLAQM_d_lf_N_p2);

  bind("IFLAQM_d_lf_P_p1_", &flp_.IFLAQM_d_lf_P_p1);
  bind("IFLAQM_d_lf_P_p2_", &flp_.IFLAQM_d_lf_P_p2);

  bind("IFLAQM_MD_p1_", &flp_.IFLAQM_MD_p1);
  bind("IFLAQM_MD_p2_", &flp_.IFLAQM_MD_p2);
  bind("IFLAQM_MD_p3_", &flp_.IFLAQM_MD_p3);
  bind("IFLAQM_MD_p4_", &flp_.IFLAQM_MD_p4);
  bind("IFLAQM_MD_p5_", &flp_.IFLAQM_MD_p5);
  bind("IFLAQM_MD_p6_", &flp_.IFLAQM_MD_p6);
  bind("IFLAQM_MD_p7_", &flp_.IFLAQM_MD_p7);


  bind("IFLAQM_AI_p1_", &flp_.IFLAQM_AI_p1);                            106
  bind("IFLAQM_AI_p2_", &flp_.IFLAQM_AI_p2);
  bind("IFLAQM_AI_p3_", &flp_.IFLAQM_AI_p3);
  bind("IFLAQM_AI_p4_", &flp_.IFLAQM_AI_p4);
  bind("IFLAQM_AI_p5_", &flp_.IFLAQM_AI_p5);
  bind("IFLAQM_AI_p6_", &flp_.IFLAQM_AI_p6);
  bind("IFLAQM_AI_p7_", &flp_.IFLAQM_AI_p7);

  // shared parameters
  bind("delta_", &flp_.p_delta); //neighourhood for steady region
  bind_bool("aggressive_", &flp_.aggressive);//when q<qref =>fraction=1
```

```
  bind("stepnumber_", &flp_.p_stepnumber); //for compute in FL
  bind("inw_", &flp_.p_inw);  //weight for averaging count
                              //in a measurement duration
  bind("q_weight_", &flp_.q_w);    // for EWMA
  bind("mean_pktsize_", &flp_.p_pktsize);
  bind("pupdtime_", &flp_.p_updtime);
  bind("pbo_", &flp_.p_bo); // ideal delay-> ideal queue size
  bind("prob_", &flv_.v_prob); // dropping probability pr
  bind("curq_", &curq_);          // current queue size
  bind("ave_", &flv_.v_ave);//average of count in a measurement duration   126
  bind("qave_", &flv_.v_qave); //average of queue in RED
  bind("deltalf", &flv_.v_deltaz);
  bind("loadfactor", &flv_.v_z);
  bind("QDLF_", &flp_.p_QDLF);
  bind("a_", &flp_.p_a);
  bind("b_", &flp_.p_b);
  bind("pmark_", &pmark_);      //number of packets being marked
  bind_bool("markpkts_", &markpkts_);/* Whether to mark or drop?*/
  bind_bool("qib_", &qib_); /* queue in bytes? */
  bind_bool("wait_", &flp_.wait); //"wait" indicates
                                  //whether the gateway should
  bind_bool("droplikered_", &flp_.p_droplikered);
  //"uniformly drop incoming packets like red.
  bind_bool("deterministic_", &flp_.p_deterministic);
  //"deterministic drop incoming packets like red.
  bind("K", &flp_.K);      //cal input rate default 0.1
  bind_bool("input_greenlike_", &flp_.input_greenlike);
  //"flag for cal input rate like GREEN.

  q_ = new PacketQueue();           // underlying queue                    146
  pq_ = q_;
  //reset();

#ifdef notdef
print_flp();
print_flv();
#endif

}

/*
 * Note: if the link bandwidth changes in the course of the
 * simulation, the bandwidth-dependent RED parameters do not change.
 * This should be fixed, but it would require some extra parameters,
 * and didn't seem worth the trouble...
 */
void FLQueue::initialize_params()
{
 /*
  * If q_weight=0, set it to a reasonable value of 1-exp(-1/C)            166
  * This corresponds to choosing q_weight to be of that value for
  * which the packet time constant -1/ln(1-q_weight) per default RTT
  * of 100ms is an order of magnitude more than the link capacity, C.
  *
  * If q_weight=-1, then the queue weight is set to be a function of
  * the bandwidth and the link propagation delay.  In particular,
  * the default RTT is assumed to be three times the link delay and
  * transmission delay, if this gives a default RTT greater than 100 ms.
  *
  * If q_weight=-2, set it to a reasonable value of 1-exp(-10/C).
  */
  if (flp_.q_w == 0.0) {
        flp_.q_w = 1.0 - exp(-1.0/flp_.ptc);
  }
}

void FLQueue::reset()
{
 //printf("here is reset\n");
 /*                                                                       186
  * Compute the "packet time constant" if we know the
```

```
    * link bandwidth.  The ptc is the max number of
    * pkts per second which can be placed on the link.
    * The link bw is given in bits/sec, so scale psize
    * accordingly.
    */
    if (link_) {
        flp_.ptc = link_->bandwidth() / (8. * flp_.p_pktsize);
        initialize_params();
    }
    flv_.v_prob = 0.0; //pr
    flv_.v_z = 0.0; //lf
    flv_.v_z_old = 0.0; //for compute deltalf
    flv_.v_deltaz = 0.0; //deltalf
    flv_.v_ave = 0.0; //average of count during a measurement period
    flv_.v_qave = 0.0;
    flv_.v_qlen_old = 0;//leftover queue at the end last period
    flv_.v_count = 0.0;// count during a measurement period
    flv_.count = 0; //count for RED
    flv_.count_bytes = 0;                                              206
    flv_.deterministic_r = 0.0; //parameter r in deterministic
    flv_.est = 0.0; // est input rate as GREEN
    arrival_time_old = 0.0;// compute GREEN input rate
    idle_ = 1;
    if (&Scheduler::instance() != NULL)
        idletime_ = Scheduler::instance().clock();
    else
        idletime_ = 0.0; /* sched not instantiated yet */

    pmark_ = 0.0;

    Queue::reset();
    set_update_timer();
}

/*
 * Make uniform instead of geometric interdrop periods.
 */
double
FLQueue::modify_p(double p, int count, int count_bytes, int bytes,    226
                  int mean_pktsize, int wait, int size)
{
    double count1 = (double) count;
    if (bytes)
    count1 = (double) (count_bytes/mean_pktsize);
    if (wait) {
        if (count1 * p < 1.0)
            p = 0.0;
        else if (count1 * p < 2.0)
            p /= (2 - count1 * p);
        else
            p = 1.0;
    } else {
        if (count1 * p < 1.0)
            p /= (1.0 - count1 * p);
     else
            p = 1.0;
    }
    if (bytes && p < 1.0) {
      p = p * size / mean_pktsize;                                     246
    }
    if (p > 1.0)
      p = 1.0;
    return p;
}


/*
 * Compute the load factor, and its change, and the marking prob..
 */
//for FLAQM (FLAQM(I))
void FLQueue::FLAQM_run_updaterule()
```

```
{
  //printf("here in update\n");
  double fraction, target_capacity, load_factor, in, in_avg;
  double fq1;
  double pr, delta_pr;
  // firstly, input rate
  // in_avg is the low pss filtered input rate
  // which is in bytes if qib_ is true and in packets otherwise.

  in = flv_.v_count;
  in_avg = flv_.v_ave;

  in_avg *= (1.0 - flp_.p_inw);

  if (qib_) {
        in_avg += flp_.p_inw*in/flp_.p_pktsize;
        //       nqueued = bcount_/flp_.p_pktsize;
  }
  else {
        in_avg += flp_.p_inw*in;
       //       nqueued = q_ -> length();
  }

  // secondly, calculate fraction for use some capacity to drain the queue
  int qlen = q_->length();
  //if(flv_.v_qave_old <= flp_.p_bo) {
  if(flv_.v_qlen_old <= flp_.p_bo) {
     //fraction = (flp_.p_b*flp_.p_bo)/((flp_.p_b-1)*flv_.v_qave_old+flp_.p_bo);
     if (!flp_.aggressive)
        fraction = 1;
     else
        fraction = 1 + (flp_.p_bo-flv_.v_qlen_old)*8.0*
                   flp_.p_pktsize/flp_.p_updtime/link_->bandwidth();
  }
  else {
     //fq1 = (flp_.p_a*flp_.p_bo)/((flp_.p_a-1)*flv_.v_qave_old+flp_.p_bo);
     fq1 = 1 - (flv_.v_qlen_old-flp_.p_bo)*8.0*
          flp_.p_pktsize/flp_.p_updtime/link_->bandwidth();
     fraction =  fq1;
     if (flp_.p_QDLF > fq1)
        fraction = flp_.p_QDLF;
  }

  // thirdly, target capacity and load factor
  double        load_factor1, load_factor2;
  target_capacity = fraction * link_->bandwidth();
  if(in == 0)
     flv_.est = 0.0;
  load_factor1 = flv_.est/target_capacity;
  if (load_factor1 > 200) {
     load_factor1 = 200;
  }
  load_factor2 = (in_avg*flp_.p_pktsize*8.0/flp_.p_updtime)/
                 target_capacity;
  if (load_factor2 > 200) {
     load_factor2 = 200;
  }
  if (flp_.input_greenlike)
     flv_.v_z = load_factor1;
  else
     flv_.v_z = load_factor2;
  double now = Scheduler::instance().clock();

