TCP CONGESTION CONTROL AND ACTIVE QUEUE MANAGEMENT MECHANISMS

by

Arsun Artel

B.S., Chemical Engineering, Boğaziçi University, 2001

Bogazici University Library
39001102309567
14

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2004

ACKNOWLEDGEMENTS

I am grateful to my thesis supervisor Assist. Prof. Murat Zeren for his guidance, patience and support throughout my studies. Without his kind efforts and friendly talks, it would be impossible for me to complete this work.

I also would like to thank Assist. Prof. Fatih Alagöz and Assoc. Prof. Kemal Cılız for participating in my jury and for their valuable time in reading my thesis and providing enlightening comments.

Special thanks are for my friends, especially Alber Amado, Itır Barutçuoğlu, Okan İrfanoğlu, Furkan Kıraç, Rabun Koşar and Özgün Özışıkyılmaz for their help and encouragement and for being such good friends. I hope that they will not make me miss our long talks and enjoyable and cheerful dinners.

More than thanks needed for Okan İrfanoğlu, Alihan Karagül and again for Furkan and Özgün for having provided challenge in table tennis when I searched for. Also thanks to Timo Boll, Jan Owe Waldner and Vladimir Samsonov for their inspiration.

How can I forget to mention Peter Sellers for having made such funny movies and again heartfelt thanks to Itır for giving me the chance to watch the whole Pink Panther Series. Itır, you are a great person, but so is Peter Sellers. R.I.P. the greatest chief inspector of all times!

And Mehmet İpekoğlu... He is and has always been like a brother to me. Thank you very much for being there whenever and wherever I needed.

Finally, my deepest gratitude is to my family. Without their existence, nothing would be possible.

ABSTRACT

TCP CONGESTION CONTROL AND ACTIVE QUEUE MANAGEMENT MECHANISMS

As the Internet continues to expand in size and diversity, congestion control mechanisms start to represent a more and more important role in meeting the expectations of different applications.

Although TCP (Transmission Control Protocol) has its own end-to-end congestion control mechanisms, which act only in a reactive manner, in that congestion control is done after the network is overloaded, IETF (Internet Engineering Task Force) is thinking to deploy additional active queue management mechanisms executed by routers, which can be proactive and drop packets before congestion occurs and thus notify the sources of the incipient congestion.

In this study, two different AQM (Active Queue Management) mechanisms and guidelines for their parameter selection are proposed. Both mechanisms are based on fuzzy control theory. The first one is a direct fuzzy controller that performs better than most of the AQM mechanisms proposed in the last few years. The second mechanism is a FMRLC (Fuzzy Model Reference Learning Controller), which adapts to different conditions in the network and shows even better results than the direct one. The simulation scenarios vary from FTP (File Transfer Protocol) connections only to wireless cases and performance metrics investigated for the comparison are the instantaneous queue length, bottleneck link utilization, dropped packets, number of connections and fairness.

ÖZET

TCP MEKANİZMALARINDA SIKIŞIKLIK DENETİMİ VE AKTİF KUYRUK YÖNETİM MEKANİZMALARI

Günümüzde İnternet, büyümeğe ve uygulama çeşitliliği olarak gelişmeğe devam ettikçe, sıkışıklık denetim mekanizmalarının oynadığı rol, bu farklı uygulamaların beklentilerini karşılamak için gitgide daha önem kazandı.

TCP (İletim Denetimi Protokolu), uç düğümler arasında çalışan ve ancak ağ aşırı derecede yüklendikten sonra tepkin, kendi sıkışıklık denetim mekanizmasına sahip olmasına rağmen, IETF (İnternet Mühendisliği Çalışma Kolu) bu mekanizmalara ilave olarak, sıkışıklık olmadan paketleri düşürerek, kaynaklara yeni başlayan sıkışıklığı önceden tepkin davranarak haber verecek ve yönlendiricilerde yürütülecek aktif kuyruk yönetim mekanizmalarını uygulamaya koymayı düşünmektedir.

Bu çalışmada, iki farklı AQM (Aktif Kuyruk Yönetimi) mekanizması ve bunların parametrelerinin seçimi konusunun ana hatları önerilmiştir. İki mekanizma da bulanık kontrol kuramına dayanmaktadır. İlk önerilen uyarlanır olmayan bulanık denetleyici, son senelerde önerilen pek çok AQM mekanizmalarından daha iyi performans göstermektedir. FMRLC (Bulanık Modele Dayanan Öğrenen Denetleyici) önerilen ikinci mekanizma olup, ağdaki farklı durumlara uyum göstermekte ve ilkine göre daha iyi sonuçlar almaktadır. Benzetim senaryoları sadece FTP (Dosya Aktarım Protokolu) bağlantılarından, kablosuz olanlara kadar değişmektedir. Karşılaştırma için incelenen performans ölçütleri, anlık kuyruk boyu, darboğaz bağının kullanımı, düşürülen paketler, bağlantıların sayısı ve adalettir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xx
LIST OF SYMBOLS/ABBREVIATIONS . . . xxi
1. INTRODUCTION . . . 1
1.1. Outline of the Thesis . . . 3
2. THEORETICAL BACKGROUND AND AQM MECHANISMS . . . 4
2.1. TCP Congestion Control . . . 5
2.1.1. Slow Start and Congestion Avoidance . . . 5
2.1.2. Fast Retransmit/Fast Recovery . . . 8
2.2. TCP Versions and Congestion Control Mechanisms . . . 9
2.3. Flow Control and Active Queue Management (AQM) . . . 10
2.3.1. RED Algorithm . . . 14
2.3.2. Gentle Variant of RED . . . 15
2.3.3. Explicit Feedback Mechanisms . . . 16
2.3.3.1. ECN - Explicit Congestion Notification . . . 16
2.3.3.2. Other Works for Explicit Feedback . . . 17
2.3.4. ARED - Adaptive RED . . . 18
2.3.5. BLUE - Another Congestion Control Approach . . . 20
2.3.6. REM - Random Exponential Marking . . . 21
2.3.7. PI Controller - Congestion Control Based on an Analytical Model 23
3. FUZZY CONTROL AND PARAMETER TUNING . . . 25
3.1. Theoretical Background of Fuzzy Control . . . 25
3.2. Fuzzy Sets, Fuzzy Logic and Rule Base . . . 27
3.2.1. Membership Functions . . . 27
3.2.2. Fuzzy Logic . . . 28
3.3. Operation of the Fuzzy Controller . . . 28

4. MOTIVATION, DETAILED VIEW ON THE DESIGNS AND PARAMETER TUNING . . . 31
4.1. Some Design Issues for DFC and AFC . . . 32
4.1.1. Output Membership Function Tuning . . . 32
4.1.2. Output Scaling Gain . . . 35
4.1.3. Design Issues Specific to Adaptive Fuzzy Controller . . . 35
4.2. Simulation Environment and Scenarios . . . 37
4.2.1. Simulation Environment . . . 38
4.2.2. Simulation Scenarios . . . 39
4.3. Search for Good Parameters for DFC and AFC . . . 40
4.3.1. Choosing Parameters for Direct Fuzzy Controller . . . 41
4.3.2. Choosing Parameters for Adaptive Fuzzy Controller . . . 43
5. COMPARISON RESULTS OF DIFFERENT AQM MECHANISMS . . . 45
5.1. Overview of the Simulation Results . . . 47
5.1.1. Change in Bottleneck Delay . . . 47
5.1.2. Change in the Buffer Size . . . 49
5.1.3. Change in the Number of Nodes . . . 51
5.1.4. Wireless Scenarios . . . 52
5.1.5. Effect of the ECN Bit . . . 52
5.1.6. Common Results in Different Scenarios . . . 52
5.2. Detailed Examination of the Results Obtained . . . 53
5.2.1. Scenarios with FTP Traffic Only and Bottleneck Delay of 5 ms . 54
5.2.2. Scenarios with Added Traffic and Bottleneck Delay of 5 ms . . . 68
5.2.3. Scenarios with Changed Bottleneck Delay of 125 ms . . . 86
5.2.4. Scenarios with Changed Number of Nodes . . . 95
5.2.5. Scenarios with Changed Buffer Size . . . 108
5.2.6. Scenarios with Wireless Cases . . . 130
6. CONCLUSIONS AND FUTURE WORK . . . 139
REFERENCES . . . 141

LIST OF FIGURES

Figure 1.1. Window flow control 2

Figure 2.1. Slow start - time flow 6

Figure 2.2. Slow start - window size diagram 6

Figure 2.3. Congestion avoidance - time flow 7

Figure 2.4. Congestion avoidance - window size diagram 7

Figure 2.5. TCP Tahoe overview 10

Figure 2.6. TCP Tahoe algorithm 11

Figure 2.7. TCP Reno overview 11

Figure 2.8. TCP Reno algorithm 11

Figure 2.9. TCP Vegas overview 12

Figure 2.10. AQM mechanism with TCP 12

Figure 2.11. RED operation 14

Figure 2.12. Gentle variant of RED 16

Figure 2.13. Floyd's version of ARED algorithm 19

Figure 2.14. BLUE algorithm 20

Figure 2.15. Marking probability of gentle variant of RED and REM 22

Figure 3.1. Fuzzy controller architecture 26

Figure 3.2. A simple membership example 27

Figure 3.3. Degree of membership determination 29

Figure 3.4. Calculation of the fuzzy output 29

Figure 3.5. COG defuzzification using min-max inferencing 30

Figure 4.1. Membership functions of the both direct and adaptive fuzzy controller 33

Figure 4.2. Output membership function of the direct fuzzy controller 33

Figure 4.3. Decision surface of the direct fuzzy controller 34

Figure 4.4. FMRLC - adaptive fuzzy controller 36

Figure 4.5. Simplified user's view of NS 38

Figure 4.6. Basis for the simulation scenario 40

Figure 4.7. Direct fuzzy controller - $power = 2$; $gain = 0.4$ 41

Figure 4.8. Direct fuzzy controller - $power = 2.2$; $gain = 0.3$ 42

Figure 4.9. Direct fuzzy controller - $power = 2.2$; $gain = 0.4$ 42

Figure 4.10. Adaptive fuzzy controller - $cautious = 3$; $gain = 0.5$ 43

Figure 4.11. Adaptive fuzzy controller - $cautious = 3$; $gain = 0.4$ 44

Figure 4.12. Adaptive fuzzy controller - $cautious = 4$; $gain = 0.4$ 44

Figure 5.1. Jain's fairness index 45

Figure 5.2. Comparison of instantaneous queue lengths for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms . 56

Figure 5.3. Comparison of bottleneck link utilization for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms 57

Figure 5.4. Comparison of dropped packets/sec for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms 58

Figure 5.5. Comparison of average individual flow delays for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms . 59

Figure 5.6. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 60

Figure 5.7. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 61

Figure 5.8. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 62

Figure 5.9. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 63

Figure 5.10. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set 64

Figure 5.11. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set 65

Figure 5.12. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set 66

Figure 5.13. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set 67

Figure 5.14. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 69

Figure 5.15. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 70

Figure 5.16. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 71

Figure 5.17. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 72

Figure 5.18. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 73

Figure 5.19. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 74

Figure 5.20. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 75

Figure 5.21. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 76

Figure 5.22. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 78

Figure 5.23. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 79

Figure 5.24. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 80

Figure 5.25. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 81

Figure 5.26. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set 82

Figure 5.27. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set 83

Figure 5.28. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set 84

Figure 5.29. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set 85

Figure 5.30. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 87

Figure 5.31. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 88

Figure 5.32. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 89

Figure 5.33. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 90

Figure 5.34. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 91

Figure 5.35. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 92

Figure 5.36. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 93

Figure 5.37. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set 94

Figure 5.38. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 96

Figure 5.39. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 97

Figure 5.40. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 98

Figure 5.41. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 99

Figure 5.42. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 100

Figure 5.43. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 101

Figure 5.44. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 102

Figure 5.45. Comparison of average individual flow delays for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic 103

Figure 5.46. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 104

Figure 5.47. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 105

Figure 5.48. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 106

Figure 5.49. Comparison of average individual flow delays for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic 107

Figure 5.50. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 109

Figure 5.51. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 110

Figure 5.52. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 111

Figure 5.53. Comparison of average individual flow delays for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic 112

Figure 5.54. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 113

Figure 5.55. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic114

Figure 5.56. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic115

Figure 5.57. Comparison of average individual flow delays for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 116

Figure 5.58. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 118

Figure 5.59. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic . . . 119

Figure 5.60. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic . . . 120

Figure 5.61. Comparison of average individual flow delays for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic . . . 121

Figure 5.62. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic . . . 122

Figure 5.63. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic 123

Figure 5.64. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic . 124

Figure 5.65. Comparison of average individual flow delays for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic . . . 125

Figure 5.66. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic . . . 126

Figure 5.67. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 127

Figure 5.68. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 128

Figure 5.69. Comparison of average individual flow delays for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic 129

Figure 5.70. Comparison of instantaneous queue lengths for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms . . 131

Figure 5.71. Comparison of bottleneck link utilization for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms . . 132

Figure 5.72. Comparison of dropped packets/sec for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms 133

Figure 5.73. Comparison of average individual flow delays for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms 134

Figure 5.74. Comparison of instantaneous queue lengths for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms . . 135

Figure 5.75. Comparison of bottleneck link utilization for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms . . 136

Figure 5.76. Comparison of dropped packets/sec for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms 137

Figure 5.77. Comparison of average individual flow delays for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms 138

LIST OF TABLES

Table 3.1. Comparison of conventional logic and fuzzy logic 28

Table 4.1. Fuzzy rule base 35

Table 5.1. Values of parameters used in different algorithms 46

Table 5.2. Values of parameters in the simulation scenarios 48

Table 5.3. Jain's fairness index for scenarios, where $buffer = 150$ 49

Table 5.4. Total packet drops for scenarios, where $buffer = 150$ 49

Table 5.5. Jain's fairness index for scenarios, where $buffer = 200$ 50

Table 5.6. Total packet drops for scenarios, where $buffer = 200$ 51

LIST OF SYMBOLS/ABBREVIATIONS

$\dot{A}$	Time derivative of the variable A
a, b	Optimized parameters to PI controller
avg_{queue}	Average queue length
C	Link capacity
$cautious$	Update parameter of output centers in AFC
c^i	Location of the input or output center i in fuzzy control
$d1$, $d2$	Update parameters for p_m in BLUE
e	Error
$edot$	Change in error
F	Fairness
$freeze_time$	Minimum time between two successive updates of p_m in BLUE
$gain$	Output gain for fuzzy controllers
$interval$	Constant time interval of 0.5 seconds in ARED
L	Threshold for queue length used by BLUE
max_i	Number of triangles for the output membership function
max_p	Drop probability parameter in RED
max_{thresh}	Maximum threshold for average queue length in RED
min_{thresh}	Minimum threshold for average queue length in RED
N	Number of connections
$p(kT)$	Drop probability at k^{th} sampling interval
p_m	Mark probability used by BLUE
$power$	Design parameter for output centers in fuzzy control
p_{REM}	Congestion measure in REM
$prob$	Drop probability in REM
q	Instantaneous sample of the queue length
Q_{len}	Length of the queue used by BLUE
q_{ref}	Reference value for queue length
r	Reference input to the fuzzy controller
R	Average RTT in analytical model for TCP

T	$1/samplingfrequency$
$target$	Target for avg_{queue}
T_p	Propagation delay
u	Plant input
W	Window size
w_q	Weight for computing a target average queue length in ARED
x	Packet arrival rate
y	Plant output
y_m	Reference model output
α	Increase factor in ARED
α_{REM}	A positive small constant in REM
β	Constant decrease factor in ARED, which equals to 0.9
γ	A positive small constant in REM
μ_A	Membership function associated with fuzzy set A
ϕ	A positive constant greater than one in REM
Ω	Universe of discourse
ACK	Acknowledgement
AFC	Adaptive Fuzzy Controller
AIMD	Additive increase multiplicative decrease
AQM	Active Queue Management
ARED	Adaptive RED
ATM	Asynchronous Transfer Mode
BLUE	Another AQM algorithm
CBR	Constant bit rate
cwnd	Congestion window
DFC	Direct Fuzzy Controller
ECN	Explicit Congestion Notification
EWA	Explicit Window Adaptation
FIFO	First In First Out

FMRLC	Fuzzy Model Reference Learning Control
FR/FR	Fast Retransmit/Fast Recovery
IETF	Internet Engineering Task Force
IP	Internet Protocol
LAN	Local area network
MAC	Medium Access Control
MEO	middle Earth orbit
MIMD	Multiplicative increase multiplicative decrease
MSS	Maximum segment size
NS	Network Simulator
OTcl	Object oriented Tcl
PI	Proportional Integral
RED	Random Early Detection - or Drop
REM	Random Exponential Marking
RFC	Request for comment
RTT	Round Trip Time
rwnd	Receiver window
SMTP	Simple Mail Transfer Protocol
SR	Source rate
ssthresh	Slow start threshold
Tcl	A common script language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VBR	Variable bit rate
VINT	Virtual InterNetwork Testbed

1. INTRODUCTION

Internet congestion occurs when the overall demand for a resource (e.g., link bandwidth) exceeds the available capacity of the resource. The effects resulting from such congestion can be very drastic such as long delays in data delivery, wasted resources due to lost or dropped packets, and even possible congestion collapse, in which all communication in the entire network stops. It is therefore clear that in order to maintain good network performance, certain mechanisms must be provided to prevent the network from being congested for any significant period time. Those mechanisms are called as congestion control mechanisms. Congestion control involves the design of mechanisms and algorithms to statistically limit the demand-capacity mismatch, dynamically control traffic sources when such a mismatch occurs. It has been shown that static solutions such as allocating more buffers, providing faster links or faster processors are not effective for congestion control purposes. Current usage of the Internet is dominated by TCP (Transmission Control Protocol) traffic such as remote terminal, FTP (File Transfer Protocol), Web traffic, and electronic mail (e.g., SMTP - Simple Mail Transfer Protocol) [1].