  // forthly, calculate the drop probability
  flv_.v_deltaz = flv_.v_z - flv_.v_z_old;
  if (flv_.v_z <= 1.0 + flp_.p_delta)
     pr = 0.0;
  else {
     if (flv_.v_deltaz < 0) {
       if (flv_.v_prob == 0.0) {
          pr = 0.0;
```

```
        } else {
            delta_pr = FLAQM_fl_MD_AQM(flv_.v_z, flv_.v_deltaz,
                        flp_.p_stepnumber);
            pr = delta_pr * flv_.v_prob;
            if(pr - flv_.v_prob > 0.025) {
               pr = 0.025 + flv_.v_prob;
            }
        }
      } else {
        delta_pr = FLAQM_fl_AI_AQM(flv_.v_z, flv_.v_deltaz,
                    flp_.p_stepnumber);
        pr = delta_pr + flv_.v_prob;
      }
    }
    if (pr < 0.0)
        pr = 0.0;
    else if (pr > 0.5)                                          346
        pr = 0.5;

    // fifthly, reset the variables
    flv_.v_count = 0.0;
    flv_.v_ave = in_avg;
    flv_.v_qlen_old = qlen;
    flv_.v_z_old = flv_.v_z;
    flv_.v_prob = pr;
}

double FLQueue::FLAQM_fl_MD_AQM(double lf, double delta_lf,
                int step_number)
{
  int sn = step_number;
  double ss = (flp_.FLAQM_MD_p6+flp_.FLAQM_MD_p1)/sn; //step_size
  double val1, val2, w, val3;

  //rule1:
  val1 = FLAQM_H1(lf);
  val2 = FLAQM_N5(delta_lf);                                    366
  w = min2(val1, val2);
  double min_1[sn+1];
  for (int i = 0; i <= sn; i++) {
     val3 = FLAQM_MD_Z(-flp_.FLAQM_MD_p1+i*ss);
     if (w <= val3)
        min_1[i] = w;
     else
        min_1[i] = val3;
  }
  //rule2:
  val1 = FLAQM_H1(lf);
  val2 = FLAQM_N4(delta_lf);
  w = min2(val1, val2);
  double min_2[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P1(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_2[i] = w;
    else
      min_2[i] = val3;                                          386
  }
  //rule3:
  val1 = FLAQM_H1(lf);
  val2 = FLAQM_N3(delta_lf);
  w = min2(val1, val2);
  double min_3[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P2(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
        min_3[i] = w;
   else
        min_3[i] = val3;
  }
```

```
//rule4:
val1 = FLAQM_H1(lf);
val2 = FLAQM_N2(delta_lf);
w = min2(val1, val2);
double min_4[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);                    406
  if (w <= val3)
    min_4[i] = w;
  else
    min_4[i] = val3;
}
//rule5:
val1 = FLAQM_H1(lf);
val2 = FLAQM_N1(delta_lf);
w = min2(val1, val2);
double min_5[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P4(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_5[i] = w;
  else
    min_5[i] = val3;
}
//rule12:
val1 = FLAQM_H2(lf);
val2 = FLAQM_N5(delta_lf);                                       426
w = min2(val1, val2);
double min_12[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P1(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_12[i] = w;
  else
    min_12[i] = val3;
}
      //rule13:
val1 = FLAQM_H2(lf);
val2 = FLAQM_N4(delta_lf);
w = min2(val1, val2);
double min_13[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P2(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_13[i] = w;
  else
    min_13[i] = val3;                                            446
}
//rule14:
val1 = FLAQM_H2(lf);
val2 = FLAQM_N3(delta_lf);
w = min2(val1, val2);
double min_14[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_14[i] = w;
  else
    min_14[i] = val3;
}
//rule15:
val1 = FLAQM_H2(lf);
val2 = FLAQM_N2(delta_lf);
w = min2(val1, val2);
double min_15[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);                    466
  if (w <= val3)
    min_15[i] = w;
```

```
    else
      min_15[i] = val3;
  }
  //rule16:
  val1 = FLAQM_H2(lf);
  val2 = FLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_16[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P4(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_16[i] = w;
    else
      min_16[i] = val3;
  }
  //rule23:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_N5(delta_lf);                                    486
  w = min2(val1, val2);
  double min_23[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P2(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_23[i] = w;
    else
      min_23[i] = val3;
  }
  //rule24:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_N4(delta_lf);
  w = min2(val1, val2);
  double min_24[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_24[i] = w;
    else
      min_24[i] = val3;                                         506
  }
  //rule25:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_N3(delta_lf);
  w = min2(val1, val2);
  double min_25[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_25[i] = w;
    else
      min_25[i] = val3;
  }
  //rule26:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_N2(delta_lf);
  w = min2(val1, val2);
  double min_26[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P4(-flp_.FLAQM_MD_p1+i*ss);                 526
    if (w <= val3)
      min_26[i] = w;
    else
      min_26[i] = val3;
  }
  //rule27:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_27[sn+1];
  for (int i = 0; i <= sn; i++) {
```

```
    val3 = FLAQM_MD_P5(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_27[i] = w;
    else
      min_27[i] = val3;
  }
  //rule34:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_N5(delta_lf);                                          546
  w = min2(val1, val2);
  double min_34[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P2(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_34[i] = w;
    else
      min_34[i] = val3;
  }
  //rule35:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_N4(delta_lf);
  w = min2(val1, val2);
  double min_35[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_35[i] = w;
    else
      min_35[i] = val3;                                               566
  }
  //rule36:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_N3(delta_lf);
  w = min2(val1, val2);
  double min_36[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P4(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_36[i] = w;
    else
      min_36[i] = val3;
  }
  //rule37:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_N2(delta_lf);
  w = min2(val1, val2);
  double min_37[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P5(-flp_.FLAQM_MD_p1+i*ss);                       586
    if (w <= val3)
      min_37[i] = w;
    else
      min_37[i] = val3;
  }
  //rule38:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_38[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_MD_P5(-flp_.FLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_38[i] = w;
    else
      min_38[i] = val3;
  }
  //rule45:
  val1 = FLAQM_H5(lf);
  val2 = FLAQM_N5(delta_lf);                                          606
```

```
w = min2(val1, val2);
double min_45[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P3(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_45[i] = w;
  else
    min_45[i] = val3;
}
//rule46:
val1 = FLAQM_H5(lf);
val2 = FLAQM_N4(delta_lf);
w = min2(val1, val2);
double min_46[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P4(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_46[i] = w;
  else
    min_46[i] = val3;
}
//rule47:
val1 = FLAQM_H5(lf);
val2 = FLAQM_N3(delta_lf);
w = min2(val1, val2);
double min_47[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P5(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_47[i] = w;
  else
    min_47[i] = val3;
}
//rule48:
val1 = FLAQM_H5(lf);
val2 = FLAQM_N2(delta_lf);
w = min2(val1, val2);
double min_48[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P5(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_48[i] = w;
  else
    min_48[i] = val3;
}
//rule49:
val1 = FLAQM_H5(lf);
val2 = FLAQM_N1(delta_lf);
w = min2(val1, val2);
double min_49[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_MD_P5(-flp_.FLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_49[i] = w;
  else
    min_49[i] = val3;
}

//maximize
double MAX[sn+1], xMAX[sn+1] ;
for (int i = 0; i <= sn; i++) {
  double x[] = {min_1[i], min_2[i], min_3[i], min_4[i],
                min_5[i], /*min_6[i], min_7[i], min_8[i],
                min_9[i], min_10[i], min_11[i],*/ min_12[i],
                min_13[i], min_14[i], min_15[i], min_16[i],
                /*min_17[i], min_18[i], min_19[i], min_20[i],
                min_21[i], min_22[i],*/ min_23[i], min_24[i],
                min_25[i], min_26[i], min_27[i], /*min_28[i],
```

```
                min_29[i], min_30[i], min_31[i], min_32[i],
                min_33[i],*/ min_34[i], min_35[i], min_36[i],
                min_37[i], min_38[i], /*min_39[i], min_40[i],
                min_41[i], min_42[i], min_43[i], min_44[i],*/
                min_45[i], min_46[i], min_47[i], min_48[i],
                min_49[i]/*, min_50[i], min_51[i], min_52[i],
                min_53[i], min_54[i], min_55[i]*/};

    MAX[i] = maxn(x, 25);
    xMAX[i] = (-flp_.FLAQM_MD_p1+i*ss)*MAX[i];
  }                                                                    686

  //defuzzification
  double y1, y2;
  y1 = integral(MAX, ss, sn);
  y2 = integral(xMAX, ss, sn);
  if (y1 == 0) {
    fprintf(stderr, "divided by 0");
  return 0;
  }
  return (y2/y1);
}

/*
 *fuzzy logic controller
 */
double FLQueue::FLAQM_fl_AI_AQM(double lf, double delta_lf,
                int step_number)
{
  int sn = step_number;
  double ss = (flp_.FLAQM_AI_p6+flp_.FLAQM_AI_p1)/sn; //step_size      706
  double val1, val2, w, val3;

  //rule6:
  val1 = FLAQM_H1(lf);
  val2 = FLAQM_P0(delta_lf);
  w = min2(val1, val2);
  double min_6[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_Z(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_6[i] = w;
    else
      min_6[i] = val3;
  }
  //rule7:
  val1 = FLAQM_H1(lf);
  val2 = FLAQM_P1(delta_lf);
  w = min2(val1, val2);
  double min_7[sn+1];
  for (int i = 0; i <= sn; i++) {                                      726
    val3 = FLAQM_AI_Z(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_7[i] = w;
    else
      min_7[i] = val3;
  }
  //rule8:
  val1 = FLAQM_H1(lf);
  val2 = FLAQM_P2(delta_lf);
  w = min2(val1, val2);
  double min_8[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_P1(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_8[i] = w;
    else
      min_8[i] = val3;
  }
  //rule9:
```

```
val1 = FLAQM_H1(lf);
val2 = FLAQM_P3(delta_lf);
w = min2(val1, val2);
double min_9[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P1(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_9[i] = w;
  else
    min_9[i] = val3;
}
//rule10:
val1 = FLAQM_H1(lf);
val2 = FLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_10[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_10[i] = w;
  else
    min_10[i] = val3;
}
//rule11:
val1 = FLAQM_H1(lf);
val2 = FLAQM_P5(delta_lf);
w = min2(val1, val2);
double min_11[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_11[i] = w;
  else
    min_11[i] = val3;
}
//rule17:
val1 = FLAQM_H2(lf);
val2 = FLAQM_P0(delta_lf);
w = min2(val1, val2);
double min_17[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_Z(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_17[i] = w;
  else
    min_17[i] = val3;
}
//rule18:
val1 = FLAQM_H2(lf);
val2 = FLAQM_P1(delta_lf);
w = min2(val1, val2);
double min_18[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P1(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_18[i] = w;
  else
    min_18[i] = val3;
}
//rule19:
val1 = FLAQM_H2(lf);
val2 = FLAQM_P2(delta_lf);
w = min2(val1, val2);
double min_19[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_19[i] = w;
  else
```

```
    min_19[i] = val3;
}
//rule20:
val1 = FLAQM_H2(lf);
val2 = FLAQM_P3(delta_lf);
w = min2(val1, val2);
double min_20[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_20[i] = w;
  else                                                          826
    min_20[i] = val3;
}
//rule21:
val1 = FLAQM_H2(lf);
val2 = FLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_21[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_21[i] = w;
  else
    min_21[i] = val3;
}
//rule22:
val1 = FLAQM_H2(lf);
val2 = FLAQM_P5(delta_lf);
w = min2(val1, val2);
double min_22[sn+1];
for (int i = 0; i <= sn; i++) {                                 846
  val3 = FLAQM_AI_P3(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_22[i] = w;
  else
    min_22[i] = val3;
}
//rule28:
val1 = FLAQM_H3(lf);
val2 = FLAQM_P0(delta_lf);
w = min2(val1, val2);
double min_28[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P1(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_28[i] = w;
  else
    min_28[i] = val3;
}
//rule29:
val1 = FLAQM_H3(lf);                                            866
val2 = FLAQM_P1(delta_lf);
w = min2(val1, val2);
double min_29[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P1(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_29[i] = w;
  else
    min_29[i] = val3;
}
//rule30: lf is H, delta_lf is NBB, then prob is P1
val1 = FLAQM_H3(lf);
val2 = FLAQM_P2(delta_lf);
w = min2(val1, val2);
double min_30[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
```

```
    if (w <= val3)
      min_30[i] = w;
    else
      min_30[i] = val3;
  }
  //rule31:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_P3(delta_lf);
  w = min2(val1, val2);
  double min_31[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_31[i] = w;
    else
      min_31[i] = val3;
  }
  //rule32:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_P4(delta_lf);
  w = min2(val1, val2);
  double min_32[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_P3(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_32[i] = w;
    else
      min_32[i] = val3;
  }
  //rule33:
  val1 = FLAQM_H3(lf);
  val2 = FLAQM_P5(delta_lf);
  w = min2(val1, val2);
  double min_33[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_P3(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_33[i] = w;
    else
      min_33[i] = val3;
  }
  //rule39:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_P0(delta_lf);
  w = min2(val1, val2);
  double min_39[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_P1(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_39[i] = w;
    else
      min_39[i] = val3;
  }
  //rule40:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_P1(delta_lf);
  w = min2(val1, val2);
  double min_40[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_40[i] = w;
    else
      min_40[i] = val3;
  }
  //rule41:
  val1 = FLAQM_H4(lf);
  val2 = FLAQM_P2(delta_lf);
  w = min2(val1, val2);
```

```
double min_41[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P3(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_41[i] = w;
  else
    min_41[i] = val3;
}
//rule42:
val1 = FLAQM_H4(lf);
val2 = FLAQM_P3(delta_lf);
w = min2(val1, val2);
double min_42[sn+1];
for (int i = 0; i <= sn; i++) {                                    966
  val3 = FLAQM_AI_P3(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_42[i] = w;
  else
    min_42[i] = val3;
}
//rule43:
val1 = FLAQM_H4(lf);
val2 = FLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_43[sn+1];
for (int i = 0; i <= sn; i++) {
 val3 = FLAQM_AI_P4(-flp_.FLAQM_AI_p1+i*ss);
 if (w <= val3)
   min_43[i] = w;
 else
   min_43[i] = val3;
}
//rule44:
val1 = FLAQM_H4(lf);                                               986
val2 = FLAQM_P5(delta_lf);
w = min2(val1, val2);
double min_44[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P4(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_44[i] = w;
  else
    min_44[i] = val3;
}
//rule50:
val1 = FLAQM_H5(lf);
val2 = FLAQM_P0(delta_lf);
w = min2(val1, val2);
double min_50[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P2(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_50[i] = w;
  else                                                            1006
    min_50[i] = val3;
}
//rule51:
val1 = FLAQM_H5(lf);
val2 = FLAQM_P1(delta_lf);
w = min2(val1, val2);
double min_51[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P3(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_51[i] = w;
  else
    min_51[i] = val3;
}
//rule52:
```

```
val1 = FLAQM_H5(lf);
val2 = FLAQM_P2(delta_lf);
w = min2(val1, val2);
double min_52[sn+1];
for (int i = 0; i <= sn; i++) {                                    1026
  val3 = FLAQM_AI_P4(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_52[i] = w;
  else
    min_52[i] = val3;
}
//rule53:
val1 = FLAQM_H5(lf);
val2 = FLAQM_P3(delta_lf);
w = min2(val1, val2);
double min_53[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P4(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_53[i] = w;
  else
    min_53[i] = val3;
}
//rule54:
val1 = FLAQM_H5(lf);                                               1046
val2 = FLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_54[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P5(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_54[i] = w;
  else
    min_54[i] = val3;
}
//rule55:
val1 = FLAQM_H5(lf);
val2 = FLAQM_P5(delta_lf);
w = min2(val1, val2);
double min_55[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = FLAQM_AI_P5(-flp_.FLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_55[i] = w;
  else                                                             1066
    min_55[i] = val3;
}


//maximize
double MAX[sn+1], xMAX[sn+1] ;
for (int i = 0; i <= sn; i++) {
  double x[] = {/*min_1[i], min_2[i], min_3[i], min_4[i],
              min_5[i],*/ min_6[i], min_7[i], min_8[i],
              min_9[i], min_10[i], min_11[i], /*min_12[i],
              min_13[i], min_14[i], min_15[i], min_16[i],*/
              min_17[i], min_18[i], min_19[i], min_20[i],
              min_21[i], min_22[i], /*min_23[i], min_24[i],
              min_25[i], min_26[i], min_27[i],*/ min_28[i],
              min_29[i], min_30[i], min_31[i], min_32[i],
              min_33[i], /*min_34[i], min_35[i], min_36[i],
              min_37[i], min_38[i],*/ min_39[i], min_40[i],
              min_41[i], min_42[i], min_43[i], min_44[i],
              /*min_45[i], min_46[i], min_47[i], min_48[i],
              min_49[i],*/ min_50[i], min_51[i], min_52[i],        1086
              min_53[i], min_54[i], min_55[i]};

  MAX[i] = maxn(x, 30);
  xMAX[i] = (-flp_.FLAQM_AI_p1+i*ss)*MAX[i];
```

```
  }

  //defuzzification
  double y1, y2;
  y1 = integral(MAX, ss, sn);
  y2 = integral(xMAX, ss, sn);
  if (y1 == 0) {
    fprintf(stderr, "divided by 0");
    return 0;
  }
  return (y2/y1);
}


                                                                    1106
// for IFLAQM (FLAQM(II))

void FLQueue::IFLAQM_run_updaterule()
{
  //printf("here in update\n");
  double fraction, target_capacity, load_factor, in, in_avg;
  double fq1;
  double pr, delta_pr;
  // firstly, input rate
  // in_avg is the low pss filtered input rate
  // which is in bytes if qib_ is true and in packets otherwise.