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks. TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols, which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks [2].

TCP congestion and its control is one of the most significant problems in today's Internet applications. TCP uses a window flow control schema using Acknowledgments (ACKs) for the packets and uses them to find out whether there is congestion or not. The ACK procedure between source and the destination is shown in the Figure 1.1. Basically if a packet reaches its destination, its ACK is sent to the source.

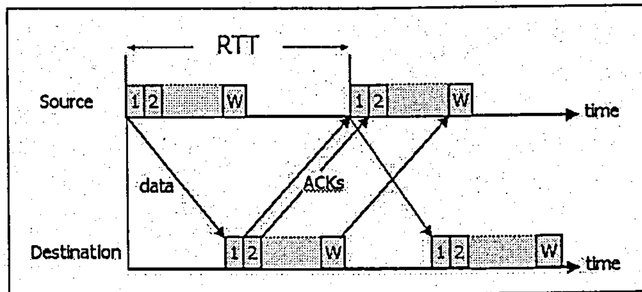


Figure 1.1. Window flow control

The data-sending rate of TCP (or the window size) is determined by the rate of incoming ACKs to previous packets. The rate of ACK arrival is in turn determined by the presence or absence of congested link(s) along the path between a source and its destination. In steady state, the sources sending rate will match the arrival rate of the ACKs. Accordingly, TCP automatically detects congestion and regulates its sending rate. This has been referred to as TCPs self-clocking behavior. Three major TCP implementations are:

- TCP Tahoe (Slow Start, Congestion Avoidance, Fast Retransmit).
- TCP Reno (Fast Recovery).
- TCP Vegas.

However, current Internet congestion control methods are expected to result in unsatisfactory performance, e.g., multiple packet losses and low link utilization, as the number of users and the size of the network increases. Accordingly, many congestion control approaches have been proposed. Network algorithms such as Active Queue Management (AQM), executed by network components such as routers, detect network congestion, packet losses, or incipient congestion, and inform traffic sources (either implicitly or explicitly). In response, source algorithms adjust the sources data-sending rate into the network. The basic design issues are what to feedback (network algorithms) and how to react (source algorithms) [1].

TCP Tahoe, TCP Reno, and TCP Vegas detect congestion only after a buffer at an intermediate router has already overloaded and packets have been lost. An AQM

schema aims to drop packets before the queue becomes full. This in turn enables the source to respond to congestion before buffers overflow. Hence the congestion can be detected before it becomes a problem and thus the network can be kept stable.

TCP/AQM dynamics can be designed and analyzed by means of feedback control modelling. The control theoretic approach can increase the speed of response and improves stability and robustness of the congestion control [1].

This work focuses on the AQM mechanisms from a control theoretic point of view and contributes with two different AQM schemes, direct fuzzy controller and adaptive fuzzy controller.

1.1. Outline of the Thesis

Section 1 stated the problem of congestion control, a summary of the approaches of TCP to the problem and the overview of the AQM mechanisms. The organization of the rest of the thesis is as follows: In Section 2, theoretical background of different TCP mechanisms and AQM approaches are given. Section 3 addresses the theoretical background of a fuzzy controller. The methodology used and the contributions made along with their implementation issues are provided in Section 4. Section 4 also explains the simulation environment and the basic scenario. Section 5 focuses mainly on the simulation scenarios in detail, their results and comparisons. Section 6 concludes the thesis by summarizing the contributions and discussing issues for future work.

2. THEORETICAL BACKGROUND AND AQM MECHANISMS

In October 1986, Internet has its first congestion collapse. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two drops) dropped from 32 Kbps to 40 bps, which was actually a factor of nearly 1000 drop. Hence there arose a question on how to tune TCP to provide better working conditions under this type of network conditions [3]. The mechanisms he proposed are the basics of today's TCP Tahoe implementation.

In window flow control, there are W packets per RTT (Round Trip Time) and the loss detection mechanism is based on ACKs. If the number of packets in the network are limited to window W, then the source rate SR becomes:

$$SR = \frac{W \times MSS}{RTT} \quad bps \tag{2.1}$$

where MSS is the Maximum Segment Size to send.

If W is too small, then the rate is much less than the capacity and if W is too big, then the rate becomes larger than the capacity which in turn yields to congestion.

The effects of congestion are various. There are packet losses and retransmissions due to those packet losses. Hence we are faced with a reduced throughput. There can also be congestion collapse due to unnecessarily retransmitted, undelivered or unusable packets. What is even worse is that the congestion may continue after the overload.

TCP mechanisms are seeking to achieve high utilization, avoid congestion and share of available bandwidth. TCP tries to adapt the window size to the network and dynamic conditions.

In TCP window control approaches, there are two types flow control, receiver flow control, which is set by the receiver, avoids overloading on the receiver side and based on (advertised) receiver window *rwnd* and network flow control, which is set by the sender, avoids overload in the network and based on congestion window *cwnd*. Then W is calculated to carefully use available network capacity as follows:

$$W = \min(cwnd, rwnd) \tag{2.2}$$

The sources calculate *cwnd* from the congestion indications. Those indications can be losses, delays and marks. The algorithms to calculate congestion window are TCP Tahoe, Reno, Vegas and their variations.

2.1. TCP Congestion Control

RFC 2581 [4] and RFC 2001 [5] specify four TCP congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery. Following sections are devoted to explain the above mentioned phases.

2.1.1. Slow Start and Congestion Avoidance

The minimum of *cwnd* and *rwnd* governs data transmission. Another state variable, the slow start threshold, *ssthresh*, is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission, as discussed below.

The slow start algorithm is used for avoiding congestion at the beginning of a transfer, or after repairing loss detected by the retransmission timer.

The initial value of *ssthresh* may be arbitrarily high, but it may be reduced in response to congestion. The slow start algorithm is used when $cwnd < ssthresh$, while the congestion avoidance algorithm is used when $cwnd > ssthresh$. When *cwnd* and *ssthresh* are equal the sender may use either slow start or congestion avoidance.

During slow start, a TCP increments *cwnd* by at most *MSS* bytes for each ACK received that acknowledges new data. Slow start ends when *cwnd* exceeds *ssthresh* (or, optionally, when it reaches it, as noted above) or when congestion is observed. The mechanism of slow start is shown in Figure 2.1 and Figure 2.2. Figure 2.1 shows the time flow of slow start, whereas Figure 2.2 shows its window size diagram. Slow start

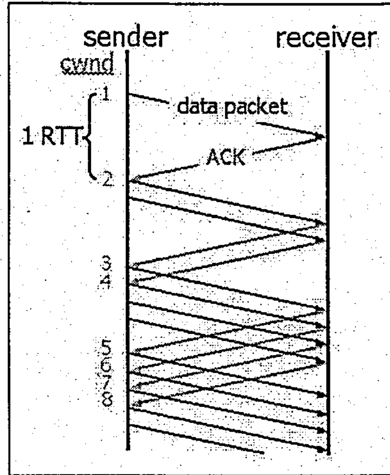


Figure 2.1. Slow start - time flow

can significantly increase latency when the object size is relatively small and the *RTT* is relatively large. Unfortunately, this is often the scenario when sending of objects over the World Wide Web.

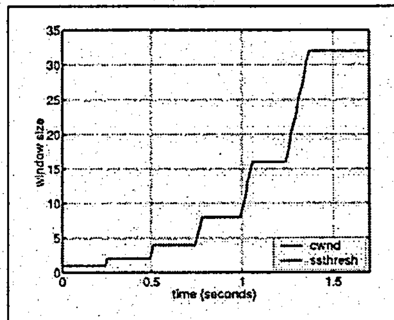


Figure 2.2. Slow start - window size diagram

During congestion avoidance, *cwnd* is incremented by one full-sized segment per round trip time *RTT*. Congestion avoidance continues until congestion is detected.

When a TCP sender detects segment loss using the retransmission timer, the value of *ssthresh* must be set to no more than the value given in Equation 2.3:

$$ssthresh = \max\left(\frac{W}{2}, 2 \times MSS\right) \tag{2.3}$$

The mechanism of congestion avoidance is shown in Figure 2.3 and Figure 2.4. Figure 2.3 shows the time flow of congestion avoidance, whereas Figure 2.4 shows its

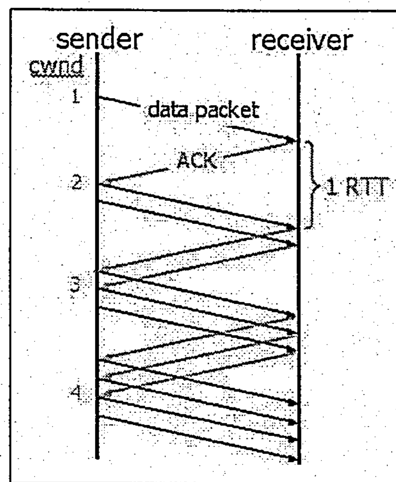


Figure 2.3. Congestion avoidance - time flow

window size diagram. If we ignore the slow start phase, we see that TCP essentially

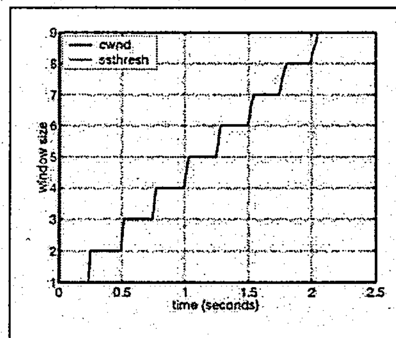


Figure 2.4. Congestion avoidance - window size diagram

increases its window size by one each RTT (and thus increases its transmission rate by an additive factor) when its network path is not congested, and decreases its window

size by a factor of two each RTT when the path is congested. For this reason, TCP is often referred to as an additive-increase, multiplicative-decrease (AIMD) algorithm.

2.1.2. Fast Retransmit/Fast Recovery

A TCP receiver should send an immediate duplicate ACK when an out-of-order segment arrives. The purpose of this ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected. From the sender's perspective, duplicate ACKs can be caused by a number of network problems. First, they can be caused by dropped segments. In this case, all segments after the dropped segment will trigger duplicate ACKs. Second, duplicate ACKs can be caused by the re-ordering of data segments by the network. Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network. In addition, a TCP receiver should send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space. This will generate more timely information for a sender recovering from a loss through a retransmission timeout or a fast retransmit.

The TCP sender should use the "fast retransmit" algorithm to detect and repair loss, based on incoming duplicate ACKs. The fast retransmit algorithm uses the arrival of three duplicate ACKs as an indication that a segment has been lost. After receiving three duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives. The reason for not performing slow start is that the receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network. In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources.

The fast retransmit and fast recovery algorithms are usually implemented to-

gether as follows:

- When the third duplicate ACK is received, set *ssthresh* to no more than the value given in Equation 2.3.
- Retransmit the lost segment and change the value of *cwnd* using Equation 2.4.

$$cwnd = ssthresh + 3 \times MSS \tag{2.4}$$

  This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
- For each additional duplicate ACK received, increment *cwnd* by MSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
- Transmit a segment, if allowed by the new value of *cwnd* and the receiver's advertised window.
- When the next ACK arrives that acknowledges new data, set *cwnd* to *ssthresh* (the value set in the first step). This is termed "deflating" the window.

This new ACK acknowledging new data, should be the acknowledgment elicited by the retransmission from the first step, one RTT after the retransmission (though it may arrive sooner in the presence of significant out-of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost [4].

2.2. TCP Versions and Congestion Control Mechanisms

Basically there are three TCP implementations. TCP Tahoe includes slow start, congestion avoidance and fast retransmit. TCP Reno has, in addition to all mechanisms in Tahoe, also a fast recovery mechanism. The fast recovery mechanism essentially cancels the slow start phase after a fast retransmission. Most TCP implementations currently use the Reno algorithm. There is, however, another algorithm in the litera-

ture, the Vegas algorithm that can improve Reno's performance. Whereas Tahoe and Reno react to congestion (i.e., to overflowing router buffers), Vegas attempts to avoid congestion while maintaining good throughput. The basic idea of Vegas is to (1) detect congestion in the routers between source and destination before packet loss occurs, and (2) lower the rate linearly when this imminent packet loss is detected. Imminent packet loss is predicted by observing the round trip times - the longer the round trip times of the packets, the greater the congestion in the routers. Vegas is currently not a part of the most popular TCP implementations.

The time flow of TCP Tahoe is shown in Figure 2.5. The algorithm of TCP

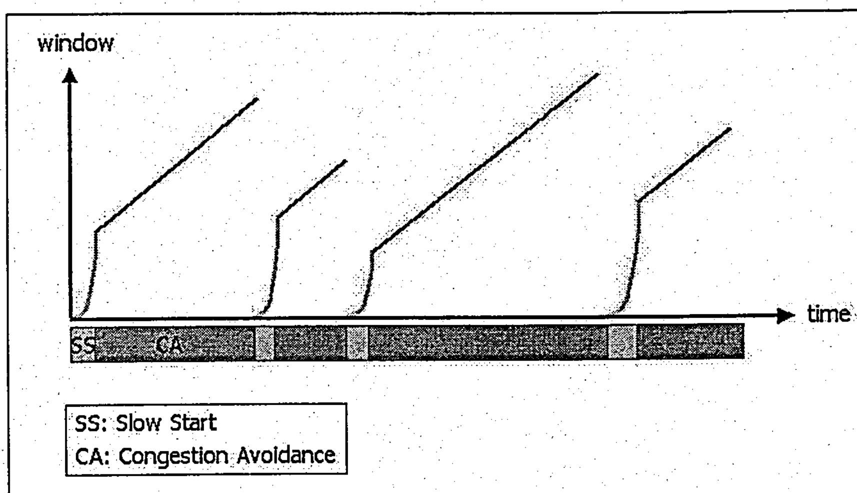


Figure 2.5. TCP Tahoe overview

Tahoe is shown in Figure 2.6. The time flow of TCP Reno is shown in Figure 2.7. The overview of TCP Reno's algorithm is shown in Figure 2.8. The time flow of TCP Vegas is shown in Figure 2.9.

2.3. Flow Control and Active Queue Management (AQM)

Flow control is a distributed algorithm to share network resources among competing users. It usually consists of two sub algorithms.

The link algorithm is executed in network devices such as routers and switches,

- For every ACK:
  - if ($W < ssthresh$) then $W = W + 1$ (Slow Start)
  - else W += $1/W$ (Congestion Avoidance)
- For every loss:
  - $ssthresh = W/2$
  - $W = 1$

Figure 2.6. TCP Tahoe algorithm

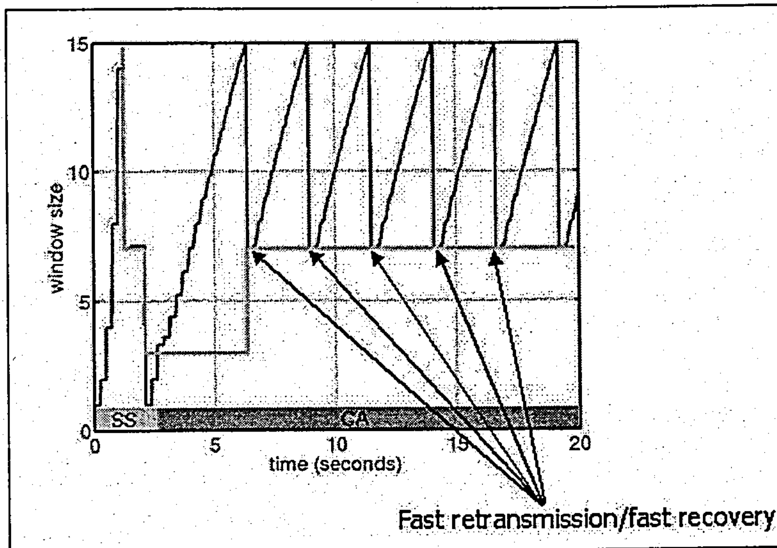


Figure 2.7. TCP Reno overview

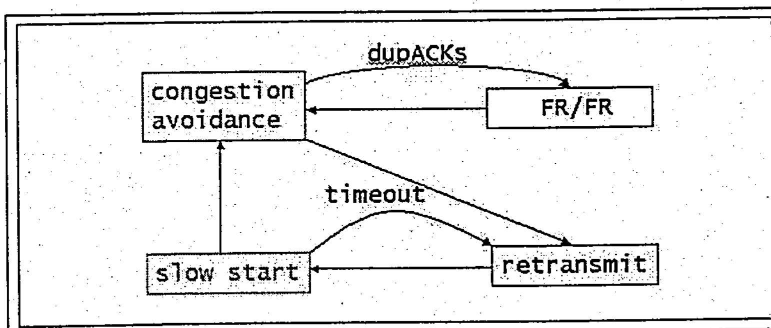


Figure 2.8. TCP Reno algorithm

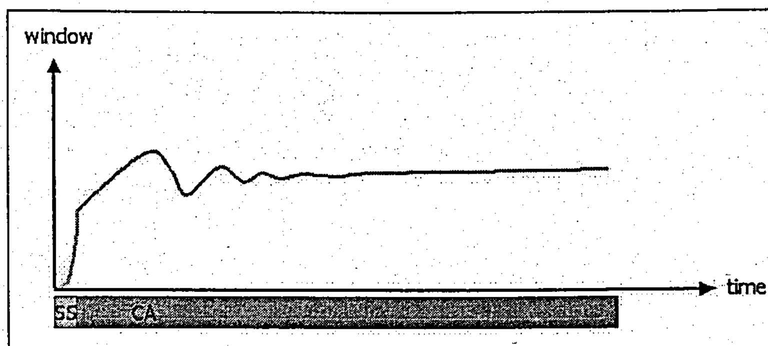


Figure 2.9. TCP Vegas overview

and the source algorithm is executed in edge devices such as host computers or edge routers.