  in = flv_.v_count;
  in_avg = flv_.v_ave;

  in_avg *= (1.0 - flp_.p_inw);

  //make sure in_avg is in pkt
  if (qib_) {                                                       1126
    in_avg += flp_.p_inw*in/flp_.p_pktsize;
    //  nqueued = bcount_/flp_.p_pktsize;
  }
  else {
    in_avg += flp_.p_inw*in;
    //  nqueued = q_ -> length();
  }

  // secondly, calculate fraction for use some capacity
  //to drain the queue
  int qlen = q_->length();
  if(flv_.v_qlen_old <= flp_.p_bo) {
    if (!flp_.aggressive)
      fraction = 1;
    else {
      fraction = 1 + (flp_.p_bo-flv_.v_qlen_old)*8.0*
                 flp_.p_pktsize/flp_.p_updtime/link_->bandwidth();
    }
  }
  else {                                                            1146
    fq1 = 1 - (flv_.v_qlen_old-flp_.p_bo)*8.0*
              flp_.p_pktsize/flp_.p_updtime/link_->bandwidth();
    fraction =  fq1;
    if (flp_.p_QDLF > fq1)
      fraction = flp_.p_QDLF;
  }

  // thirdly, target capacity and load factor
  double        load_factor1, load_factor2;
  target_capacity = fraction * link_->bandwidth();
  if(in == 0)
    flv_.est = 0.0;
  if (flv_.est  == 0)
    load_factor1 = 500;
  else {
```

```
    load_factor1 = target_capacity/flv_.est;
    if (load_factor1 > 500)
      load_factor1 = 500;
  }
  if (in_avg == 0)
    load_factor2 = 500;
  else {
    load_factor2 = target_capacity/(in_avg*flp_.p_pktsize*
                   8.0/flp_.p_updtime);
    if (load_factor2 > 500)
       load_factor2 = 500;
  }
  if (flp_.input_greenlike)
    flv_.v_z = load_factor1;
  else
    flv_.v_z = load_factor2;
  double now = Scheduler::instance().clock();

  // forthly, calculate the drop probability
  flv_.v_deltaz = flv_.v_z - flv_.v_z_old;
  if (flv_.v_deltaz < -2)
    flv_.v_deltaz = -2;
  if (flv_.v_z >= 1.0 + flp_.p_delta)
    pr = 0.0;
  else {
    if (flv_.v_deltaz <= 0) {
      delta_pr = IFLAQM_fl_AI_AQM(flv_.v_z, flv_.v_deltaz,
                 flp_.p_stepnumber);
      pr = delta_pr + flv_.v_prob;
    } else {
      if (flv_.v_prob == 0.0) {
        pr = 0.0;
       } else {
        delta_pr = IFLAQM_fl_MD_AQM(flv_.v_z, flv_.v_deltaz,
                                    flp_.p_stepnumber);
        pr = delta_pr * flv_.v_prob;
        if(pr - flv_.v_prob > 0.025) {
          pr = 0.025 + flv_.v_prob;
        }
      }
    }
  }

  if (pr < 0.0)
    pr = 0.0;
  else if (pr > 0.5)
    pr = 0.5;

  // fifthly, reset the variables
  flv_.v_count = 0.0;
  flv_.v_ave = in_avg;
  flv_.v_qlen_old = qlen;
  flv_.v_z_old = flv_.v_z;
  flv_.v_prob = pr;
  //     printf(" pro %6.4f\n", pr);
}

/*
 *fuzzy logic controller
 */
double FLQueue::IFLAQM_fl_MD_AQM(double lf, double delta_lf,
                  int step_number)
{
  int sn = step_number;
  double ss = (flp_.IFLAQM_MD_p7+flp_.IFLAQM_MD_p1)/sn; //step_size
  double val1, val2, w, val3;

  //rule1:
  val1 = IFLAQM_H1(lf);
  val2 = IFLAQM_P1(delta_lf);
```

```
w = min2(val1, val2);
double min_1[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P7(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_1[i] = w;
  else
    min_1[i] = val3;
}
//rule2:
val1 = IFLAQM_H1(lf);
val2 = IFLAQM_P2(delta_lf);
w = min2(val1, val2);
double min_2[sn+1];                                          1246
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P7(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_2[i] = w;
  else
    min_2[i] = val3;
}
//rule3:
val1 = IFLAQM_H1(lf);
val2 = IFLAQM_P3(delta_lf);
w = min2(val1, val2);
double min_3[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P6(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_3[i] = w;
  else
    min_3[i] = val3;
}
//rule4:                                                     1266
val1 = IFLAQM_H1(lf);
val2 = IFLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_4[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P5(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_4[i] = w;
  else
    min_4[i] = val3;
}

//rule12:
val1 = IFLAQM_H2(lf);
val2 = IFLAQM_P1(delta_lf);
w = min2(val1, val2);
double min_12[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P7(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)                                             1286
    min_12[i] = w;
  else
    min_12[i] = val3;
}
//rule13:
val1 = IFLAQM_H2(lf);
val2 = IFLAQM_P2(delta_lf);
w = min2(val1, val2);
double min_13[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P6(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_13[i] = w;
  else
    min_13[i] = val3;
```

```
}
//rule14:
val1 = IFLAQM_H2(lf);
val2 = IFLAQM_P3(delta_lf);
w = min2(val1, val2);                                          1306
double min_14[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P5(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_14[i] = w;
  else
    min_14[i] = val3;
}
//rule15:
val1 = IFLAQM_H2(lf);
val2 = IFLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_15[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P4(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_15[i] = w;
  else
    min_15[i] = val3;
}                                                              1326

 //rule23:
val1 = IFLAQM_H3(lf);
val2 = IFLAQM_P1(delta_lf);
w = min2(val1, val2);
double min_23[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P6(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_23[i] = w;
  else
    min_23[i] = val3;
}
//rule24:
val1 = IFLAQM_H3(lf);
val2 = IFLAQM_P2(delta_lf);
w = min2(val1, val2);
double min_24[sn+1];
for (int i = 0; i <= sn; i++) {                                1346
  val3 = IFLAQM_MD_P5(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_24[i] = w;
  else
    min_24[i] = val3;
}
//rule25:
val1 = IFLAQM_H3(lf);
val2 = IFLAQM_P3(delta_lf);
w = min2(val1, val2);
double min_25[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_MD_P4(-flp_.IFLAQM_MD_p1+i*ss);
  if (w <= val3)
    min_25[i] = w;
  else
    min_25[i] = val3;
}
//rule26:
val1 = IFLAQM_H3(lf);                                          1366
val2 = IFLAQM_P4(delta_lf);
w = min2(val1, val2);
double min_26[sn+1];
for (int i = 0; i <= sn; i++) {
```

```
    val3 = IFLAQM_MD_P3(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_26[i] = w;
    else
      min_26[i] = val3;
  }

  //rule34:
  val1 = IFLAQM_H4(lf);
  val2 = IFLAQM_P1(delta_lf);
  w = min2(val1, val2);
  double min_34[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P5(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)                                          1386
      min_34[i] = w;
    else
      min_34[i] = val3;
  }
  //rule35:
  val1 = IFLAQM_H4(lf);
  val2 = IFLAQM_P2(delta_lf);
  w = min2(val1, val2);
  double min_35[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P4(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_35[i] = w;
    else
      min_35[i] = val3;
  }
  //rule36:
  val1 = IFLAQM_H4(lf);
  val2 = IFLAQM_P3(delta_lf);
  w = min2(val1, val2);                                     1406
  double min_36[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P3(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_36[i] = w;
    else
      min_36[i] = val3;
  }
  //rule37:
  val1 = IFLAQM_H4(lf);
  val2 = IFLAQM_P4(delta_lf);
  w = min2(val1, val2);
  double min_37[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P2(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_37[i] = w;
    else
      min_37[i] = val3;
  }                                                         1426

  //rule45:
  val1 = IFLAQM_H5(lf);
  val2 = IFLAQM_P1(delta_lf);
  w = min2(val1, val2);
  double min_45[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P4(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_45[i] = w;
    else
      min_45[i] = val3;
  }
```

```
  //rule46:
  val1 = IFLAQM_H5(lf);
  val2 = IFLAQM_P2(delta_lf);
  w = min2(val1, val2);
  double min_46[sn+1];
  for (int i = 0; i <= sn; i++) {                                    1446
    val3 = IFLAQM_MD_P3(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_46[i] = w;
    else
      min_46[i] = val3;
  }
  //rule47:
  val1 = IFLAQM_H5(lf);
  val2 = IFLAQM_P3(delta_lf);
  w = min2(val1, val2);
  double min_47[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P2(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_47[i] = w;
    else
      min_47[i] = val3;
  }
  //rule48:
  val1 = IFLAQM_H5(lf);                                              1466
  val2 = IFLAQM_P4(delta_lf);
  w = min2(val1, val2);
  double min_48[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_MD_P1(-flp_.IFLAQM_MD_p1+i*ss);
    if (w <= val3)
      min_48[i] = w;
    else
      min_48[i] = val3;
  }


  //maximize
  double MAX[sn+1], xMAX[sn+1] ;
  for (int i = 0; i <= sn; i++) {
    double x[] = {min_1[i], min_2[i], min_3[i], min_4[i],/*
                  min_5[i], min_6[i], min_7[i], min_8[i],
                  min_9[i], min_10[i], min_11[i],*/ min_12[i],
                  min_13[i], min_14[i], min_15[i], /*min_16[i],
                  min_17[i], min_18[i], min_19[i], min_20[i],        1486
                  min_21[i], min_22[i],*/ min_23[i], min_24[i],
                  min_25[i], min_26[i], /*min_27[i], min_28[i],
                  min_29[i], min_30[i], min_31[i], min_32[i],
                  min_33[i],*/ min_34[i], min_35[i], min_36[i],
                  min_37[i], /*min_38[i], min_39[i], min_40[i],
                  min_41[i], min_42[i], min_43[i], min_44[i],*/
                  min_45[i], min_46[i], min_47[i], min_48[i]/*,
                  min_49[i], min_50[i], min_51[i], min_52[i],
                  min_53[i], min_54[i], min_55[i]*/};

     MAX[i] = maxn(x, 20);
     xMAX[i] = (-flp_.IFLAQM_MD_p1+i*ss)*MAX[i];
  }

  //defuzzification
  double y1, y2;
  y1 = integral(MAX, ss, sn);
  y2 = integral(xMAX, ss, sn);
  if (y1 == 0) {
    fprintf(stderr, "divided by 0");                                 1506
    return 0;
  }
    return (y2/y1);
}
```

```
/*
 *fuzzy logic controller
 */
double FLQueue::IFLAQM_fl_AI_AQM(double lf, double delta_lf,
                int step_number)
{
  int sn = step_number;
  double ss = (flp_.IFLAQM_AI_p7+flp_.IFLAQM_AI_p1)/sn; //step_size
  double val1, val2, w, val3;

  //rule6:
  val1 = IFLAQM_H1(lf);
  val2 = IFLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_6[sn+1];
  for (int i = 0; i <= sn; i++) {                                    1526
    val3 = IFLAQM_AI_P7(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_6[i] = w;
    else
      min_6[i] = val3;
  }
  //rule7:
  val1 = IFLAQM_H1(lf);
  val2 = IFLAQM_N2(delta_lf);
  w = min2(val1, val2);
  double min_7[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P6(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_7[i] = w;
    else
      min_7[i] = val3;
  }
  //rule8:
  val1 = IFLAQM_H1(lf);                                              1546
  val2 = IFLAQM_N3(delta_lf);
  w = min2(val1, val2);
  double min_8[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P5(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_8[i] = w;
    else
      min_8[i] = val3;
  }

  //rule17:
  val1 = IFLAQM_H2(lf);
  val2 = IFLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_17[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P6(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)                                                   1566
      min_17[i] = w;
    else
      min_17[i] = val3;
  }
  //rule18:
  val1 = IFLAQM_H2(lf);
  val2 = IFLAQM_N2(delta_lf);
  w = min2(val1, val2);
  double min_18[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P5(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_18[i] = w;
    else
```

```
      min_18[i] = val3;
  }
  //rule19:
  val1 = IFLAQM_H2(lf);
  val2 = IFLAQM_N3(delta_lf);
  w = min2(val1, val2);
  double min_19[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P4(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_19[i] = w;
    else
      min_19[i] = val3;
  }

  //rule28:
  val1 = IFLAQM_H3(lf);
  val2 = IFLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_28[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P5(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_28[i] = w;
    else
      min_28[i] = val3;
  }
  //rule29:
  val1 = IFLAQM_H3(lf);
  val2 = IFLAQM_N2(delta_lf);
  w = min2(val1, val2);
  double min_29[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P4(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_29[i] = w;
    else
      min_29[i] = val3;
  }
  //rule30:
  val1 = IFLAQM_H3(lf);
  val2 = IFLAQM_N3(delta_lf);
  w = min2(val1, val2);
  double min_30[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P3(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_30[i] = w;
    else
      min_30[i] = val3;
  }

  //rule39:
  val1 = IFLAQM_H4(lf);
  val2 = IFLAQM_N1(delta_lf);
  w = min2(val1, val2);
  double min_39[sn+1];
  for (int i = 0; i <= sn; i++) {
    val3 = IFLAQM_AI_P4(-flp_.IFLAQM_AI_p1+i*ss);
    if (w <= val3)
      min_39[i] = w;
    else
      min_39[i] = val3;
  }
  //rule40:
  val1 = IFLAQM_H4(lf);
  val2 = IFLAQM_N2(delta_lf);
  w = min2(val1, val2);
```

```
double min_40[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_AI_P3(-flp_.IFLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_40[i] = w;
  else
    min_40[i] = val3;
}
//rule41:
val1 = IFLAQM_H4(lf);
val2 = IFLAQM_N3(delta_lf);
w = min2(val1, val2);
double min_41[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_AI_P2(-flp_.IFLAQM_AI_p1+i*ss);
  if (w <= val3)                                                    1666
    min_41[i] = w;
  else
    min_41[i] = val3;
}

//rule50:
val1 = IFLAQM_H5(lf);
val2 = IFLAQM_N1(delta_lf);
w = min2(val1, val2);
double min_50[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_AI_P3(-flp_.IFLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_50[i] = w;
  else
    min_50[i] = val3;
}
//rule51:
val1 = IFLAQM_H5(lf);                                               1686
val2 = IFLAQM_N2(delta_lf);
w = min2(val1, val2);
double min_51[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_AI_P2(-flp_.IFLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_51[i] = w;
  else
    min_51[i] = val3;
}
//rule52:
val1 = IFLAQM_H5(lf);
val2 = IFLAQM_N3(delta_lf);
w = min2(val1, val2);
double min_52[sn+1];
for (int i = 0; i <= sn; i++) {
  val3 = IFLAQM_AI_P1(-flp_.IFLAQM_AI_p1+i*ss);
  if (w <= val3)
    min_52[i] = w;
  else                                                              1706
    min_52[i] = val3;
}

//maximize
double MAX[sn+1], xMAX[sn+1] ;
for (int i = 0; i <= sn; i++) {
  double x[] = {/*min_1[i], min_2[i], min_3[i], min_4[i],
                min_5[i],*/ min_6[i], min_7[i], min_8[i],
                /*min_9[i], min_10[i], min_11[i], min_12[i],
                min_13[i], min_14[i], min_15[i], min_16[i],*/
                min_17[i], min_18[i], min_19[i], /*min_20[i],
                min_21[i], min_22[i], min_23[i], min_24[i],
                min_25[i], min_26[i], min_27[i],*/ min_28[i],
```

```
                min_29[i], min_30[i], /*min_31[i], min_32[i],
                min_33[i], min_34[i], min_35[i], min_36[i],
                min_37[i], min_38[i],*/ min_39[i], min_40[i],
                min_41[i], /*min_42[i], min_43[i], min_44[i],
                min_45[i], min_46[i], min_47[i], min_48[i],
                min_49[i],*/ min_50[i], min_51[i], min_52[i]/*,
                min_53[i], min_54[i], min_55[i]*/};

    MAX[i] = maxn(x, 15);
    xMAX[i] = (-flp_.IFLAQM_AI_p1+i*ss)*MAX[i];
  }

  //defuzzification
  double y1, y2;
  y1 = integral(MAX, ss, sn);
  y2 = integral(xMAX, ss, sn);
  if (y1 == 0) {
    fprintf(stderr, "divided by 0");
    return 0;
  }
  return (y2/y1);
}


/*
 * Return the next packet in the queue for transmission.
 */
Packet* FLQueue::deque()
{
/* this is the best place to detect when the congestions start and end.
 * for cal Mice/Elephants proportion during congestion for ME */
  double now = Scheduler::instance().clock();
  if (q_->length() >= 2) { /* buffer is not empty */
      if (!no_empty) {
         busytime_ = now;
      }
      no_empty = 1;
  } else { /*buffer is empty*/
    if (no_empty) {
      busydur_ = now - busytime_;
      ME_pro_sum_time_under += busydur_;
      Epkts_pro_over += Epkts_durcong*busydur_;
      Mpkts_pro_over += Mpkts_durcong*busydur_;
      Apkts_pro_over += Apkts_durcong*busydur_;
      Efls_pro_over += elephant_fl_num*busydur_;
      Mfls_pro_over += mice_fl_num*busydur_;
      Afls_pro_over += all_fl_num*busydur_;
      Epkts_durcong = 0;
      Mpkts_durcong = 0;
      Apkts_durcong = 0;
      delete [] flowbase;
      flowbase = 0;
      flowbaseSize = 0;
      nslot_ = 0;
      elephant_fl_num = 0;
      mice_fl_num = 0;
      all_fl_num = 0;
    }
    no_empty = 0;
  }
  /* end for ME */

  Packet *p = q_->deque();
  if (p != 0) {
    idle_ = 0;
  } else {
    if (idle_ && idletime_ > 0) {} //zhi
      else {
        idle_ = 1;
        // deque() may invoked by Queue::reset at init
        // time (before the scheduler is instantiated).
```