The link algorithm detects congestion and feeds back information to sources, and the source algorithm, in response, adjusts the rate at which user traffic is injected into the network. Basically the link algorithm gives what to feed back and the source algorithm gives how to react.

Examples of source algorithms on the Internet are TCP Tahoe, Reno and Vegas, which adjust the window size in response to congestion signals. An example of a link algorithm is RED (Random Early Detection - or Drop), which drops or marks packets probabilistically in times of congestion. There are also other implementations such as BLUE, REM (Random Exponential Marking), PI (Proportional Integral), etc. The dashed line in Figure 2.10 shows the feed back coming from the link algorithm to the source algorithm.

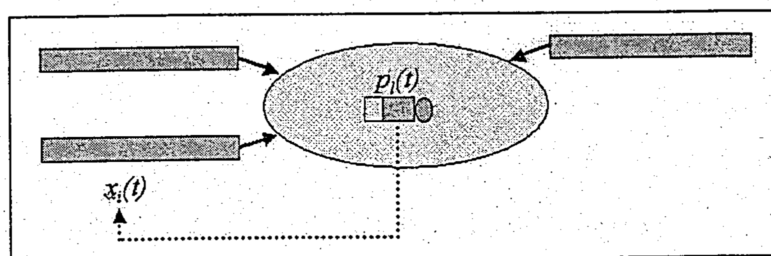


Figure 2.10. AQM mechanism with TCP

In the literature, AQM refers to link algorithms and their implementations with the source algorithms. AQM consists of a queue management algorithm, which manages the length of the packet queues by dropping packets when necessary or appropriate.

In the best-effort Internet traffic, the simplest approach, which has served Internet well for years, is drop-tail (also referred as tail-drop) with a FIFO (First In First Out) queue. Drop-tail refers to the action of dropping a packet that was intended to be added at the tail of a full FIFO queue. Drop-tail is an on-off control strategy. It is known in control theory that such an on-off mechanism leads to oscillations (limit-cycles) that can exhibit complex and chaotic behavior. If we look at a detailed level, this approach has mainly two drawbacks:

- Lock-out phenomenon: In some situations, drop-tail allows a single connection or a few flows to monopolize queue space, preventing other connections getting room in the queue.
- Full-queues: The drop-tail discipline allows queues to maintain a full (or, almost full) status for long periods of time, since drop-tail signals congestion (via a packet drop) only when the queue has become full. If the queue is full or almost full, an arriving burst will cause multiple packets to be dropped. This can result in a global synchronization of flows throttling back, followed by a sustained period of lowered link utilization, reducing overall throughput.

Besides drop-tail, there are also two alternative queue disciplines that can be applied when the queue becomes full are "random drop on full" or "drop front on full". Both of these solve the lock-out problem, but neither solves the full-queues problem.

Hence, one of the biggest problems with TCPs congestion control algorithm over drop-tail queues is that the sources reduce their transmission rates only after detecting packet loss due to queue overflow. Also since considerable amount of time may elapse between the packet drop at the router and its detection at the source, a large number of packets may be dropped as the senders continue transmission at a rate that the network cannot support. AQM algorithms, such as RED, try to overcome this phenomenon by

detecting incipient congestion early and delivering congestion notification to the end-hosts, allowing them to reduce their transmission rates before queue overflow occurs. Thus, routers operating with an AQM algorithm drop packets before their buffers become full and solve the so-called full-queues problem.

2.3.1. RED Algorithm

With the motivation of overcoming of lock-out phenomenon and full-queues problems and stabilizing oscillations in the queue size, a proactive approach, the RED algorithm, was introduced [6]. The marking (dropping) probability of RED algorithm is shown in Figure 2.11 and it has basically the following steps:

- Maintain running average of queue length avg_{queue}.
- If $avg_{queue} < min_{thresh}$ do nothing, which means that there is low queuing, we can send packets through.
- If $avg_{queue} > max_{thresh}$, drop packet, which means that we have to drop packets for protecting from misbehaving sources.
- Else mark packet in a manner proportional to queue length in order to notify sources of developing congestion.

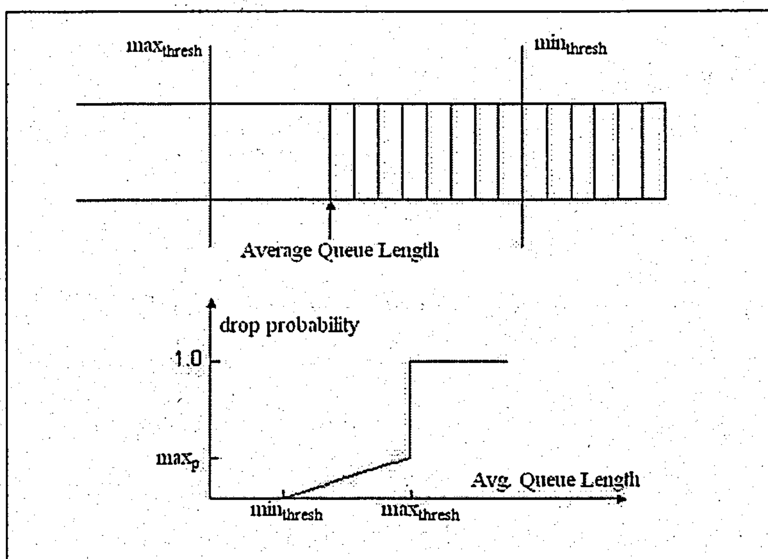


Figure 2.11. RED operation

In the literature, it has been shown that RED has some limitations [7]. A limitation of RED is the direct coupling between queue length and loss feedback. This results in load dependent queue levels. Higher load levels results in lower bandwidth per flow. RED also shows a bad response if the parameters are not well chosen and the link utilization is above 90 per cent [8]. Since RED introduces a range of reference input values, i.e., any queue length between two thresholds min_{thresh} and max_{thresh} rather than a constant reference input for control, the TCP/RED model shows oscillatory system dynamics. As a result, RED gives poor performance under a wide range of traffic environments. Furthermore, RED also shows sluggish response to the traffic dynamics [1]. The stable designs are slow and more responsive, faster designs may come with a price of fluctuating loss levels and inefficient utilization of resources.

Low *et al.* presented the form of TCP/RED's stability region. They also state that TCP/RED becomes unstable when the network scales up in delay or even link capacity. The main point of this analysis is that it indicates the difficulty of setting RED parameters to stabilize TCP. The analysis demonstrates that the parameters can be tuned to improve stability, but only at the cost of large queues even when they are dynamically adjusted [9].

2.3.2. Gentle Variant of RED

In the literature, there are different variants of RED, one of them being the "gentle" one. This variant is the basis for the ARED implementations of Floyd *et al.* [10].

The "gentle mode" is basically a modification to the original design in which the mark or drop probability increases linearly between max_p and one, as the average queue length varies between max_{thresh} and two times max_{thresh} [11, 12]. This fixes a problem in the original RED design caused by the non-linearity in drop probability (increasing from max_p to one immediately when max_{thresh} is reached). Figure 2.12 shows the marking probability of this variant [13].

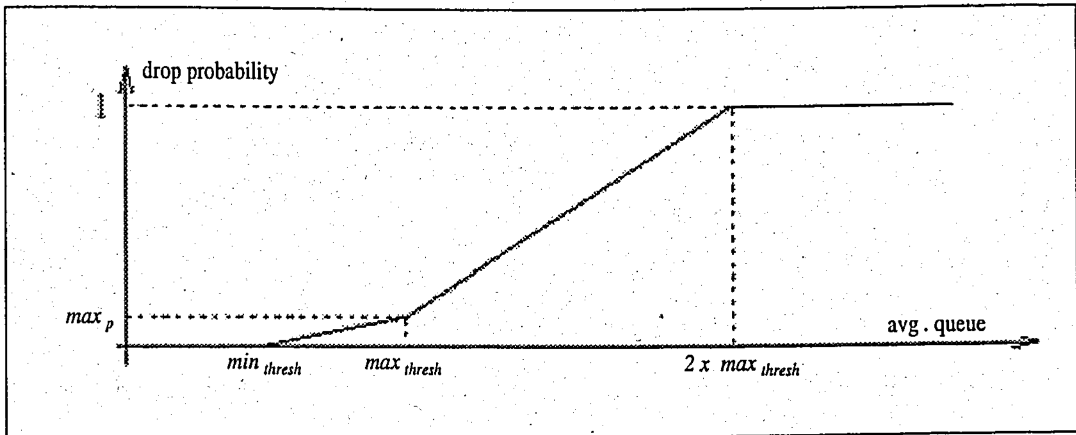


Figure 2.12. Gentle variant of RED

2.3.3. Explicit Feedback Mechanisms

In the literature, there are also proposals, which suggest to deploy explicit feedback mechanisms to assist AQM mechanisms, such as RED, which is known to have some drawbacks as explained above.

2.3.3.1. ECN - Explicit Congestion Notification. The most important one of the explicit feedback mechanisms is the use of ECN. S. Floyd discusses the use of Explicit Congestion Notification (ECN) mechanisms and explores, using simulations, the benefits and drawbacks of ECN in TCP/IP (Internet Protocol) networks. Those simulations use RED gateways modified to set an ECN bit in the IP packet header, with TCP Reno modified to respond to ECN as well as to packet drops as indications of congestion. For the simulations, the RED gateways were given an option to set the ECN bit in the packet header, rather than dropping the packet, as an indication of congestion when the buffer had not yet overflowed. When the TCP receiver receives a data packet with the ECN bit set in the packet header, it sets the ECN bit in the next outgoing ACK packet. TCP should react to an ECN at most once per round trip time in order to avoid overreacting. Because of the same principle, if the TCP source receives triple duplicate ACKs (in the same round trip time), indicating that a packet is dropped, it should not repeat the reduction in the congestion window also [14].

In terms of congestion control, TCP connections respond to a single ECN mark as they would to a single packet loss. One of the key advantages of ECN will not be for TCP traffic, but instead for traffic such as real-time or interactive traffic, where the cost of an unnecessary packet drop is either the unnecessary delay of retransmitting the packet, or possibly continuing without that packet altogether. TCP congestion control dynamics with ECN are similar to those without ECN. The main difference is that the TCP sender does not have to retransmit the marked packet (as it would if the packet had been dropped). For example, ECN would mean shorter transfer times for the small number of short flows that might otherwise have the final packet of a transfer dropped. It has been shown that there are some performance advantages of ECN for TCP short transfers. One of the advantages of ECN is that, by replacing a packet drop by a packet mark, a TCP connection with a small congestion window can avoid a retransmit timeout [14, 15].

Paganini *et al.* designed a congestion control system that scales gracefully with network capacity, providing high utilization, low queueing delay, dynamic stability, and fairness among users [16]. They developed a packet-level implementation using ECN marking. However, they also state that the main obstacle to the implementation of these kinds of protocols is that they require substantial changes to current practice at both routers and end-systems.

We also do not want to propose an approach, which requires changes in host computers and edge routers. Layer interaction creates a big problem and we rather want to use TCP's end-to-end congestion control mechanism. A control method, which requires only changes in the link algorithm that is executed in network devices, will be easier to implement in today's world.

2.3.3.2. Other Works for Explicit Feedback. There are also other literature works, which avoid to use ECN and try to develop other explicit feedback mechanisms, however they make further assumptions on the underlying network. Kalampoukas *et al.* [17] focuses mainly on an environment, when the end systems are part of IP datagram net-

works that are interconnected by a rate-controlled ATM (Asynchronous Transfer Mode) virtual circuit over a wide area. Hence, they put their interest on the behavior of TCP congestion control algorithms in an internetwork consisting of both rate-controlled and non-rate controlled segments. They proposed an explicit feedback scheme, called Explicit Window Adaptation (EWA). Savoric [18] makes further enhancements in the EWA algorithm, while calculating the utilization factor in the original algorithm by a fuzzy controller depending on the current and the last measured queue length in a router. For some of the ideas proposed in his work, there is the need of changing the TCP in the end systems, which cannot be easily implemented in today's world.

2.3.4. ARED - Adaptive RED

RED can be very aggressive and can cause under-utilization when number of flows sharing the bottleneck link is small. Feng *et al.* concluded that RED needs to be tuned for the dynamic characteristics of the aggregate traffic on a given link [19]. They proposed a self-configuring algorithm for RED by adjusting max_p every time the average queue length falls out of the target range between min_{thresh} and max_{thresh}. When the average queue length is smaller than min_{thresh}, max_p is decreased multiplicatively to reduce REDs aggressiveness in marking or dropping packets; when the queue length is larger than max_{thresh}, max_p is increased multiplicatively. Floyd *et al.* improved upon this original adaptive RED proposal by replacing the MIMD (multiplicative increase multiplicative decrease) approach with an AIMD (additive increase multiplicative decrease) approach [10]. They also provided guidelines for choosing min_{thresh}, max_{thresh} and the weight w_q, for computing a target average queue length.

They state that min_{thresh} is a policy choice and use Equation 2.5 to calculate it as a function of link capacity C in packets per second.

$$min_{thresh} = \max\left[5, \frac{5ms \times C}{2}\right] \tag{2.5}$$

The default value for max_{thresh} is 3 times min_{thresh}. The weight for calculating the

average queue length is determined as in Equation 2.6.

$$w_q = 1 - \exp(-\frac{1}{C}) \tag{2.6}$$

where C is the link capacity in packets per second, computed for packets of the specified default size.

Figure 2.13 shows the operation of ARED and explains the algorithm.

Every *interval* seconds:

- if ($avg_{queue} > target$ and $max_p \leq 0.5$) then
  - increase max_p:
  - $max_p \leftarrow max_p + \alpha$
- else if ($avg_{queue} < target$ and $max_p \geq 0.01$) then
  - decrease max_p:
  - $max_p \leftarrow max_p \times \beta$

where we have as

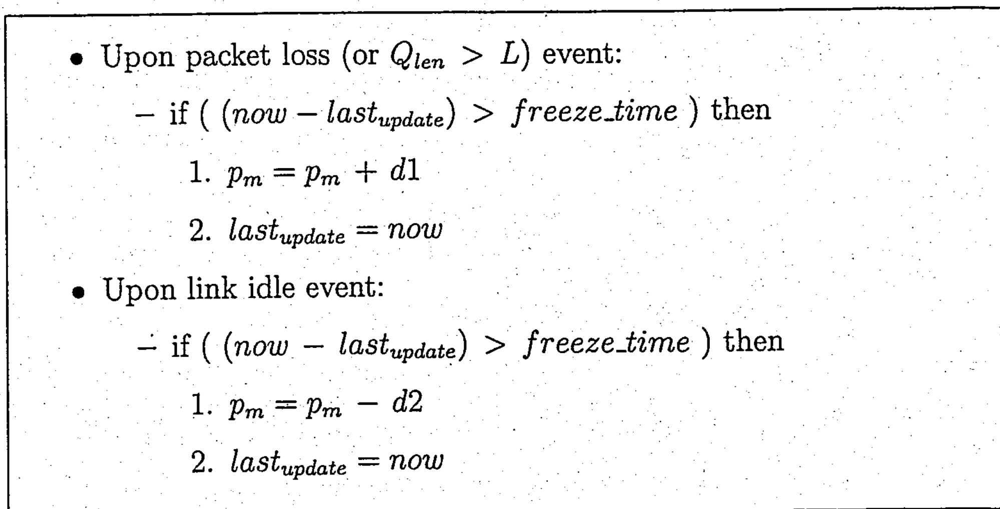
- variables:
  - avg_{queue} is the average queue length
- fixed parameters:
  - *interval* is time $\rightarrow$ 0.5 seconds
  - *target* is the target for $avg_{queue} \rightarrow$ $min_{thresh} + [0.4, 0.6] \times (max_{thresh} - min_{thresh})$
  - α is the increase factor $\rightarrow \min\left(0.01, \frac{max_p}{4}\right)$
  - β is the decrease factor $\rightarrow$ 0.9

Figure 2.13. Floyd's version of ARED algorithm

2.3.5. BLUE - Another Congestion Control Approach

Most of the queue management algorithms use queue lengths as the indicator of the severity of congestion. BLUE is a different congestion control algorithm in the sense that it uses packet loss and link utilization history to manage congestion. Feng *et al.* [20] state that if congestion is indicated by the presence of a persistent queue, its length gives very little information as to the severity of congestion, that is, the number of competing connections sharing the link. In a busy period, a single source transmitting at a rate greater than the bottleneck link capacity can cause a queue to build up just as easily as a large number of sources can. Since the RED algorithm relies on average queue lengths, it has an inherent problem in determining the severity of congestion. As a result, RED requires a wide range of parameters to operate correctly under different congestion scenarios.

BLUE maintains a single probability, p_m, which it uses to mark (or drop) packets when they are queued. If the queue is continually dropping packets due to buffer overflow, BLUE increments the marking probability, thus increasing the rate at which it sends back congestion notification. Conversely, if the queue becomes empty or if the link is idle, BLUE decreases its marking probability. Basically the BLUE algorithm has the following steps shown in Figure 2.14. As it can be seen from the steps of the



- Upon packet loss (or $Q_{len} > L$) event:
  - if ($(now - last_{update}) > freeze_time$) then
    1. $p_m = p_m + d1$
    2. $last_{update} = now$
- Upon link idle event:
  - if ($(now - last_{update}) > freeze_time$) then
    1. $p_m = p_m - d2$
    2. $last_{update} = now$

Figure 2.14. BLUE algorithm

algorithm, if the queue is continually dropping packets due to buffer overflow, BLUE increments p_m, thus increasing the rate at which it sends back congestion notification. Conversely, if the queue becomes empty or if the link is idle, BLUE decreases its marking probability. This effectively allows BLUE to "learn" the correct rate it needs to send back congestion notification. The steps of the algorithm also indicate that the marking probability is updated when the queue length exceeds a certain value. This modification allows room to be left in the queue for transient bursts and allows the queue to control queuing delay when the size of the queue being used is large. Besides the marking probability, BLUE uses two other parameters which control how quickly the marking probability changes over time. The first is $freeze_time$. This parameter determines the minimum time interval between two successive updates of p_m. This allows the changes in the marking probability to take effect before the value is updated again. The other parameters used, $d1$ and $d2$, determine the amount by which p_m is incremented when the queue overflows or is decremented when the link is idle respectively.