```
        // deal with this case
        if (&Scheduler::instance() != NULL)
          idletime_ = Scheduler::instance().clock();
        else
          idletime_ = 0.0;
      }
  }
  return (p);
}


/*
 * Compute the average queue size.
 * Nqueued can be bytes or packets.
 */                                                          1806
double FLQueue::estimator(int nqueued, int m, double ave, double q_w)
{
  double new_ave;
  new_ave = ave;
  while (--m >= 1) {
    new_ave *= 1.0 - q_w;
  }
  new_ave *= 1.0 - q_w;
  new_ave += q_w * nqueued;
  return new_ave;
}

/*
 * Receive a new packet arriving at the queue.
 * The packet is dropped if the maximum queue size is exceeded.
 */

void FLQueue::enque(Packet* pkt)
{
                                                             1826
  /* firstly, m and queue ave
   * if we were idle, we pretend that m packets arrived during
   * the idle period.  m is set to be the ptc times the amount
   * of time we've been idle for
   */
  double now = Scheduler::instance().clock();
  hdr_cmn* ch = hdr_cmn::access(pkt);
  int m = 0;
  if (idle_) {
    // A packet that arrives to an idle queue will never be dropped.
    /* To account for the period when the queue was empty. */
    idle_ = 0;
    // Use idle_pktsize instead of mean_pktsize, for
    // a faster response to idle times.
    m = int(flp_.ptc * (now - idletime_));
  }
  /* for cal Mice/Elephants proportion during congestion for ME
   * here to detect whether congestions start or not.
   */
  if (no_empty) {/*buffer is no-empty*/                       1846
    hdr_ip *iph = hdr_ip::access(pkt);
    Apkts_durcong += ch->size();
    if (iph->MorE) {
      Epkts_durcong += ch->size();
    } else {
      Mpkts_durcong += ch->size();
    }
    if (is_a_new_flow(pkt)) {
      addto_flowbase(pkt);
    }
  }
  /* end for ME cal proportion during congestion */

/*
 * Run the estimator with either 1 new packet arrival, or with
 * the scaled version above [scaled by m due to idle time]
```

```
*/
int qlen = qib_ ? q_->byteLength() : q_->length();
double qave = estimator(qlen, m + 1, flv_.v_qave, flp_.q_w);
flv_.v_qave = qave;                                                    1866

curq_ = qlen; // for trace

/*
 * count and count_bytes keeps a tally of arriving traffic
 * that has not been dropped (i.e. how long, in terms of traffic,
 * it has been since the last early drop)
 */
 /* estimate the input rate by weighted averaging */
double arrival_delay = now - arrival_time_old;
double w_input = exp((-arrival_delay)/flp_.K);
flv_.est = (1-w_input)*(ch->size()*8/arrival_delay)+w_input*flv_.est;
arrival_time_old = now;


/* secondly, input no. during a droprate update period time
 * count the no of inputs within the current fl drop rate update
 */
if (qib_) {
   flv_.v_count += ch->size();                                         1886
}
else {
   ++flv_.v_count;
}

/* thirdly, judge to drop the incoming packet or not
 * with the caculation of flv_.count for uniformly drop
 */
double qlim = qib_ ? (qlim_*flp_.p_pktsize) : qlim_ ;
q_->enque(pkt);
qlen = qib_ ? q_->byteLength() : q_->length();
if (qlen > qlim) {
  flv_.count = 0;
  flv_.count_bytes = 0;
  q_->remove(pkt);
  drop(pkt);
  //printf("overflow\n");
} else {
  double pro = flv_.v_prob;
  if (pro == 0) {                                                      1906
    flv_.count = 0;
    flv_.count_bytes = 0;
    //printf("nonedrop\n");
} else {
   if (flp_.p_deterministic) {
     flv_.deterministic_r += (1-flv_.v_prob);
       if (flv_.deterministic_r >= 1) {
         flv_.deterministic_r -= 1;
       } else {
         if (!markpkts_) {
           q_->remove(pkt);
           drop(pkt);
         } else {
           hdr_flags* hf = hdr_flags::access(pkt);
           if((hf->ect() || hf->ecnecho()) && flv_.v_prob < 0.1) {
              hf->ce() = 1;
              pmark_++;
           } else {
              q_->remove(pkt);
              drop(pkt);                                               1926
           }
       }
     }
  } else {
    double u = Random::uniform();
    if (u == 0)
```

```
        printf ("uniform random number is 0, shouldn't be. \n");
      if (flp_.p_droplikered) {
        pro = modify_p(pro, flv_.count, flv_.count_bytes,
                       qib_, flp_.p_pktsize, flp_.wait, ch->size());
      }
      /*if (qib_) {
        pro = flv_.v_prob*ch->size()/flp_.p_pktsize;
      }*/
        if ( u <= pro ) {
          flv_.count = 0;
          flv_.count_bytes = 0;
          if (!markpkts_) {
            q_->remove(pkt);
            drop(pkt);                                                    1946
          } else {
            hdr_flags* hf = hdr_flags::access(pkt);
            if((hf->ect() || hf->ecnecho()) && flv_.v_prob < 0.1) {
              hf->ce() = 1;
              pmark_++;
            } else {
              q_->remove(pkt);
              drop(pkt);
            }
          }
          //printf("unforced drop\n");
        } else {
          flv_.count++;
          flv_.count_bytes += ch->size();
          //printf("luckynonedrop\n");
        }
      }
    }
  }
  return;                                                                 1966
}

int FLQueue::command(int argc, const char*const* argv)
{

  Tcl& tcl = Tcl::instance();
  if (argc == 2) {
    if (strcmp(argv[1], "reset") == 0) {
      reset();
      return (TCL_OK);
    }
  } else if (argc == 3) {
    // attach a file for variable tracing
    if (strcmp(argv[1], "attach") == 0) {
       int mode;
       const char* id = argv[2];
       tchan_ = Tcl_GetChannel(tcl.interp(), (char*)id, &mode);
       if (tchan_ == 0) {
         tcl.resultf("FL: trace: can't attach %s for writing", id);       1986
         return (TCL_ERROR);
       }
       return (TCL_OK);
    }
    // tell FL about link stats
    if (strcmp(argv[1], "link") == 0) {
      LinkDelay* del = (LinkDelay*)TclObject::lookup(argv[2]);
      if (del == 0) {
        tcl.resultf("FL: no LinkDelay object %s", argv[2]);
        return(TCL_ERROR);
      }
      // set ptc now
      link_ = del;
      flp_.ptc = link_->bandwidth() / (8. * flp_.p_pktsize);
      return (TCL_OK);
    }
```

```
    if (!strcmp(argv[1], "packetqueue-attach")) {
      delete q_;
      if (!(q_ = (PacketQueue*) TclObject::lookup(argv[2])))
        return (TCL_ERROR);                                          2006
      else {
        pq_ = q_;
        return (TCL_OK);
      }
    }
  }
  return (Queue::command(argc, argv));
}

/*
 * Routine called by TracedVar facility when variables change values.
 * Currently used to trace values of avg queue size, drop probability,
 * and the instantaneous queue size seen by arriving packets.
 * Note that the tracing of each var must be enabled in tcl to work.
 */

void
FLQueue::trace(TracedVar* v)
{
  char wrk[500], *p;                                                 2026

  if( ((p = strstr(v->name(), "prob")) == NULL) &&
      ((p = strstr(v->name(), "loadfactor")) == NULL) &&
      ((p = strstr(v->name(), "deltalf")) == NULL) &&
      ((p = strstr(v->name(), "qave")) == NULL) &&
      ((p = strstr(v->name(), "curq")) == NULL)) {
        fprintf(stderr, "FL:unknown trace var %s\n", v->name());
        return;
  }

  if (tchan_) {
    int n;
    double t = Scheduler::instance().clock();
    if (*p == 'c') {
        sprintf(wrk, "Q %g %d", t, int(*((TracedInt*) v)));
    } else {
        sprintf(wrk, "%c %g %g", *p, t,
                double(*((TracedDouble*) v)));
    }
    n = strlen(wrk);                                                 2046
    wrk[n] = '\n';
    wrk[n+1] = 0;
    (void)Tcl_Write(tchan_, wrk, n+1);
  }
  return;
}

/* for debugging help */
void FLQueue::print_flp()
{
        printf("==========\n");
}

void FLQueue::print_flv()
{
        printf("==========\n");
}
```

A.2 The *.tcl* Source Code for Testing FLAQM(I) and FLAQM(II)

A.2.1 FLAQM_experiment1.tcl

```
# FLAQM (FLAQM(I)) + IFLAQM (FLAQM(II))

set simulation_duration [lindex $argv 0]
set random_run [lindex $argv 1]
set num_longbursts [lindex $argv 2]
#set traffic_load [lindex $argv 2]
set ECN [lindex $argv 3]
set QueueType [lindex $argv 4]

set warmuptime [expr ($simulation_duration-2)*1.0/2.0]
# set filelen threshold to distinguish mice and elephants
set filelen_threshold 15
# set period interval for active_conn_no measurment
set interval 25

#open a file for active connection number                                    16
set activef [open ../results/active_conn_num.$QueueType\
            $num_longbursts$random_run w]


#topology parameters
set bottleneck_bdw 1.5; #Mbits
set qlimit 160; #packets
set pareto_shape 1.2; #1.2 or 1.4
set pareto_delta 1.0

#packet average for pareto
#set ave_pkts [expr $pareto_delta*$pareto_shape/($pareto_shape-1)];
set ave_pkts 12;
set int_arrival_time [expr (40+1040*$ave_pkts)*8.0/\
                     ($bottleneck_bdw*1000000)]
#puts "lambda: [expr 1.0/$int_arrival_time]"

remove-all-packet-headers        ;# removes all except common
add-packet-header Flags IP TCP   ;# hdrs reqd for TCP
                                                                            36
set ns [new Simulator]
if {$QueueType == "RED" || $QueueType == "ARED"} {
   Queue/RED set q_weight_ -1
   Queue/RED set setbit_ $ECN
   if {$QueueType == "ARED"} {
      Queue/RED set adaptive_ 1
   }
   Queue/RED set thresh_  30
   Queue/RED set maxthresh_ 90
} elseif {$QueueType == "REM"} {
   Queue/REM set pbo_ 60.0
} elseif {$QueueType == "PI"} {
   Queue/PI set qref_ 60.0
} elseif {$QueueType == "FLAQM" || $QueueType == "IFLAQM"} {
   Queue/FL set delta_ 0.05
   Queue/FL set pupdtime_ 0.5
   Queue/FL set pbo_ 60
   Queue/FL set markpkts_ $ECN
   Queue/FL set droplikered_ false
   Queue/FL set deterministic_ true                                         56

   if {$QueueType == "FLAQM"} {
       Queue/FL set whichFLAQM 0
       Queue/FL set FLAQM_lf_p1_ 1.1
       Queue/FL set FLAQM_lf_p2_ 1.5
       Queue/FL set FLAQM_lf_p3_ 2.0
       Queue/FL set FLAQM_lf_p4_ 2.5
       Queue/FL set FLAQM_lf_p5_ 3.0

       Queue/FL set FLAQM_d_lf_N_p1_ -2.0
       Queue/FL set FLAQM_d_lf_N_p2_ -1.0
       Queue/FL set FLAQM_d_lf_N_p3_ -0.5
       Queue/FL set FLAQM_d_lf_N_p4_ -0.2
```

```
        Queue/FL set FLAQM_d_lf_P_p1_ 0.2
        Queue/FL set FLAQM_d_lf_P_p2_ 0.5
        Queue/FL set FLAQM_d_lf_P_p3_ 1.0
        Queue/FL set FLAQM_d_lf_P_p4_ 1.5
        Queue/FL set FLAQM_d_lf_P_p5_ 2.0

        Queue/FL set FLAQM_MD_p1_ 0.8
        Queue/FL set FLAQM_MD_p2_ 0.85
        Queue/FL set FLAQM_MD_p3_ 0.9
        Queue/FL set FLAQM_MD_p4_ 1.0
        Queue/FL set FLAQM_MD_p5_ 1.1
        Queue/FL set FLAQM_MD_p6_ 1.15

        Queue/FL set FLAQM_AI_p1_ 0.01
        Queue/FL set FLAQM_AI_p2_ 0.02
        Queue/FL set FLAQM_AI_p3_ 0.03
        Queue/FL set FLAQM_AI_p4_ 0.04
        Queue/FL set FLAQM_AI_p5_ 0.05
        Queue/FL set FLAQM_AI_p6_ 0.06
    } elseif {$QueueType == "IFLAQM"} {
        Queue/FL set whichFLAQM 1
        Queue/FL set IFLAQM_lf_p1_ 0.25
        Queue/FL set IFLAQM_lf_p2_ 0.5
        Queue/FL set IFLAQM_lf_p3_ 0.75

        Queue/FL set IFLAQM_d_lf_N_p1_ -0.5

        Queue/FL set IFLAQM_d_lf_P_p1_ 0.25
        Queue/FL set IFLAQM_d_lf_P_p2_ 0.5

        Queue/FL set IFLAQM_MD_p1_ 0.5
        Queue/FL set IFLAQM_MD_p2_ 0.95
        Queue/FL set IFLAQM_MD_p3_ 1.0
        Queue/FL set IFLAQM_MD_p4_ 1.05
        Queue/FL set IFLAQM_MD_p5_ 1.1
        Queue/FL set IFLAQM_MD_p6_ 1.15
        Queue/FL set IFLAQM_MD_p7_ 1.2

        Queue/FL set IFLAQM_AI_p1_ 0.01
        Queue/FL set IFLAQM_AI_p2_ 0.02
        Queue/FL set IFLAQM_AI_p3_ 0.03
        Queue/FL set IFLAQM_AI_p4_ 0.04
        Queue/FL set IFLAQM_AI_p5_ 0.05
        Queue/FL set IFLAQM_AI_p6_ 0.06
        Queue/FL set IFLAQM_AI_p7_ 0.07
    }
}

Agent/TCP set packetSize_ 1000
Agent/TCP set window_ 100
Agent/TCP set ecn_ $ECN


#
#                                        Bottleneck
#        10mb 40ms 100mb 40ms        1.5mb 10ms   10mb 1ms
#SERVERs---------ns-------------ne-----------np----------CLIENTs
#                 core          Access      Premises
#Cable: y=27Mbps x=7Mbps
#ADSL: y=8-155Mbps x=1.5Mbps
#

set n_sHTTP1 [$ns node]
set n_cHTTP1 [$ns node]
set n_s [$ns node]
set n_e [$ns node]
set n_p [$ns node]
$ns duplex-link $n_sHTTP1 $n_s 10Mb 40ms DropTail
$ns duplex-link $n_cHTTP1 $n_p 10Mb 1ms DropTail
$ns duplex-link $n_s $n_e 100Mb 40ms DropTail
if {$QueueType == "ARED"} {
```

```
    $ns simplex-link $n_e $n_p [expr $bottleneck_bdw]Mb 10ms RED
} elseif {$QueueType == "FLAQM" || $QueueType == "IFLAQM"} {
    $ns simplex-link $n_e $n_p [expr $bottleneck_bdw]Mb 10ms FL
} else {
    $ns simplex-link $n_e $n_p [expr $bottleneck_bdw]Mb 10ms $QueueType
}
$ns simplex-link $n_p $n_e [expr $bottleneck_bdw]Mb 10ms DropTail
$ns queue-limit $n_e $n_p $qlimit

set monitorq_ [open monitorqueueEmpty w]
set qmon_ [$ns monitor-queue $n_e $n_p $monitorq_]

set monq [[$ns link $n_e $n_p] queue]
set tchan_ [open $QueueType$num_longbursts$random_run.q w]
$monq attach $tchan_                                                    156
if {$QueueType == "FLAQM" || $QueueType == "IFLAQM"} {
   $monq trace loadfactor
   $monq trace curq_
} else {
    $monq trace curq_
}

#set PPBP traffic
#deal with random varibles and seeds for independent
replications of the simulation
# seed the default RNG
global defaultRNG
$defaultRNG seed 9999

# create the RNGs and set them to the correct substream
set starttime1RNG [new RNG]
set flowSize1RNG [new RNG]
for {set j 0} {$j < $random_run} {incr j} {
   $starttime1RNG next-substream
   $flowSize1RNG next-substream                                         176
}

set numSession 100000000
set starttime1 [new RandomVariable/Exponential]
$starttime1 use-rng $starttime1RNG
$starttime1 set avg_ $int_arrival_time
set flowSize1 [new RandomVariable/Pareto]
$flowSize1 use-rng $flowSize1RNG
$flowSize1 set avg_ $ave_pkts
$flowSize1 set shape_ $pareto_shape

set launchTime 0.0
for {set i 0} {$i < $numSession} {incr i} {
  set fl_size_http1($i) [expr round([$flowSize1 value])]
  if {$fl_size_http1($i) < 2} {
     set fl_size_http1($i) 2;
  } elseif {$fl_size_http1($i) > 1000}  {
     set fl_size_http1($i) 1000
  }
                                                                        196
  set launchTime  [expr $launchTime + [$starttime1 value]]
  set laun_time_http1($i) $launchTime
  if {$laun_time_http1($i) >= $simulation_duration} {
     break
  }
  #  puts "$i: $laun_time_http1($i)  $fl_size_http1($i)"
}
set real_http1_num $i
for {set i 0} {$i < $real_http1_num} {incr i} {
 set tcp_http1($i) [new Agent/TCP/Reno]
 set sink_http1($i) [new Agent/TCPSink]
 $ns attach-agent $n_sHTTP1 $tcp_http1($i)
 $ns attach-agent $n_cHTTP1 $sink_http1($i)
 $ns connect $tcp_http1($i) $sink_http1($i)
 #$tcp_http1($i) set window_ $window
```