2.3.6. REM - Random Exponential Marking

In 2001, Athuraliya *et al.* proposed the Random Exponential Marking (REM) AQM scheme [21]. REM periodically updates a congestion measure called "price" that reflects any mismatch between packet arrival and departure rates at the link (i.e., the difference between the demand and the service rate) and any queue size mismatch (i.e., the difference between the actual queue length and its target value). The measure p_{REM} is computed by:

$$p_{REM}(t) = \max\left[0, p_{REM}(t-1) + \gamma \times (\alpha_{REM} \times (q(t) - q_{ref}) + x(t) - C)\right] \quad (2.7)$$

where C is the link capacity (in packet departures per unit time), $p_{REM}(t)$ is the congestion measure, $q(t)$ is the queue length, and $x(t)$ is the packet arrival rate, all determined at time t and $\gamma > 0$ and $\alpha_{REM} > 0$ are small constants. As with ARED, the control target is expressed by the queue size. The mark/drop probability in REM

is defined as in Equation 2.8:

$$prob(t) = 1 - \phi^{-p_{REM}(t)} \tag{2.8}$$

where $\phi > 1$ is a constant.

The comparison of the marking probabilities between RED (gentle variant) and REM is shown in Figure 2.15.

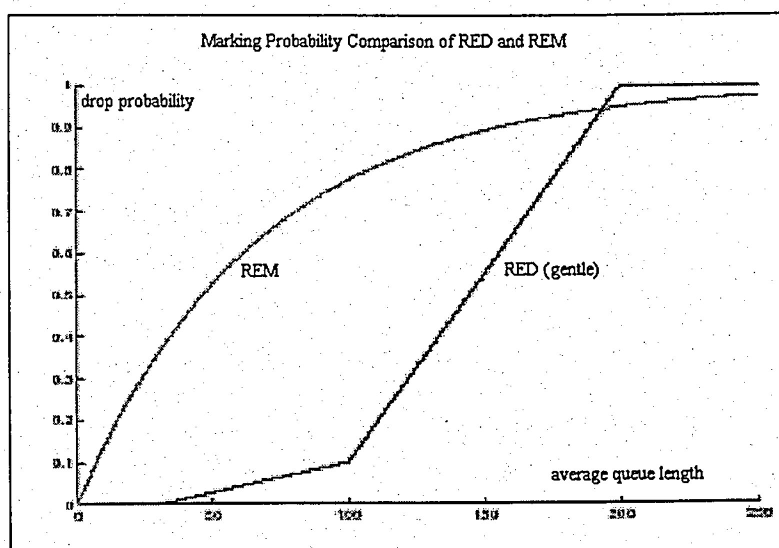


Figure 2.15. Marking probability of gentle variant of RED and REM

In overload situations, the congestion price increases due to the rate mismatch and the queue mismatch. Thus, more packets are dropped or marked to signal TCP senders to reduce their transmission rate. When congestion diminishes, the congestion price is reduced because the mismatches are now negative. This causes REM to drop or mark fewer packets and allows the senders to potentially increase their transmission rate. It is easy to see that a positive rate mismatch over a time interval will cause the queue size to increase. Conversely, a negative rate mismatch over a time interval will drain the queue. In summary, the rate mismatch in REM can be detected by comparing the instantaneous queue length with its previous sampled value.

2.3.7. PI Controller - Congestion Control Based on an Analytical Model

This design is based on the classical control theory. In classical control, the controller tries to keep the plant output on a desired level via performing the necessary action based on the error between the reference value and the output. This is basically what inspired us to design a feedback controller, as it is done in PI design, however using fuzzy control approach.

To design a feedback controller, there is a need of a model of TCP, if fuzzy control is not used. An analytical model has been developed by Misra *et al.* [22], which is capable of tracking the dynamics of TCP Reno and its AIMD behaviour. There are also some controller designs [7, 23, 24] and different simulation scenarios including satellite networks, which are used to tune RED parameters [25], based on the analytical model in [22].

One of the drawbacks of the analytical model is that it ignores the time out mechanism. The packet losses are indicated either by a time out or by triple duplicate ACKs. TCP Reno gives different responses to each of them and changes the window size accordingly. Widmer *et al.* [23] suggested taking time out into consideration, while modelling TCP throughput. Balakrishnan *et al.* [26] indicated that almost 50 per cent of all losses in a busy Internet server required a time out to recover. This also suggests including a time out model in the controlling approach, if an approach based on an analytical model will be used. In that case, one can refer to Padhye *et al.* [27] who proposed an analytical model for time out mechanism and also provided its simplification.

Hollot *et al.* analyze RED from a control theoretic point of view using the model described in [22]. In this approach however, TCP timeout mechanism is ignored. They also simplified the TCP/AQM model to a linear system [7] and designed a Proportional Integrator (PI) controller that regulates the queue length to a target value called the "queue reference", q_{ref} [24]. The PI controller uses instantaneous samples of the queue length taken at a constant sampling frequency as its input. The drop probability is

computed as:

$$p(kT) = a \times \Big(q(kT) - q_{ref}\Big) - b \times \Big(q([k-1]T) - q_{ref}\Big) + p\Big((k-1)T\Big) \tag{2.9}$$

where $p(kT)$ is the drop probability at the kth sampling interval, $q(kT)$ is the instantaneous sample of the queue length, a and b are optimized parameters, which are positive small and close to zero and a being slightly greater than b, for the PI design based on the analytical model in [22] and T is $1/samplingfrequency$. A close examination of this equation shows that the drop probability increases in sampling intervals when the queue length is higher than its target value. Furthermore, the drop probability also increases if the queue has grown since the last sample (reflecting an increase in network traffic). Conversely, the drop probability in a PI controller is reduced when the queue length is lower than its target value or the queue length has decreased since its last sample. The sampling interval and the coefficients in the equation depend on the link capacity, the maximum RTT and the expected number of active flows using the link.

The simulations are done using NS and it is claimed that implementation of P controller has less complexity than that of RED and implementation of PI has similar one. The simulation results show that both controllers provide less delay and PI having better results than P only. From the results it can be seen that the P and PI controllers work with a stable queue size, meaning that they show good performance against congestion at steady state. It is also mentioned that implementing the PI controller in RED capable routers requires a modification to the averaging algorithm. It is required to keep the states of two additional variables. For details, please refer to [28].

3. FUZZY CONTROL AND PARAMETER TUNING

This thesis consists of two different implementations of AQM methods, which are based on the fuzzy control theory. The first one is a DFC (Direct Fuzzy Controller) and the second one is an AFC (Adaptive Fuzzy Controller), which are explained in detail in the following sections. This section addresses mainly the theoretical background.

3.1. Theoretical Background of Fuzzy Control

The motivation behind fuzzy control is to construct nonlinear controllers without having an analytical model and with only using simple heuristics. In today's world, many plants, the dynamics of which cannot be modelled because of the high nonlinearity, are manually controlled by experienced users. Therefore, many problems arise in transferring process knowledge to the control algorithm. Fuzzy control aims to model operator's control actions and thus avoids to deal with a complex model. It can also be used when the model of the plant is very difficult, highly nonlinear and when it does not reflect the real time dynamics properly. In other words, fuzzy control provides a formal methodology for representing, manipulating, and implementing a humans heuristic knowledge about how to control a system.

The heuristics are linguistic words, called rules, and are simply logical actions that a person will perform. A simple example of a rule, which everyone does when driving a car can be the following: *IF Speed is High AND Traffic is Heavy, THEN reduce Gas A Bit.*

The fuzzy controller block diagram is given in Figure 3.1, where a fuzzy controller is shown embedded in a closed-loop control system. The plant outputs are denoted by $y(t)$, its inputs are denoted by $u(t)$, and the reference input to the fuzzy controller is denoted by $r(t)$.

The fuzzy controller has four main components:

Boğaziçi Üniversitesi Kütüphanesi

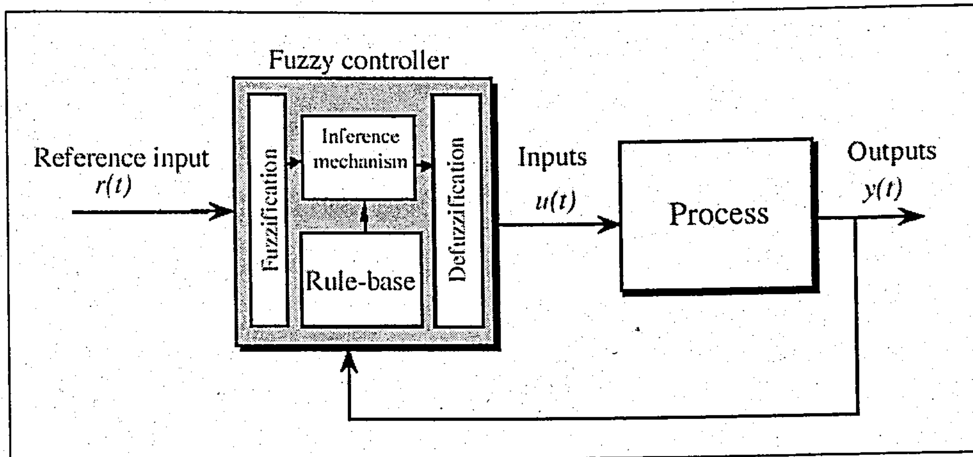


Figure 3.1. Fuzzy controller architecture

- The "rule-base" holds the knowledge, in the form of a set of rules, of how best to control the system.
- The inference mechanism evaluates which control rules are relevant at the current time and then decides what the input to the plant should be.
- The fuzzification interface simply modifies the inputs so that they can be interpreted and compared to the rules in the rule-base.
- The defuzzification interface converts the conclusions reached by the inference mechanism into the inputs to the plant.

Basically, one should view the fuzzy controller as an artificial decision maker that operates in a closed-loop system in real time. It gathers plant output data $y(t)$, compares it to the reference input $r(t)$, and then decides what the plant input $u(t)$ should be to ensure that the performance objectives will be met.

To design the fuzzy controller, the designer should have an insight on how the artificial decision maker should act in the closed-loop system. This way, a set of rules can be written down about how to control the system without outside help. These "rule" basically say, "If the plant output and reference input are behaving in a certain manner, then the plant input should be some value". A whole set of such "If-Then" rules is loaded into the rule-base, and an inference strategy is chosen, then the system is ready to be tested to see if the closed-loop specifications are met [29].

3.2. Fuzzy Sets, Fuzzy Logic and Rule Base

This section will focus on the mathematical definitions of the fuzzy theory. The definitions and the methods provided here are the design guidelines of the direct and adaptive fuzzy controllers proposed.

3.2.1. Membership Functions

In fuzzy set theory, an element x belongs to a fuzzy set A up to a certain degree. The function $\mu_A(x)$ associated with A that maps the universe of discourse Ω to $[0,1]$ is called a "membership function". The membership function describes the "certainty" and hence expresses the degree that the element x belongs to the set A and is mathematically defined as in 3.1.

$$\mu_A(x) : \Omega \longrightarrow [0,1] \tag{3.1}$$

In Figure 3.2, a simple example is shown. This is to demonstrate, how a fuzzy membership can be calculated.

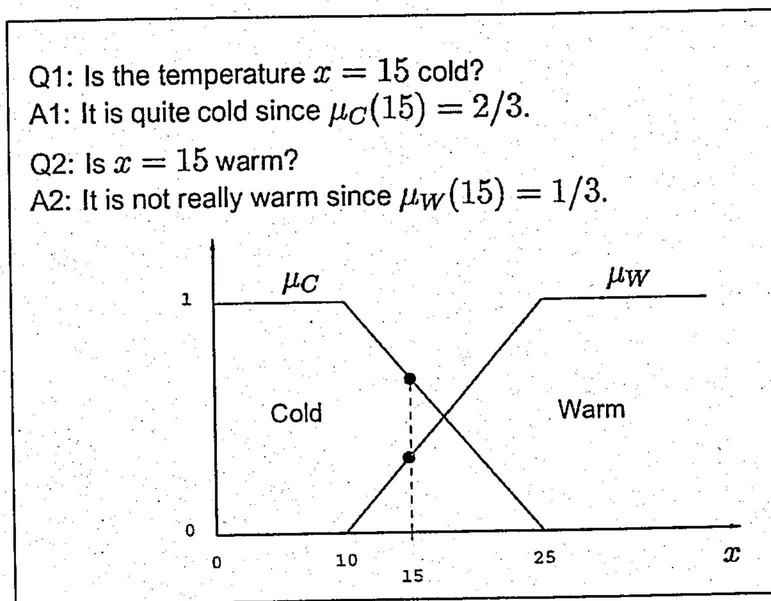


Figure 3.2. A simple membership example

3.2.2. Fuzzy Logic

In fuzzy logic, there is the need to express the heuristics, which are in the form of "If-Then" rules in a solid, mathematical way. Hence, linking words, like "AND", "OR" and "NOT" must be defined mathematically. Although it is not the only option to express them, the *min* and *max* functions are used to express "AND" and "OR" respectively. Table 3.1 provides a comparison of the operations mentioned above for the conventional logic and fuzzy logic.

Table 3.1. Comparison of conventional logic and fuzzy logic

Comparison	Conventional Logic	Fuzzy Logic
AND	$A \bigcap B$	$\mu_{A \bigcap B}(x) = \min(\mu_A(x), \mu_B(x))$
OR	$A \bigcup B$	$\mu_{A \bigcup B}(x) = \max(\mu_A(x), \mu_B(x))$
NOT	A'	$\mu_{A'}(x) = 1 - \mu_A(x)$

3.3. Operation of the Fuzzy Controller

In Section 3.1 the operation of the fuzzy controller is explained. A fuzzy controller consists of the following parts:

- Fuzzifier is responsible of determining degree of membership for the input functions.
- Fuzzy Inference performs mainly fuzzy set calculations.
  - Calculate degree of fulfilment for each rule.
  - Calculate fuzzy output for each rule.
  - Aggregate rule outputs.
- Defuzzifier maps fuzzy set to output u.

The Figures 3.3 - 3.5 demonstrate those operations in respective order.