```
 set ftp_http1($i) [new Application/FTP]
 $ftp_http1($i) attach-agent $tcp_http1($i)
 $ns at $laun_time_http1($i) "$ftp_http1($i) produce $fl_size_http1($i)"
}
                                                                    216
#set window 20 try 100 first
set longbursts_launchtime 0.0
set longbursts_interval 0.2
for {set i 0} {$i < $num_longbursts} {incr i} {
  set n_slongbursts($i) [$ns node]
  set n_clongbursts($i) [$ns node]
  $ns duplex-link $n_slongbursts($i) $n_s 10Mb 40ms DropTail
  $ns duplex-link $n_clongbursts($i) $n_p 10Mb 1ms DropTail
  set tcp_longbursts($i) [new Agent/TCP/Reno]
  set sink_longbursts($i) [new Agent/TCPSink]
  $ns attach-agent $n_slongbursts($i) $tcp_longbursts($i)
  $ns attach-agent $n_clongbursts($i) $sink_longbursts($i)
  $ns connect $tcp_longbursts($i) $sink_longbursts($i)
  #$tcp_longbursts($i) set window_ $window
  set ftp_longbursts($i) [new Application/FTP]
  $ftp_longbursts($i) attach-agent $tcp_longbursts($i)
  set longbursts_launchtime [expr $longbursts_launchtime + \
        $longbursts_interval]
  $ns at $longbursts_launchtime "$ftp_longbursts($i) start"
  set laun_time_tcp_longbursts($i) $longbursts_launchtime           236
}

set warmup_bthrput 0
set total_bthrput 0
set warmup_bdepartures 0
set warmup_barrivals 0
set warmup_bdrops 0
set total_bdepartures 0
set total_barrivals 0
set total_bdrops 0

proc warmup_time {warmuptime} {
  global qmon_
  global QueueType
  global warmup_bthrput
  global warmup_bdepartures
  global warmup_barrivals
  global warmup_bdrops
  global real_http1_num num_longbursts
  global laun_time_http1 laun_time_tcp_longbursts                   256
  global sink_http1 sink_longbursts
  global tcp_http1 tcp_longbursts
# warmup period network thrput, traffic load, drop rate,
#link utilization
  for {set i 0} {$i < $real_http1_num} {incr i} {
    if {$laun_time_http1($i) < $warmuptime} {
      set next [$sink_http1($i) set next_]
      if {$next > 1} {
        set size [$tcp_http1($i) set packetSize_]
        set warmup_bthrput [expr $warmup_bthrput+($next-1)*$size]
      }
    }
  }
  for {set i 0} {$i < $num_longbursts} {incr i} {
    if {$laun_time_tcp_longbursts($i) < $warmuptime} {
      set next [$sink_longbursts($i) set next_]
      if {$next > 1} {
        set size [$tcp_longbursts($i) set packetSize_]
        set warmup_bthrput [expr $warmup_bthrput+($next-1)*$size]
      }                                                             276
    }
  }

  set warmup_bdepartures [$qmon_ set bdepartures_]
  set warmup_barrivals [$qmon_ set barrivals_]
```

```
  set warmup_bdrops [$qmon_ set bdrops_]
}

# periodically check the active connection number
proc active_conn_no {} {
  global real_http1_num  num_longbursts
  global laun_time_http1
  global tcp_http1
  global fl_size_http1
  global filelen_threshold
  global activef
  global interval

  # for measure active no.
  set active_num 0                                                    296
  set active_num_mice 0
  set active_num_elephant 0

  set ns [Simulator instance]
  # set interval [expr $simulation_duration*1.0/300.0]
  # set interval 100
  set now [$ns now]
  puts "now is $now"
  for {set i 0} {$i < $real_http1_num} {incr i} {
    if {$laun_time_http1($i) <= $now} {
      set ack [$tcp_http1($i) set ack_]
        if {$ack < $fl_size_http1($i)} {
          set active_num [expr $active_num + 1]
          set flowlen [expr $fl_size_http1($i)]
          if {$flowlen <= $filelen_threshold} {
            set active_num_mice  [expr $active_num_mice + 1]
          }  else {
          set active_num_elephant [expr $active_num_elephant + 1]
          }
        }                                                             316
     }
  }

  puts $activef "$now [expr $active_num+$num_longbursts] \
    $active_num_mice [expr $active_num_elephant+$num_longbursts]"
  $ns at [expr $now + $interval] "active_conn_no"
}


proc finish {stoptime} {
  global random_run
  global qmon_  warmuptime bottleneck_bdw
  global QueueType
  global warmup_bthrput
  global warmup_bdepartures
  global warmup_barrivals
  global warmup_bdrops
  global total_bthrput
  global total_bdepartures
  global total_barrivals                                              336
  global total_bdrops
  global real_http1_num num_longbursts
  global laun_time_http1 laun_time_tcp_longbursts
  global sink_http1 sink_longbursts
  global tcp_http1 tcp_longbursts
  global monitorq_
  global fl_size_http1
  global activef
  global tchan_

  close $monitorq_
  close $activef

  set awkCode {
  {
    if ($2 >= warmuptime) {
      if ($1 == "Q" && NF>2) {
```

```
        print $2, $3 >> ("../results/" QueueType num_longbursts \
                        random_run ".queue");
      } else if ($1 == "l" && NF > 2) {                                    356
        print $2, $3 >> ("../results/" QueueType  num_longbursts \
                        random_run ".loadfactor");
      }
    }
}
            }

  if {$QueueType == "FLAQM" || $QueueType == "IFLAQM"} {
   exec rm -f ../results/$QueueType$num_longbursts$random_run.loadfactor
   exec touch ../results/$QueueType$num_longbursts$random_run.loadfactor
   exec rm -f ../results/$QueueType$num_longbursts$random_run.queue
   exec touch ../results/$QueueType$num_longbursts$random_run.queue
  } else {
   exec rm -f ../results/$QueueType$num_longbursts$random_run.queue
   exec touch ../results/$QueueType$num_longbursts$random_run.queue
  }
  if { [info exists tchan_] } {
   close $tchan_
  }
  exec awk $awkCode warmuptime=$warmuptime QueueType=$QueueType \          376
  num_longbursts=$num_longbursts random_run=$random_run \
  $QueueType$num_longbursts$random_run.q
  exec rm -f $QueueType$num_longbursts$random_run.q

  # total network thrput, traffic load, drop rate, link utilization
  for {set i 0} {$i < $real_http1_num} {incr i} {
   if {$laun_time_http1($i) < $stoptime} {
    set next [$sink_http1($i) set next_]
    if {$next > 1} {
      set size [$tcp_http1($i) set packetSize_]
      set total_bthrput [expr $total_bthrput+($next-1)*$size]
    }
   }
  }
  for {set i 0} {$i < $num_longbursts} {incr i} {
    if {$laun_time_tcp_longbursts($i) < $stoptime} {
      set next [$sink_longbursts($i) set next_]
       if {$next > 1} {
         set size [$tcp_longbursts($i) set packetSize_]
         set total_bthrput [expr $total_bthrput+($next-1)*$size]          396
       }
     }
  }

  set total_bdepartures [$qmon_ set bdepartures_]
  set total_barrivals [$qmon_ set barrivals_]
  set total_bdrops [$qmon_ set bdrops_]

############network measure calculation############################
  set network_measure [open \
  network_measure.$QueueType$num_longbursts$random_run w]
# network throughput
  puts $network_measure "Thrput: [expr ($total_bthrput-$warmup_\
    bthrput)*8.0/($stoptime-$warmuptime)/1000000.0/$bottleneck_bdw]"
# link utilization
  puts $network_measure "LinkUtilization: [expr ($total_bdepartures-\
   $warmup_bdepartures)*8.0/($stoptime-$warmuptime)/1000000.0/\
   $bottleneck_bdw]"
# traffic load
  puts $network_measure "TrafficLoad: [expr ($total_barrivals-\          416
  $warmup_barrivals)*8.0/($stoptime-$warmuptime)/1000000.0/\
  $bottleneck_bdw]"
#loss rate
  puts $network_measure "LossRate: [expr ($total_bdrops-$warmup_bdrops)\
  *1.0/($total_barrivals-$warmup_barrivals)]"

  close $network_measure
```

```
##########user measure calculation##########################
  # user goodput and response time
  set goodput_http1f [open \
  $QueueType$num_longbursts$random_run.goodput_http1 w]
  set responsetime_http1f [open \
  $QueueType$num_longbursts$random_run.responsetime_http1 w]

  for {set i 0} {$i < $real_http1_num} {incr i} {
   set next [$sink_http1($i) set next_]
   if {$next > $fl_size_http1($i)} {
     set last_sink_recv_time [$sink_http1($i) set last_sink_recv_time]
     set size [$tcp_http1($i) set packetSize_]
     puts $goodput_http1f "fileSize: $fl_size_http1($i) thrput(Kbps): \     436
     [expr ($next-1)*$size*8.0/1000.0/($last_sink_recv_time-\
       $laun_time_http1($i))]"
     puts $responsetime_http1f "fileSize: $fl_size_http1($i) \
    responsetime: [expr $last_sink_recv_time-$laun_time_http1($i)]"
   }
  }
  close $goodput_http1f
  close $responsetime_http1f

  exec rm -f monitorqueueEmpty
  exit 0
}

$ns at $warmuptime "active_conn_no"
$ns at $warmuptime "warmup_time warmuptime"
$ns at $simulation_duration "finish $simulation_duration"
$ns run
```