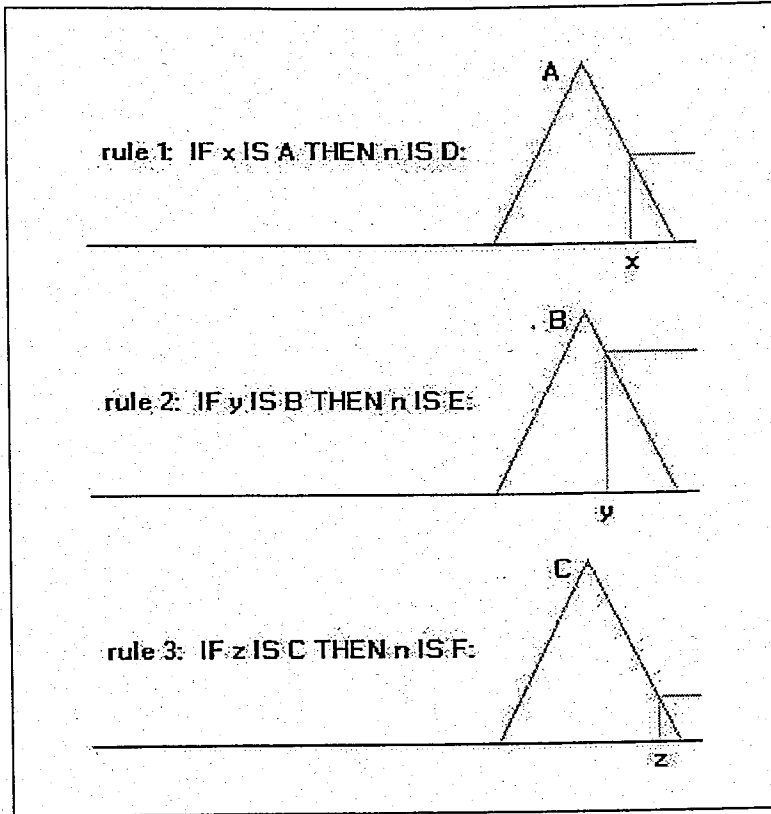


Figure 3.3. Degree of membership determination

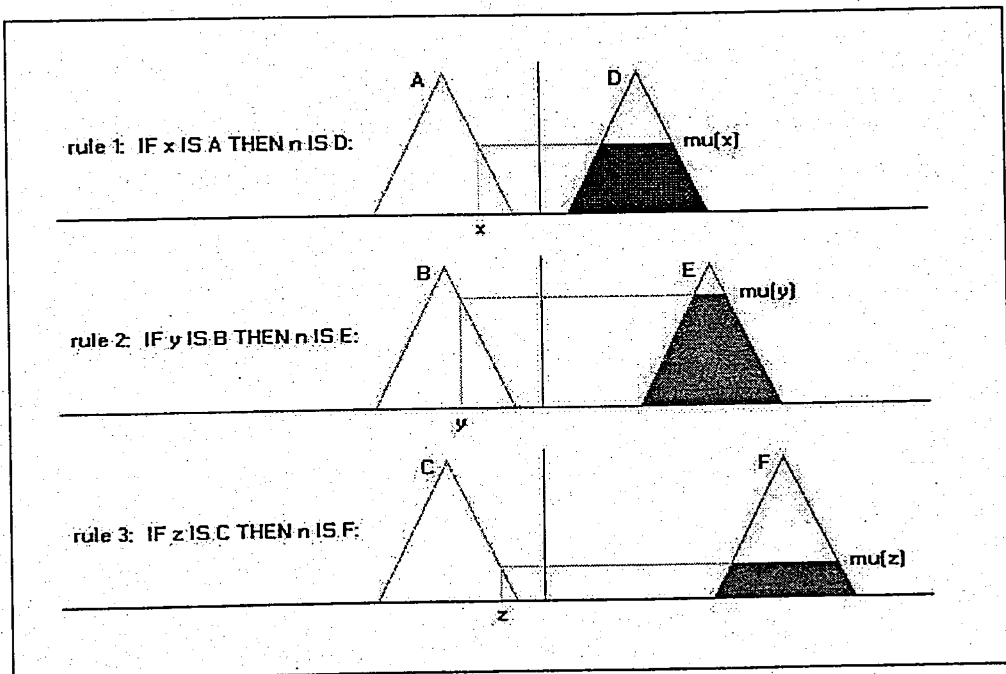


Figure 3.4. Calculation of the fuzzy output

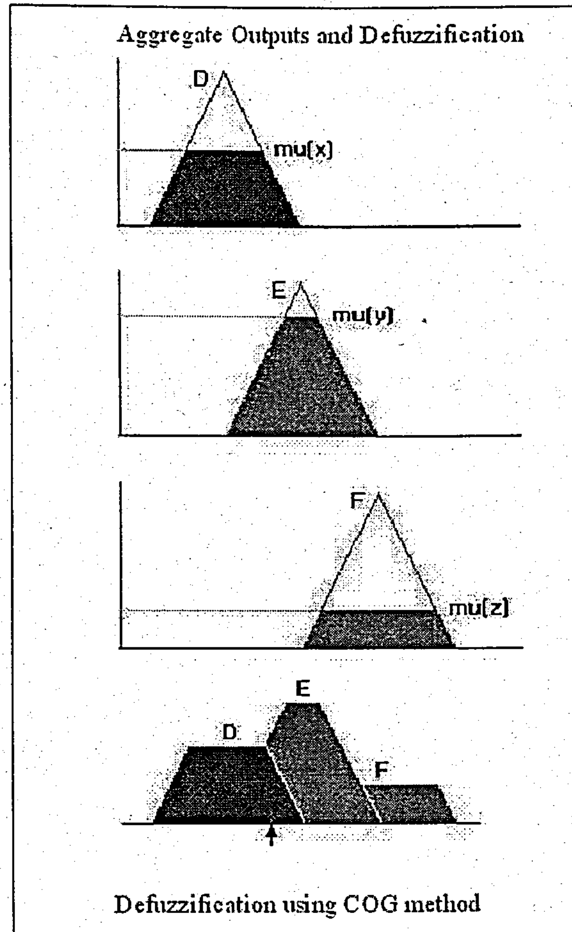


Figure 3.5. COG defuzzification using min-max inferencing

The defuzzifier used throughout the design of both direct fuzzy controller and adaptive fuzzy controller is based on a simple center of gravity calculation, in which the center of gravity is determined from the aggregate area calculated from individual areas that are the fuzzy outputs for each rule and which in turn is determined by the fuzzy inference procedure.

4. MOTIVATION, DETAILED VIEW ON THE DESIGNS AND PARAMETER TUNING

Our motivation for designing a fuzzy controller is mainly based on the PI controller explained in Subsection 2.3.7. The PI model is a feedback controller and it is an approach based on the classical control theory. The drawbacks of the PI design are that slow-start and time-out are ignored. The analytical model for the TCP is highly nonlinear and slow-start cannot be included because of the linearization done, since the operating point is the congestion avoidance phase. This nonlinearity of the TCP mechanism are shown with the analytical model in [28] using Equations 4.1 - 4.3:

$$\dot{W}(t) = \frac{1}{R(t)} - \frac{W(t)}{2} \times \frac{W(t-R(t))}{R(t-R(t))} \times p(t-R(t)) \tag{4.1}$$

$$\dot{q}(t) = \begin{cases} -C\frac{N(t)}{R(t)} \times W(t), & q > 0 \\ \max\left\{0, -C\frac{N(t)}{R(t)} \times W(t)\right\}, & q = 0 \end{cases} \tag{4.2}$$

where $\dot{A}$ denotes time derivative, $q(t)$ is the queue length at time t, $R(t)$is the RTT at time t and calculated as in Equation 4.3,

$$R(t) = \frac{q(t)}{C} + T_p(sec) \tag{4.3}$$

W is the average TCP window size, C is the link capacity, T_p is the propagation delay, N is the number of TCP connections, and p is the mark-drop probability, which itself is a function of $q(t)$ in an AQM mechanism. We do not want to go into the details of those Equations and linearization method, however, note that the Equations listed above along with the Equation for p, which is specific for the AQM mechanism that is used, build together a differential equation system, i.e. they are coupled and have to be solved simultaneously. Hollot *et al.* [28] state the linearization technique and list the assumptions they made, which are also discussed to some extent in Subsection 2.3.7.

Our aim is to overcome the lack of dynamics arose from the simplifying assumptions in the design of the PI controller. The success of capturing the slow-start dynamics can be examined from the simulation results in Section 5.

4.1. Some Design Issues for DFC and AFC

This section aims to present the design guidelines used for both direct and adaptive fuzzy controllers. It is found that those design guidelines showed better results than the standard implementations.

4.1.1. Output Membership Function Tuning

In this method the positioning of the output membership functions are tuned by characterizing their centers by a function. To tune the output membership functions of both adaptive and direct fuzzy controllers Equation 4.4 is used, which calculates the centers c^i as:

$$c^i = i^{power}/max_i^{power} \tag{4.4}$$

where c^i is the location of the output center i, *power* is the design parameter, which is different for both fuzzy controllers and max_i is the number of triangles for the output membership function and used for normalization purposes. This procedure will have the effect of making the output membership function centers near the origin be more closely spaced than the membership functions farther out on the horizontal axis. The effect of this is to make the "gain" of the fuzzy controller smaller when the signals are small and larger as the signals grow larger. Hence, the use of Equation 4.4 for the centers indicates that if the values of the input membership functions are near where they should be, then do not make the force input to the plant too big, otherwise the force input should be much bigger so that it quickly returns the queue length to near the referenced position.

Figure 4.1 indicates the view of the input membership functions for the direct

and adaptive fuzzy controller. Figure 4.2 shows the output membership function for

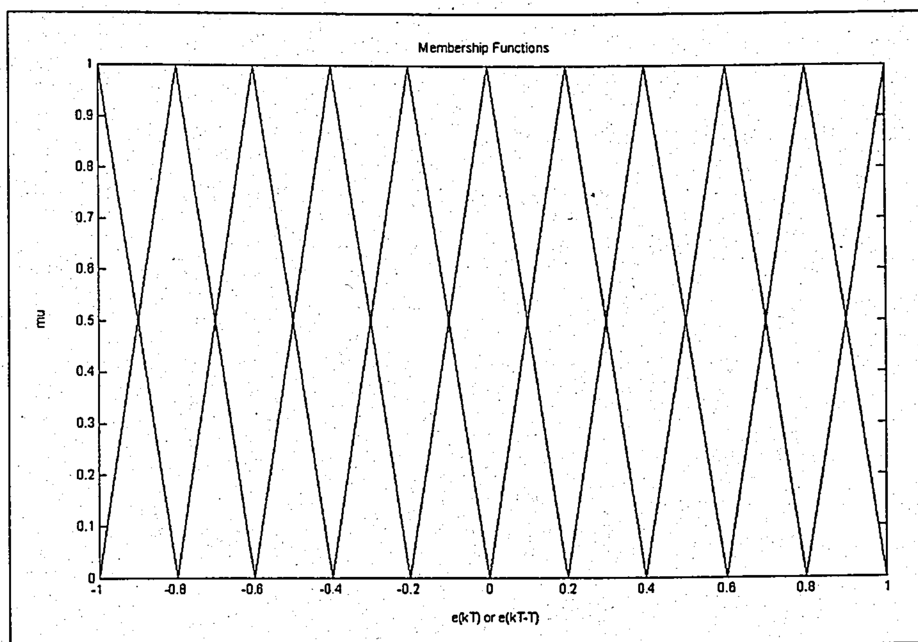


Figure 4.1. Membership functions of the both direct and adaptive fuzzy controller

the direct fuzzy controller after tuning.

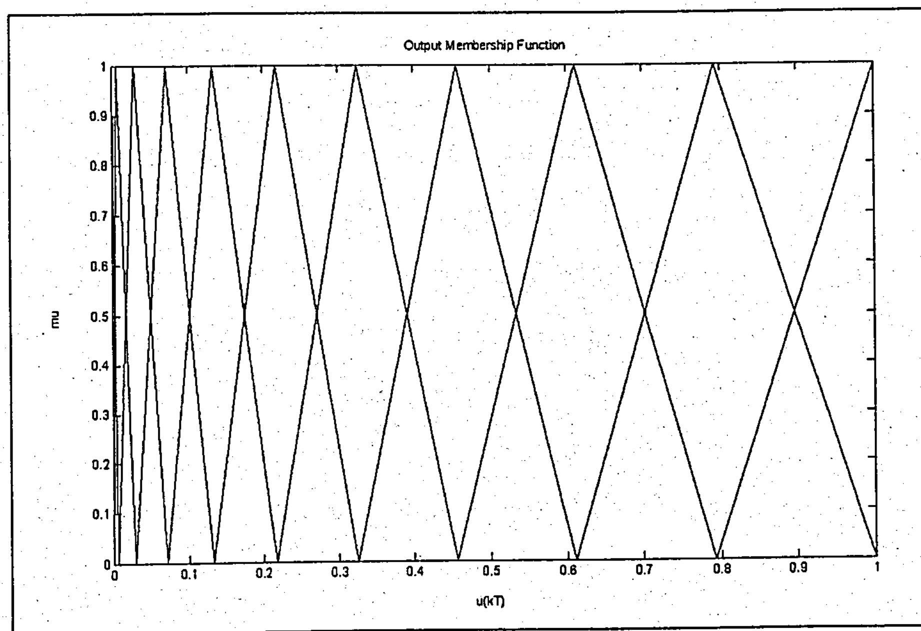


Figure 4.2. Output membership function of the direct fuzzy controller

Ultimately, the goal of tuning is to shape the nonlinearity that is implemented by the fuzzy controller. This nonlinearity, sometimes called the "control surface", is affected by all the main fuzzy controller parameters. Figure 4.3 shows the control surface of the fuzzy inference engine for the direct fuzzy controller. This control surface is shaped by the rule base and the linguistic values of the linguistic variables, where the output of the fuzzy controller is plotted against its two inputs. The surface represents in a compact way all the information in the fuzzy controller [29].

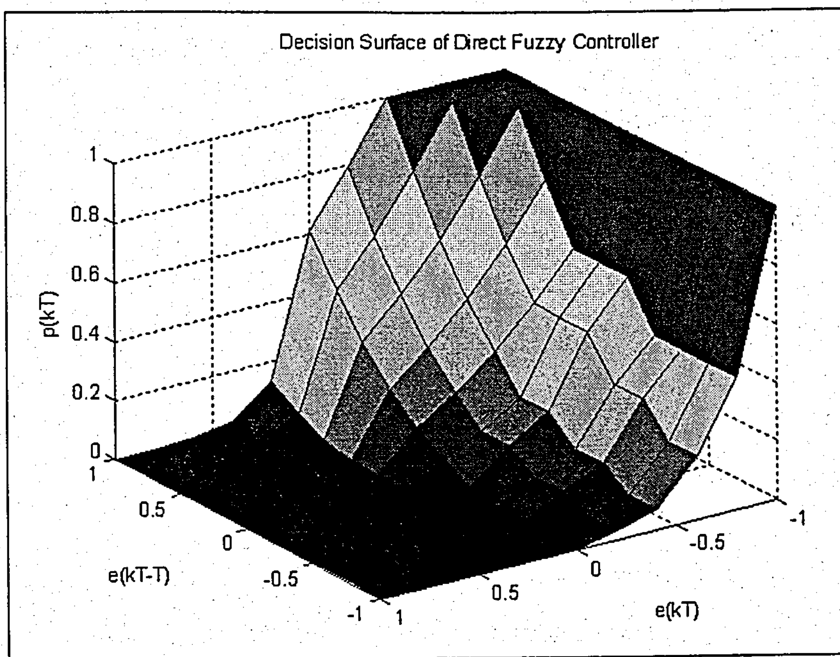


Figure 4.3. Decision surface of the direct fuzzy controller

Table 4.1 shows the fuzzy rule base for the direct fuzzy controller and the adaptive part of the adaptive fuzzy controller. The intersection of the instantaneous error and the previous error in Table 4.1 indicates which triangle to use in the output membership function. Those represent the heuristics of a human controller and the linguistics of them are based on the design of the fuzzy controller of Chrysostomou *et al.* [30]. Although the shape and number of the triangles are different, linguistic words and the logic behind is very similar.

Table 4.1. Fuzzy rule base

$p(kT)$		$q_{ref}(kT-T)-q(kT-T)$										
		-5	*-4*	*-3*	*-2*	*-1*	*0*	*1*	*2*	*3*	*4*	*5*
$q_{ref}(kT)-q(kT)$	*-5*	10	10	10	10	10	10	10	10	10	10	10
	-4	7	7	7	7	8	8	8	9	10	10	10
	-3	5	5	6	6	7	7	7	8	9	10	10
	-2	3	3	4	4	5	5	6	7	8	9	10
	-1	2	2	3	3	4	4	5	6	7	8	9
	0	0	0	0	1	2	3	4	5	6	7	8
	1	0	0	0	0	0	0	0	1	2	3	4
	2	0	0	0	0	0	0	0	0	1	1	2
	3	0	0	0	0	0	0	0	0	0	0	1
	4	0	0	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	0	0	0	0	0	0

4.1.2. Output Scaling Gain

For both designs in this thesis, an output factor, *gain*, is applied to the output membership functions, a good value of which is found to be 0.4, Since it is less than one, there is the effect of contracting the output membership functions and hence making the meaning of their associated linguistics quantify smaller numbers.

4.1.3. Design Issues Specific to Adaptive Fuzzy Controller

The AFC is a FMRLC (Fuzzy Model Reference Learning Controller) and its functional block diagram is shown in Figure 4.4. It has four main parts: the plant, the fuzzy controller to be tuned, the reference model, and the learning mechanism (an adaptation mechanism). The FMRLC uses the learning mechanism to observe numerical data from a fuzzy control system (i.e., $r(kT)$ and $y(kT)$ where T is the sampling period). Using this numerical data, it characterizes the fuzzy control systems current performance and automatically synthesizes or adjusts the fuzzy controller so

that some given performance objectives are met. These performance objectives are characterized via the reference model shown in Figure 4.4. The learning mechanism seeks to adjust the fuzzy controller so that the closed-loop system (the map from $r(kT)$ to $y(kT)$) acts like the given reference model (the map from $r(kT)$ to $y_m(kT)$). Basically, the fuzzy control system loop (the lower part of Figure 4.4) operates to make $y(kT)$ track $r(kT)$ by manipulating $u(kT)$, while the upper-level adaptation control loop (the upper part of Figure 4.4) seeks to make the output of the plant $y(kT)$ track the output of the reference model $ym(kT)$ by manipulating the fuzzy controller parameters [29].

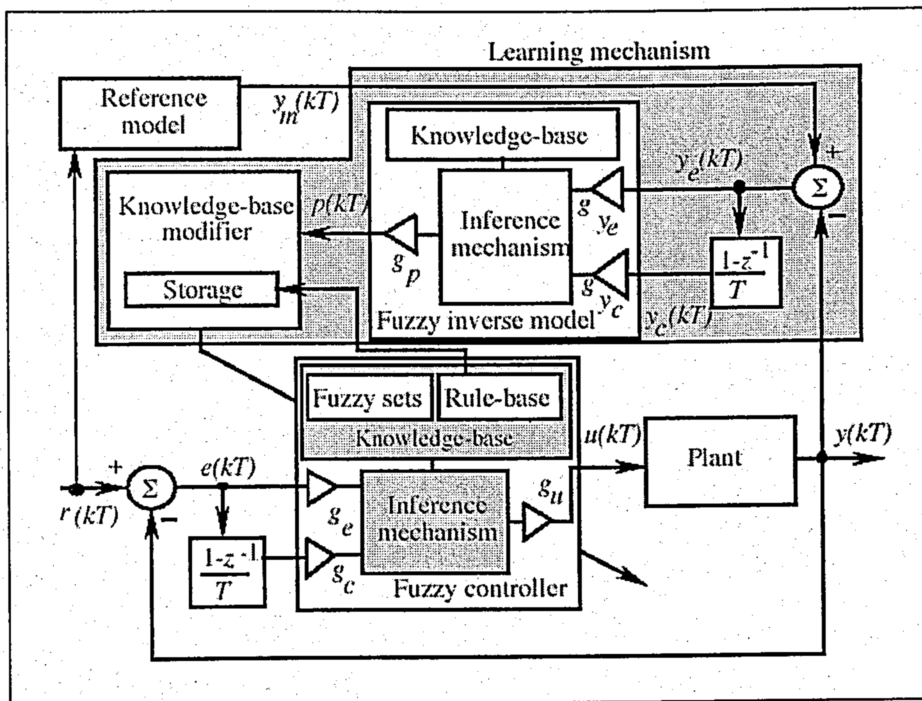


Figure 4.4. FMRLC - adaptive fuzzy controller

In our design of the FMRLC, there is no mapping from $r(kT)$ to $y_m(kT)$. We only have a reference value for the queue length q_{ref} and it is independent from the sampling period. The model part is based on the input membership functions of error, e defined as in Equation 4.5,

$$e(kT) = \frac{q_{ref} - q(kT)}{q_{ref}} \tag{4.5}$$

and the change of error, *edot*, defined as in Equation 4.6:

$$edot = \frac{e(kT) - e(kT - T)}{T} \tag{4.6}$$

The learning part operates a little bit different than the reference part in the sense that it uses $e(kT)$ and $e(kT - T)$ as the input membership functions.

The input membership functions are defined to characterize the premises of the rules that define the various situations in which rules should be applied. The input membership functions are left constant and are not tuned by the FMRLC. They are the same as in the case of direct fuzzy controller. The membership functions on the output universe of discourse are assumed to be unknown. They are what the FMRLC will automatically synthesize or tune. Hence, the FMRLC tries to fill in what actions ought to be taken for the various situations that are characterized by the premises.

The centers are initialized to zero at the beginning. Passino *et al.* suggest simply placing the membership function centers at zero as a common choice, since it is not always possible to pick such a controller that is the best for the operating condition that the plant will begin in because of the inability to measure the operating condition of the plant [29].

The difference in the design of the adaptive fuzzy controller lies first in the center update procedure. The principle used in the update is the following: *When a center is updated, wait one or more steps before updating the center again.* This can be useful as a more "cautious" update procedure. It updates, then waits to see if the update was sufficient to correct the error ye before it updates again. According to Passino *et al.*, this is used to avoid inducing oscillations when operating at a set-point [29].

4.2. Simulation Environment and Scenarios

This section explains the simulation environment and provides information about the simulation scenarios used throughout this work. This way, it will be easier to

understand the method for choosing the parameters of the both fuzzy controllers.

4.2.1. Simulation Environment

For the simulations the latest version (currently 2.27) of NS (Network Simulator) [31] is used. NS is an event driven network simulator developed at UC Berkeley that simulates variety of IP networks. It implements network protocols such as TCP and UDP (User Datagram Protocol), traffic source behavior such as FTP, Telnet, Web, CBR (Constant bit rate) and VBR (Variable bit rate), router queue management mechanism such as Drop-Tail and RED, routing algorithms such as Dijkstra, and more. NS also implements multicasting and some of the MAC (Medium Access Control) layer protocols for LAN (Local area network) simulations. The NS project is now a part of the VINT (Virtual InterNetwork Testbed) project that develops tools for simulation results display, analysis and converters that convert network topologies generated by well-known generators to NS formats. As shown in Figure 4.5, NS is object ori-

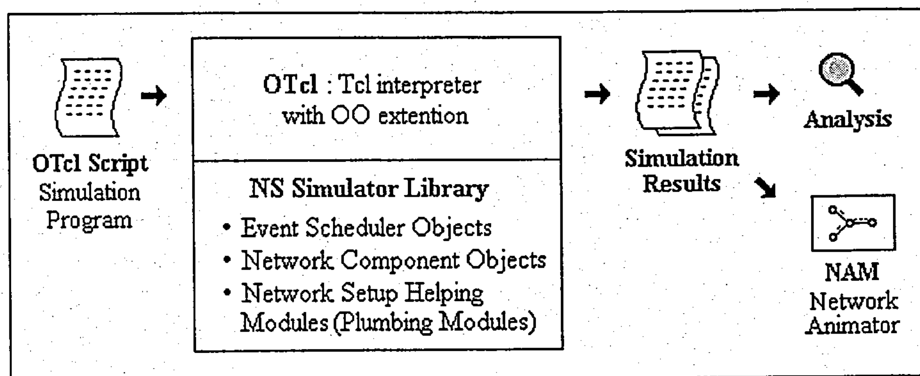


Figure 4.5. Simplified user's view of NS

ented Tcl (OTcl) script interpreter that has a simulation event scheduler and network component object libraries, and network setup (plumbing) module libraries (actually, plumbing modules are implemented as member functions of the base simulator object). In other words, to use NS, the programs are written in OTcl script language. To setup and run a simulation network, a user should write an OTcl script that initiates an event scheduler, sets up the network topology using the network objects and the plumbing functions in the library, and tells traffic sources when to start and stop transmitting

packets through the event scheduler. The term "plumbing" is used for a network setup, because setting up a network is plumbing possible data paths among network objects by setting the "neighbor" pointer of an object to the address of an appropriate object. When a user wants to make a new network object, he or she can easily make an object either by writing a new object or by making a compound object from the object library, and plumb the data path through the object.

NS is written not only in OTcl but in C++ also. For efficiency reason, NS separates the data path implementation from control path implementations. In order to reduce packet and event processing time (not simulation time), the event scheduler and the basic network component objects in the data path are written and compiled using C++. These compiled objects are made available to the OTcl interpreter through an OTcl linkage that creates a matching OTcl object for each of the C++ objects and makes the control functions and the configurable variables specified by the C++ object act as member functions and member variables of the corresponding OTcl object. In this way, the controls of the C++ objects are given to OTcl [31]. This is how we added and configured our designs of direct and adaptive fuzzy controllers.

4.2.2. Simulation Scenarios

The simulation scenarios include different number of parameters, however the basic design is as shown in Figure 4.6.

The simulation time is 120 seconds for every simulation and the packet size is 1000 bytes. The bottleneck link has a rate of 2 Mbps, whereas the sources have links of 10 Mbps. The links to the destinations have also the capacity of 10 Mbps with a delay of 4 ms. For the simulations, some metrics are changed for different cases, which are:

- Buffer size at router
- Number of nodes creating connections
- Delay at the bottleneck link

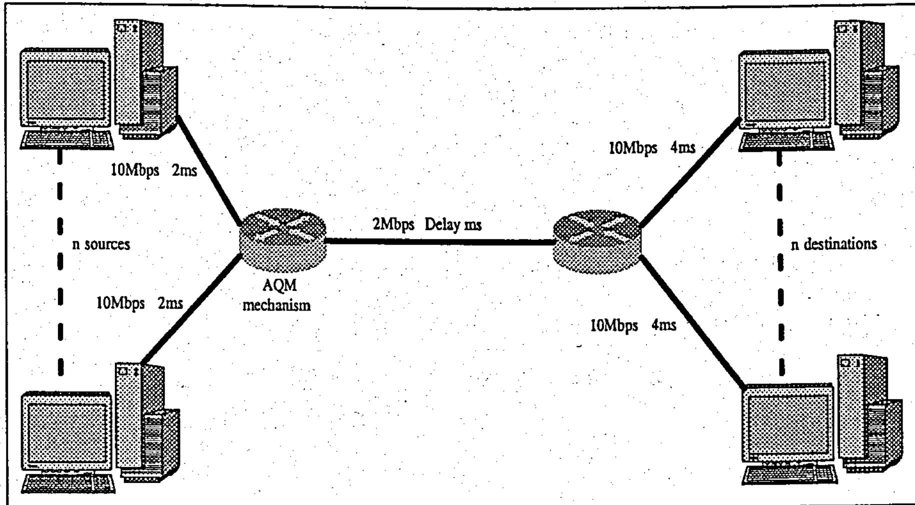


Figure 4.6. Basis for the simulation scenario

- Type of traffic
  - FTP
  - CBR
  - WEB
- Dynamic change in the FTP traffic

The results of those changes are examined in Section 5.

For the case of wireless simulations, the basic design for the scenario is also based on Figure 4.6. However, source nodes in the wireless case are mobile nodes and their link delays are higher. In the wireless scenarios, the router using the AQM algorithm can be imagined as a wireless router.

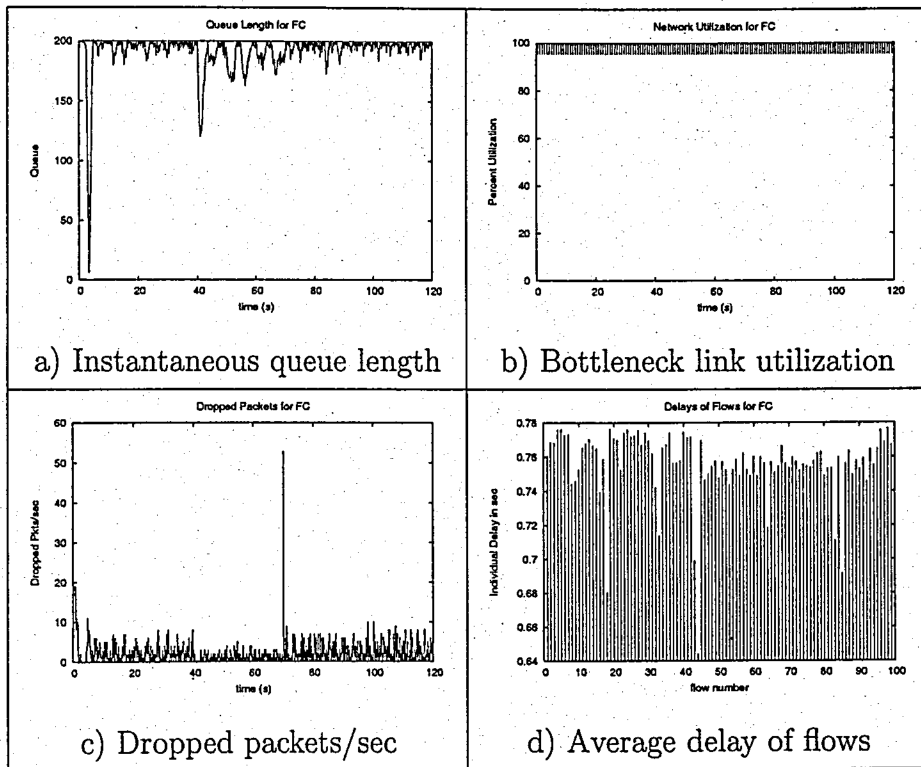
4.3. Search for Good Parameters for DFC and AFC

This section provides the results for good parameters which are suitable for our designs. It is not claimed that the designs of the direct fuzzy controller and the adaptive one are optimum. Designing a fuzzy controller is based on the designer only, since it does not use an analytical model, and one can come up with another solution using different heuristics or even using the same heuristics. However, the values of the design

parameters found are good ones for those particular designs.

4.3.1. Choosing Parameters for Direct Fuzzy Controller

In the design of direct fuzzy controller, the important parameters are the *power* factor used for determining the width of the output membership function and the output gain. The Figures 4.7 - 4.9 provide some of the simulation results that are near the good values.



a) Instantaneous queue length

b) Bottleneck link utilization

c) Dropped packets/sec

d) Average delay of flows

Figure 4.7. Direct fuzzy controller - $power = 2$; $gain = 0.4$

As it can be seen from the Figures, it is found that a good choice for *power* is 2.2 and a good choice for the output gain *gain* is 0.4, since those values lead to less oscillations in the queue length, less packet drops/sec and more fair with the same network utilization.

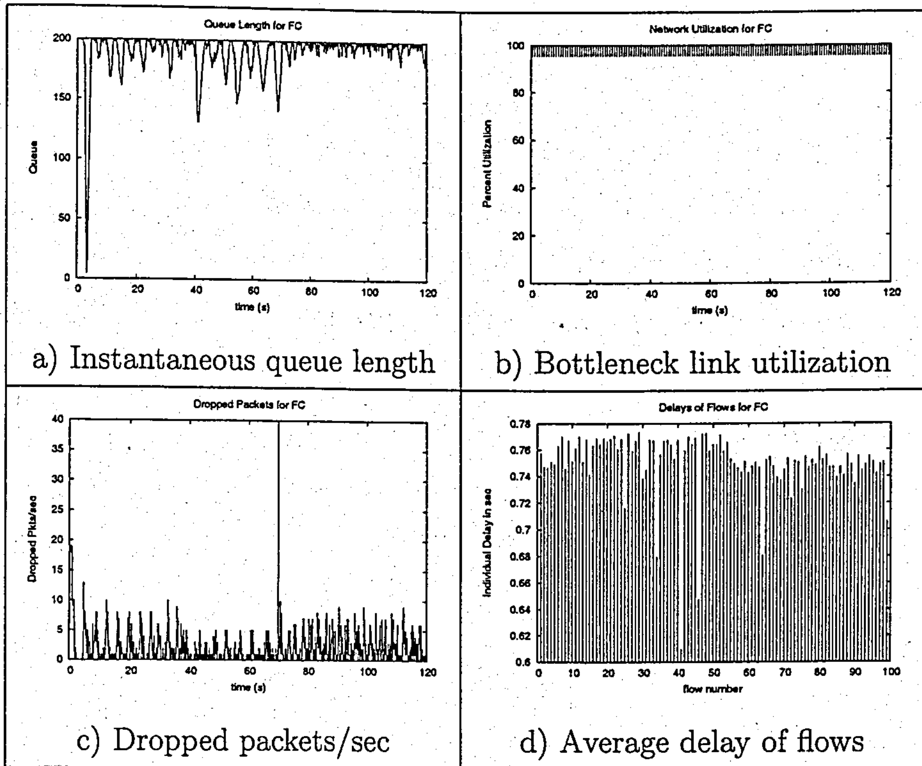


Figure 4.8. Direct fuzzy controller - *power* = 2.2; *gain* = 0.3

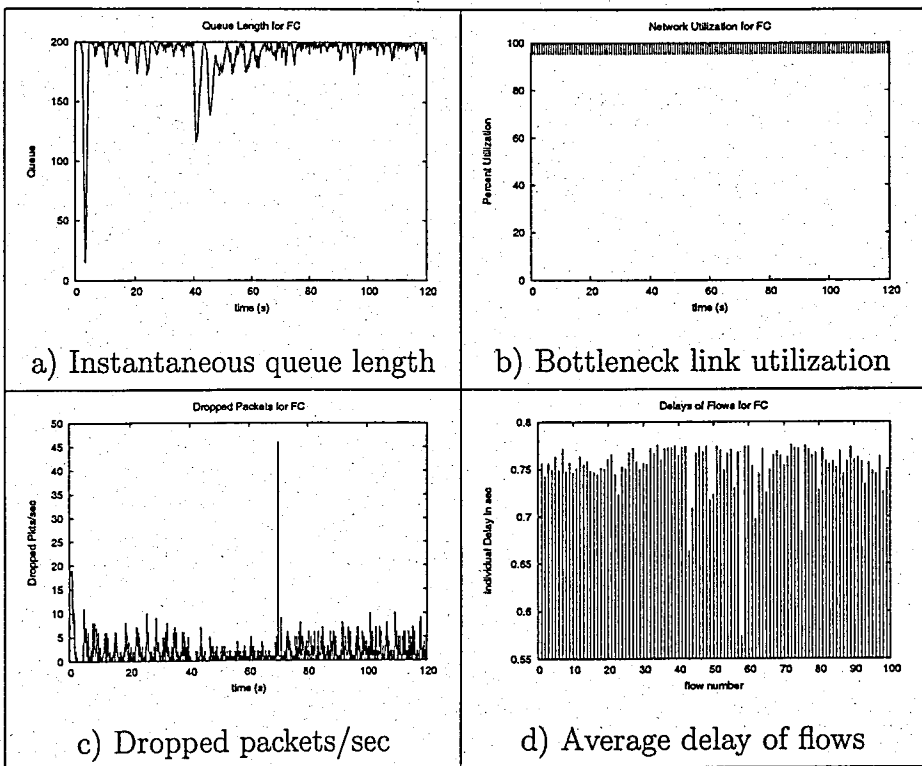
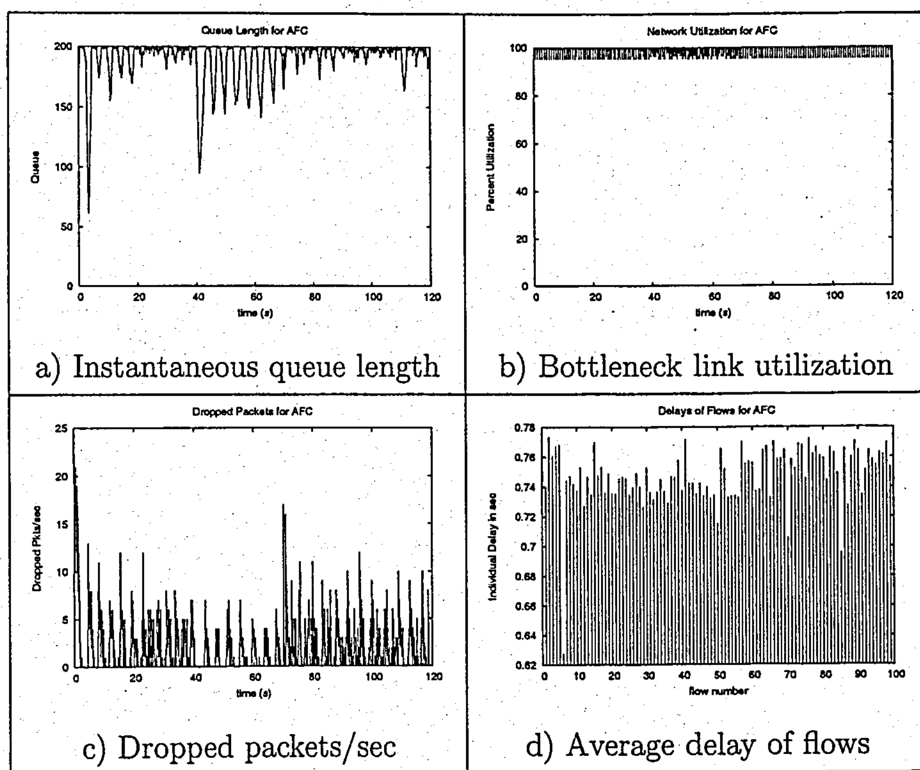


Figure 4.9. Direct fuzzy controller - *power* = 2.2; *gain* = 0.4

4.3.2. Choosing Parameters for Adaptive Fuzzy Controller

In the design of adaptive fuzzy controller, the important design parameters are *cautious* parameter, which is used for a cautious update of the center values of input membership functions. The *cautious* parameter, if it is set to a value grater than one, is used to update the centers only after $cautious - 1$ steps later, if they have been updated already. This is a similar approach to the $freeze_time$ parameter implemented in BLUE. Another important parameter is output gain, as in the design of the direct fuzzy controller. The Figures 4.10 - 4.12 provide some of the simulation results that are near the good values.



a) Instantaneous queue length
b) Bottleneck link utilization
c) Dropped packets/sec
d) Average delay of flows

Figure 4.10. Adaptive fuzzy controller - $cautious = 3$; $gain = 0.5$

As it can be seen from the Figures, it is found that a good choice for *cautious* is three and a good choice for the output gain, *gain*, is 0.4 (as in direct fuzzy controller), since those values lead to less oscillations in the queue length, less packet drops/sec and less delay variation with the same network utilization.

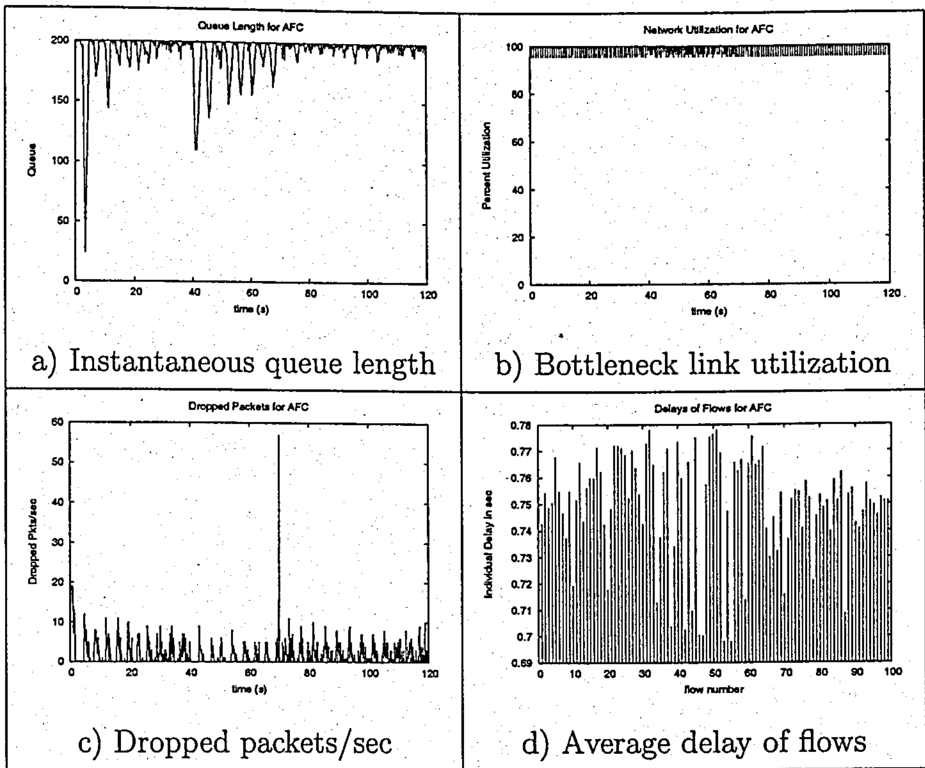


Figure 4.11. Adaptive fuzzy controller - $cautious = 3$; $gain = 0.4$

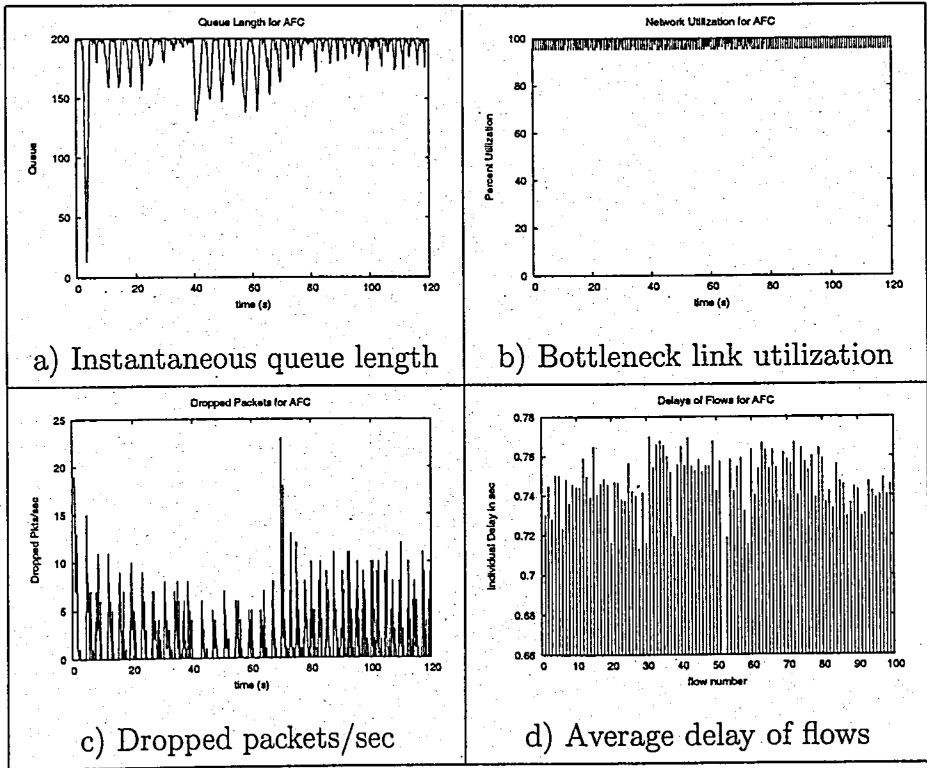


Figure 4.12. Adaptive fuzzy controller - $cautious = 4$; $gain = 0.4$

5. COMPARISON RESULTS OF DIFFERENT AQM MECHANISMS

For the comparisons, different scenarios are prepared, which aim to examine queue length, bottleneck link utilization, average packet delay of individual flows, dropped packets per second and fairness. For the calculation of fairness, F, Jain's fairness index is used [32], which is calculated as in Equation 5.1.

$$F(x) = \frac{\left(\sum x_i\right)^2}{N \times \left(\sum x_i^2\right)} \tag{5.1}$$

where $F(x)$ is the fairness index, x_i the ith user load on the bottleneck link, and N is the number of users sharing the bottleneck resource. Figure 5.1 shows the behavior of Jain's fairness index with only two flows. It can be seen that for the equal share of bandwidth, the index approaches to one.

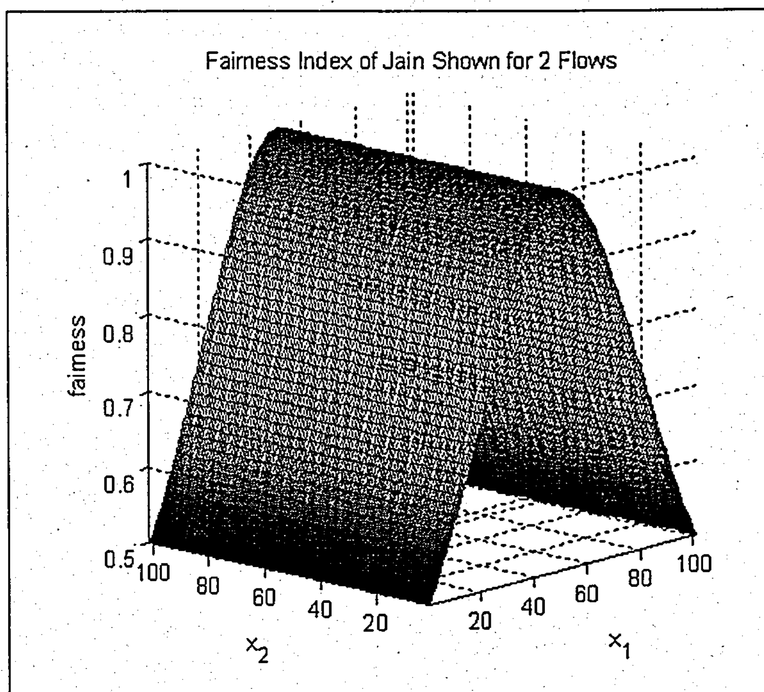


Figure 5.1. Jain's fairness index

Table 5.1 lists the values for the AQM mechanisms used in the simulations. All

Table 5.1. Values of parameters used in different algorithms

AQM	Parameter Details
RED	• operates in gentle mode • parameters are explained in 2.3.2 • initialization is done like ARED, see Subsection 2.3.4 (automatic mode)
ARED	• $interval = 0.5$ sec • initial value of α is 0.01 • $\beta = 0.9$ • operates in gentle mode • initialization is done as in Subsection 2.3.4
REM	• $\gamma = 0.001$ • $\alpha_{REM} = 0.1$ • $\phi = 1.001$
BLUE	• increment: $d1 = 0.0025$ • decrement: $d2 = 0.00025$ • $freeze_time = 100$ ms • $p_m = 1$ • p_m must be close to one for heavy congestion
PI	• parameters are based on the analytical model • $a = 0.00001822$ • $b = 0.00001816$
DFC	• $power = 2.2$ • $gain = 0.4$ • c^i of input membership functions initialized to $0.2 \times i$
AFC	• $power = 18$, used to update output centers • $cautious = 3$ • $gain = 0.4$ • c^i of output membership functions initialized to 0 • c^i of input membership functions initialized to $0.2 \times i$

values are the recommended and the default values for the mechanisms. To provide a

better comparison platform, nothing is changed on those advised values. The parameter setting in RED is not the standard one, however this is the method that S. Floyd [10] suggests and the values are the default ones in NS.

Table 5.2 is shown to summarize different scenarios that are used for the comparison of the AQM mechanisms. Some simulation concepts are based on papers published [25], [30], but the general environment is prepared to provide a high load on the congested router to demonstrate the actions of the different AQM mechanisms compared.

Table 5.3 and Table 5.4 represent Jain's fairness index and the total number of packet drops, both for the case, where $buffer = 150$ in respective order. The Tables aim to show the differences, when only a parameter is changed. This is exactly the reason for the big number of simulations.

Table 5.5 and Table 5.6 represent Jains's fairness index and the total number of packet drops, both for the case, where $buffer = 200$ in respective order. In those simulations, the effect of setting ECN bit to true is also examined.

5.1. Overview of the Simulation Results

This section aims to give a general idea of the results obtained from the simulations. Detailed examination will follow after this overview in order to give the reader a better understanding.

5.1.1. Change in Bottleneck Delay

There are some important factors, which are common for the cases, where the bottleneck delay is increased from 5 ms to 125 ms, i.e. RTT is increased. 125 ms is enough to simulate MEO (middle Earth orbit) satellite links [25]. If there is an increase in the RTT, it can be seen from the simulation results that packet drop is decreased for each mechanism. The reason of this is because of the increase in the RTT, the sources

Table 5.2. Values of parameters in the simulation scenarios

Simulation Cases	Explanation and Used Parameters
Buffer at router	• values: 150, 200
Number of nodes	• values: 60, 100
Wireless cases	• with 30 and 60 nodes, $buffer = 200$ • only FTP traffic • 5 ms bottleneck, 50 ms connection link delay
FTP	• always on
Dynamic FTP	• either on or off • dynamic change from t = 40 sec to t = 70 sec (from [30]) • Half of the nodes are randomly selected • stopped at random times starting from t= 40 sec • started at 70 sec again
CBR traffic	• either on or off • starts at a random time, continues for 30 sec. • Total number of CBR traffic nodes are 20. • The traffic is attached to the existing nodes. • Rate = 384 Kbps
WEB traffic	• 10 nodes (with 10 sessions each)
Bottleneck delay	• values: 5 ms, 125 ms. • Latter one simulates satellite links (value based on [25]). • Delay is much larger
ECN bit	• either set or not set. Mostly off.

receive the ACKs for their sent packets later than the first case, hence they keep a RTT value higher than the first one and increase their cwnd's more slowly, meaning that less packets are transferred for the same simulation time leading to not so heavy congestion as in the first case.

Table 5.3. Jain's fairness index for scenarios, where $buffer = 150$

scenario	$Buffer = 150$ and dynamic FTP traffic, ECN not set						
	REM	RED	ARED	DFC	AFC	BLUE	PI
100 nodes and 5 ms of bottleneck delay							
CBR off	0.89	0.66	0.70	0.89	0.81	0.89	0.76
CBR on	0.77	0.74	0.62	0.84	0.79	0.82	0.80
CBR remains on, bottleneck delay changed							
125 ms	0.82	0.81	0.77	0.77	0.78	0.87	0.75
CBR remains on, nodes changed to 60							
5 ms	0.84	0.78	0.78	0.90	0.88	0.92	0.82
125 ms	0.88	0.89	0.92	0.87	0.86	0.92	0.86

Table 5.4. Total packet drops for scenarios, where $buffer = 150$

scenario	$Buffer = 150$ and dynamic FTP traffic, ECN not set						
	REM	RED	ARED	DFC	AFC	BLUE	PI
100 nodes and 5 ms of bottleneck delay							
CBR off	6679	7192	7294	3751	3412	6180	3115
CBR on	13636	12159	11981	10500	10468	12936	10794
CBR remains on, bottleneck delay changed							
125 ms	11750	11418	11658	8490	8516	11340	8488
CBR remains on, nodes changed to 60							
5 ms	10630	10409	10894	8078	7637	11066	7810
125 ms	9223	8190	8473	5751	5370	7726	5497

5.1.2. Change in the Buffer Size

The reference value for the queue length, q_{ref}, is set to a value very close to the buffer size in order to increase the usage of the resources at the router. The change in the buffer size is implemented to examine the ability of the AQM mechanism to keep track of the changed, new q_{ref}. The offered mechanisms and the PI controller can keep track of the reference value easily, whereas other mechanisms RED, REM,

Table 5.5. Jain's fairness index for scenarios, where $buffer = 200$

scenario	Buffer = 200, if not said, ECN = 0						
	REM	RED	ARED	DFC	AFC	BLUE	PI
100 nodes and 5 ms of bottleneck delay							
FTP only	0.89	0.66	0.73	0.94	0.93	0.97	0.89
FTP changed to dynamic traffic							
ECN = 0	0.88	0.75	0.70	0.86	0.86	0.93	0.86
ECN = 1	0.86	0.56	0.75	0.87	0.89	0.93	0.84
CBR traffic is added							
ECN = 0	0.78	0.62	0.64	0.77	0.82	0.83	0.77
ECN = 1	0.80	0.58	0.65	0.84	0.85	0.91	0.83
WEB traffic is added							
ECN = 0	0.68	0.58	0.55	0.76	0.72	0.75	0.74
ECN = 1	0.68	0.52	0.58	0.75	0.79	0.83	0.79
bottleneck delay changed to 125 ms, WEB traffic is cancelled							
ECN = 0	0.82	0.80	0.79	0.79	0.86	0.81	0.87
ECN = 1	0.77	0.81	0.81	0.77	0.86	0.85	0.87
WEB traffic is added again							
ECN = 0	0.74	0.77	0.80	0.70	0.75	0.77	0.70
Dynamic FTP, CBR and WEB traffics remain, nodes changed to 60							
5 ms	0.72	0.63	0.68	0.78	0.80	0.74	0.72
125 ms	0.78	0.79	0.75	0.79	0.80	0.80	0.75
Wireless Scenario, FTP only (not dynamic), bottleneck delay = 5 ms							
30 nodes	0.98	0.97	0.98	0.99	0.98	0.98	0.98
60 nodes	0.97	0.92	0.93	0.98	0.97	0.98	0.97

ARED and BLUE cannot keep track of the new reference value. The BLUE mechanism performs somewhat better, however the introduce of dynamic FTP traffic causes the queue length maintained by it to drop to 0.

Table 5.6. Total packet drops for scenarios, where $buffer = 200$

scenario	Buffer = 200, if not said, ECN = 0						
	REM	RED	ARED	DFC	AFC	BLUE	PI
100 nodes and 5 ms of bottleneck delay							
FTP only	7341	7484	7955	3592	3291	5121	3209
FTP changed to dynamic traffic							
ECN = 0	6469	7399	7265	3073	2777	4712	2735
ECN = 1	6578	6445	7623	2296	2628	619	2642
CBR traffic is added							
ECN = 0	12871	12190	12577	9926	9778	13377	10024
ECN = 1	13093	11862	11711	9598	9631	11603	9676
WEB traffic is added							
ECN = 0	13258	11391	11791	9957	9367	13561	9787
ECN = 1	13304	11134	12452	9755	9671	12188	9905
bottleneck delay changed to 125 ms, WEB traffic is cancelled							
ECN = 0	11558	11161	11762	7437	7374	11027	7844
ECN = 1	11620	11472	11817	7365	7084	10249	7824
WEB traffic is added again							
ECN = 0	11878	11456	11747	7631	7421	11169	7417
Dynamic FTP, CBR and WEB traffics remain, nodes changed to 60							
5 ms	10876	10051	10153	6715	7139	9662	6997
125 ms	9162	8366	8610	5443	4692	8012	5203
Wireless Scenario, FTP only (not dynamic), bottleneck delay = 5 ms							
30 nodes	2937	3412	3772	938	828	1080	837
60 nodes	5081	5517	6017	1901	1789	2504	1760

5.1.3. Change in the Number of Nodes

If the number of nodes is changed from 100 to 60, the load on the congested router decreases and as a result total number of packets decreases. The mechanisms also act more fair towards the sources.

5.1.4. Wireless Scenarios

The wireless scenarios are implemented with two different number of nodes, 30 and 60. We are aware of the fact that the wireless scenarios do not reflect the real world implementations in the sense that mobile nodes initiate FTP traffic. However, the scenarios are added to examine the behaviour of the AQM mechanisms in a wireless environment, they do not have any intention to imitate the real world. The scenarios show that AFC obtains best result in terms of packet drop, followed by the PI controller and DFC one. Other mechanisms perform unnecessary drops, which is an unwanted situation for the mobile environment since the easiness of the loss of the packets even without a dropping mechanism.

5.1.5. Effect of the ECN Bit

Setting ECN bit enables to inform the sources about the incipient congestion with marking the packets. Hence, one can expect that there will be less drops for the case, where ECN bit is set and other factors remaining constant. For some heavy loaded cases however, there are more drops, which can happen, since if the AQM mechanisms do not drop the packet arrived and only mark it, then this packet uses the necessary resource and from the high load, packets are dropped because of the buffer overflow. For those cases the AQM mechanisms seem not to benefit from setting the ECN bit. However, the AQM mechanisms start to behave more fair in almost every case if the ECN bit is set, provided that other factors being equal. This is also understandable since in most cases there are less drops and only marks.

5.1.6. Common Results in Different Scenarios

In almost all the simulations the AFC performs much better results during slow-start. This phenomenon can be verified if the graphs for bottleneck link utilization are carefully examined. To achieve fewer drop of packets and a better utilization during slow-start, the dynamics of the FMRLC in AFC are tuned to behave less aggressively compared to the DFC. This is the reason, why DFC leads to less oscillations around

q_{ref}. The FMRLC can be tuned to behave more aggressively and can show similar results to DFC by simply decreasing the value of *cautious* parameter. This way, AFC updates the center of the output membership functions before the previous update shows its whole effect. This can be beneficial in some cases, where the main concern is to keep track of a reference queue length. However, in our choice of parameters, this is not the case. The default offered parameters provide good results and also the packet drops are less.

The Figures following demonstrate the results for the different scenarios explained. For the ease of comparison, each metric obtained from AQM mechanisms are put in a different Figure. The Figures show that DFC and AFC methods outperform the others in most cases in terms of the degree of oscillation around q_{ref}, bottleneck link utilization (note that we have 2 Mbps link and a simulation time of 120 sec, meaning 100 per cent utilization is achieved with an aggregate flow of 240 Mb) and dropped packets. The deviation in the delay variation is also less for AFC especially, meaning that flows are more equally treated. This is another way of looking to the fairness, which is from traffic point of view. Jain's fairness index shows that BLUE is the most fair among the mechanisms, which is from the load point of view, however it usually makes big oscillations around the reference point and it drops more packets than the AFC or DFC mechanism except for the case, where there is only FTP traffic and ECN bit is set. If CBR traffic is added onto it, BLUE shows again bad results. Another important observation is that RED and ARED settle down to a reference value for the queue length, however they perform to much drops compared to the PI, DFC and AFC and the reference value that they settle down to is not the reference value we want.

5.2. Detailed Examination of the Results Obtained

The objective of this section is to examine the results in detail in order to compare the behaviors of different AQM mechanisms in different environments. The main scenario consists of a buffer size of $buffer = 200$ and 100 nodes with 5 ms of bottleneck link delay and FTP traffic. Other scenarios are derived from this one using some

modifications and adding new traffic.

5.2.1. Scenarios with FTP Traffic Only and Bottleneck Delay of 5 ms

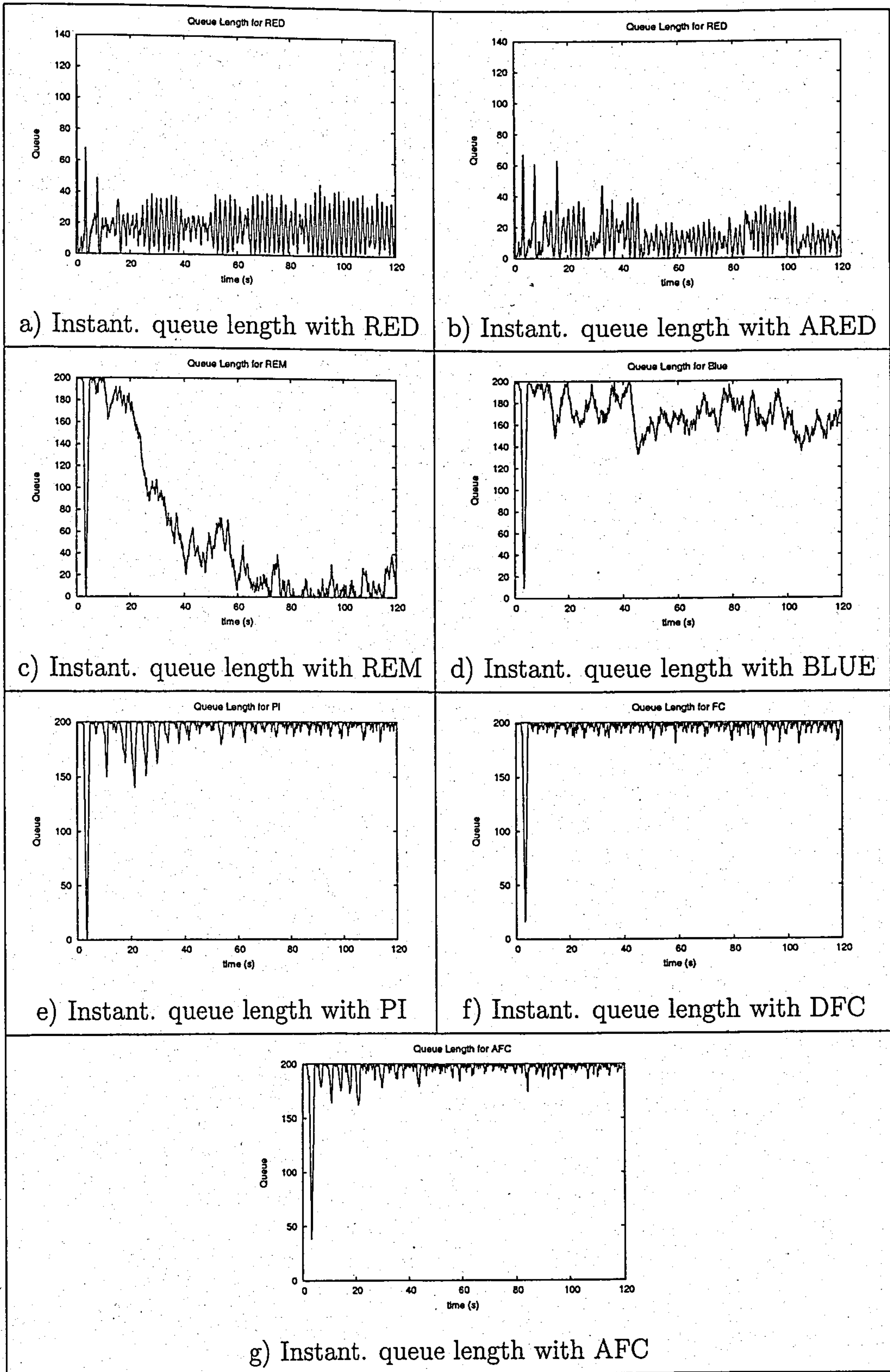
There are two different cases to study, one being FTP traffic starting at time t equals zero and ends at the end of the simulation, and the other one being so-called "dynamic FTP traffic", in which random half of the nodes stops transmission at time $t = 40$ and then starts transmitting again at $t = 70$. There is always a high load on the bottleneck queue, so there are always packet drops.

If we focus on the queue lengths in Figure 5.2, DFC and AFC provides the least fluctuations and DFC is even better to track the reference value, since it acts more aggressive and total number of packets dropped are more than AFC, which is shown in Table 5.6. PI is similar to the proposed algorithms for metrics of packet drops and queue length tracking, however its lack in slow start because of the assumptions in the design procedure can be seen in the bottleneck link utilization in Figure 5.3. Other AQM algorithms, RED, ARED, REM and BLUE perform worse than the DFC, AFC and PI for every metric we studied. Only BLUE is a bit better and can keep track of the reference value to some extent. Other mechanisms have difficulties in approaching to the reference value, they even result in instantaneous queue lengths of zero. For the fairness, we actually have two metrics to compare, one being Jain's fairness index, which focuses on a load fairness, and Figures showing average individual delays for each flow, like Figure 5.5. If there is less variation in the delay, then this means the AQM mechanism acts more fairly towards the connections that generated the traffic, meaning we focus on a fairness from a traffic point of view. Jain's fairness index suggests that BLUE is acting most fairly, however BLUE makes unnecessary drops and its delay variation is much than DFC and AFC. PI is worse in terms of delay variation. Other remaining mechanisms cannot compete with the ability of the proposed mechanisms in capturing the dynamics of the network.

Other scenarios with FTP traffic only are the ones with added dynamic behavior, one with ECN bit set and one not. If we look first at the effect of added dynamic

FTP behavior (Figure 5.6 - 5.9), we observe that RED, ARED and REM have even more fluctuations in the queue length and BLUE has difficulty to keep track of the reference value for the times when dynamic behavior is induced. The discussion for the network utilization, dropped packets and queue length remains the same for PI, DFC and AFC. Jain's fairness index suggests again being BLUE more fair, however, the effect of induced dynamic behavior shows itself in the large fluctuation in the delay metric for the individual flows. Average delay deviation for the individual flows is the least in AFC, meaning that it acts more fair than the others. In this scenario, PI has again difficulty in capturing slow-start dynamics.

BLUE and DFC seems to have the most benefits from setting the ECN bit. BLUE shows very few packet drops compared to the other mechanisms. However, there is less traffic on the router, if we use BLUE mechanism, which can also be seen from the queue lengths in Figure 5.10. BLUE has again difficulties in the dynamic region and the queue length drops to about 90 packets. However, in Figure 5.11, the bottleneck utilization for it seems to be near 100 per cent. The reason that there is less traffic on the router, if it uses BLUE algorithm and ECN bit is set, can be that the packets marked reach the destination and their corresponding ACKs reach the sources, before buffer overflows. This yields in reporting the sources of the incipient congestion and sources halve their congestion window in response to it. Therefore, there is less traffic during whole simulation and less total number of drops. DFC produces much less packet drops compared to the case, where ECN bit is not set. It also tracks the reference value much better. From the traffic fairness point of view, shown in Figure 5.13, again AFC acts more fair towards the connections. However, Jain's index is higher for the BLUE algorithm. Other points are similar to the case of ECN bit not set.



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

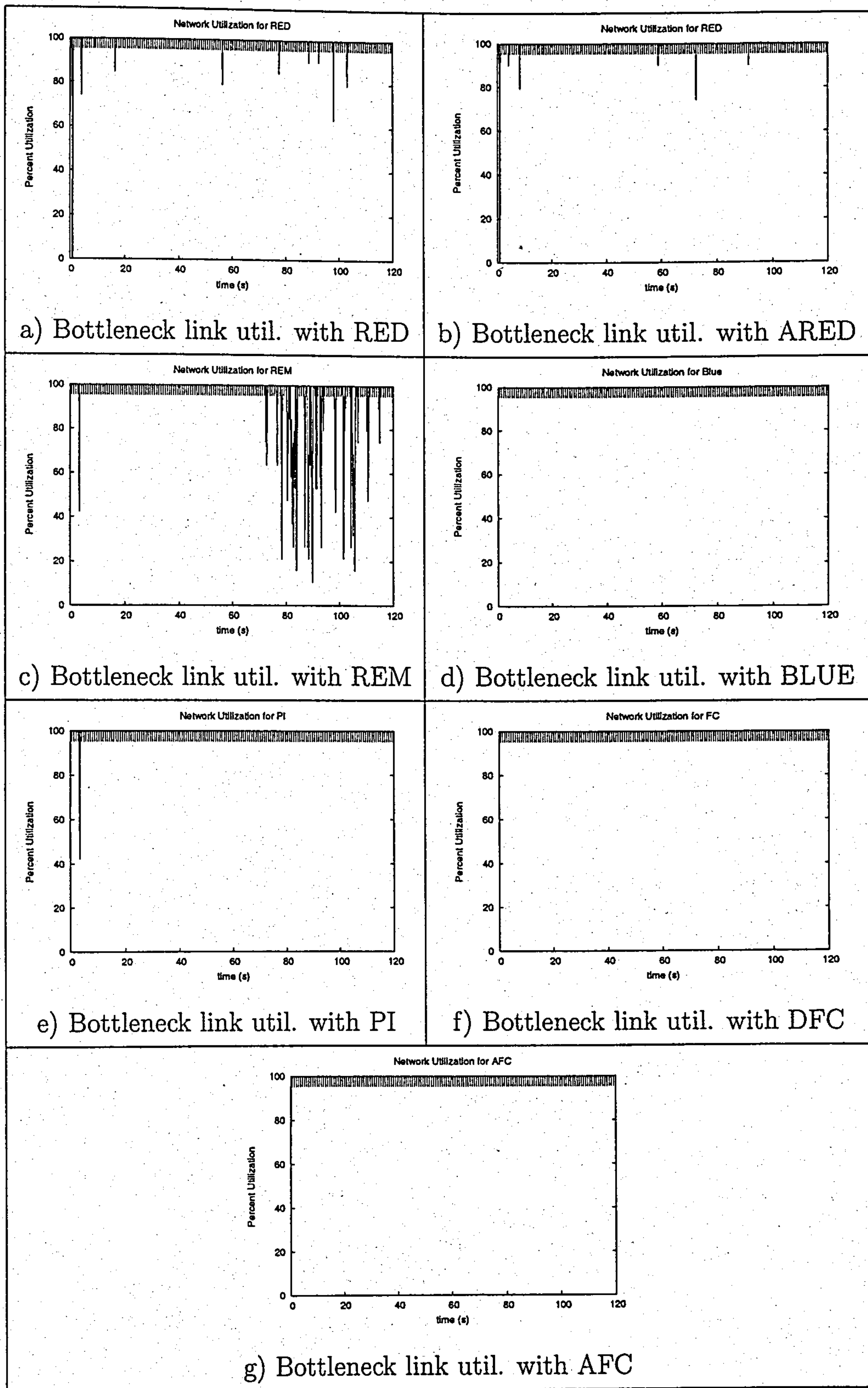
d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.2. Comparison of instantaneous queue lengths for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

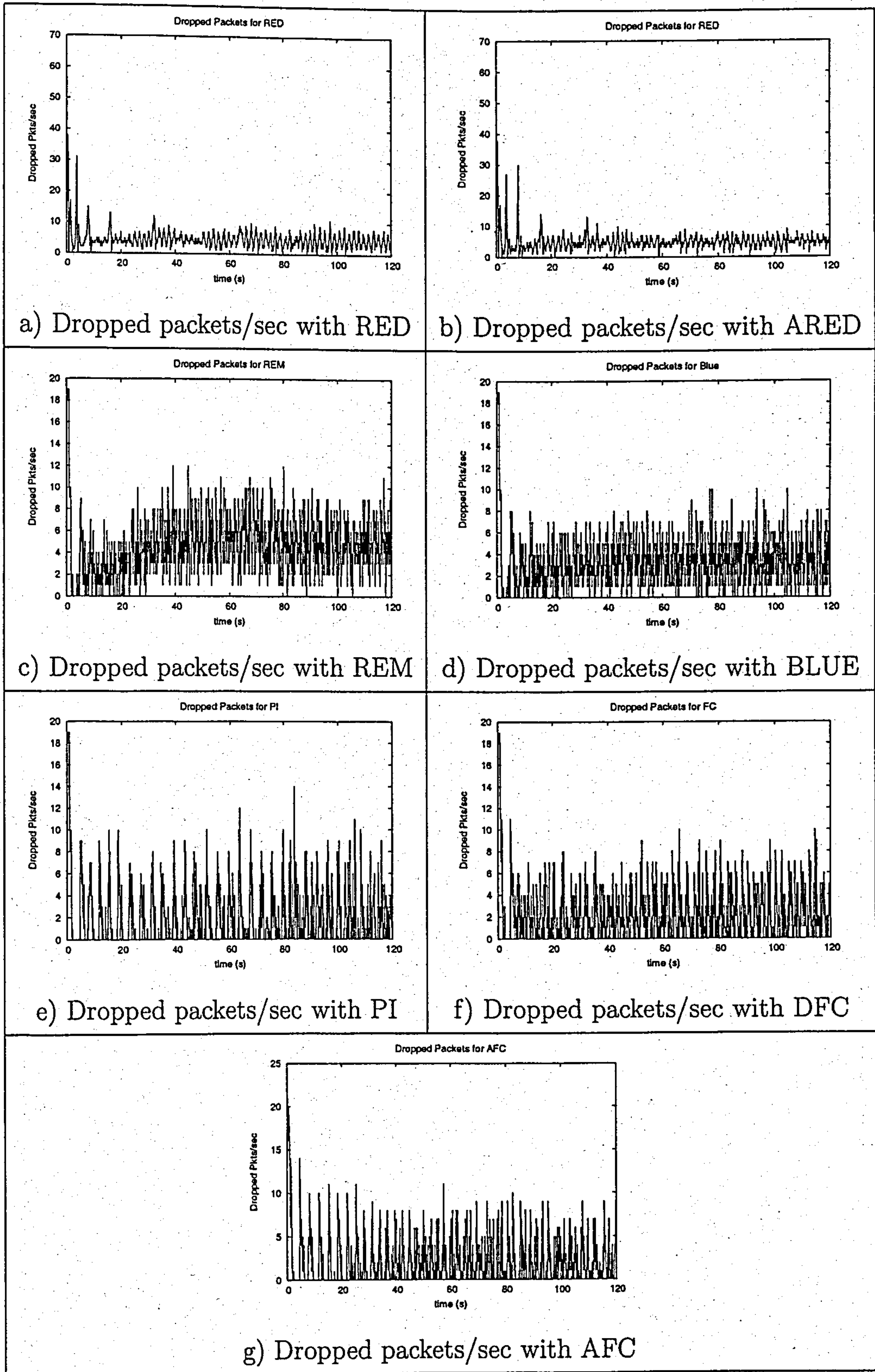
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.3. Comparison of bottleneck link utilization for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.4. Comparison of dropped packets/sec for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms

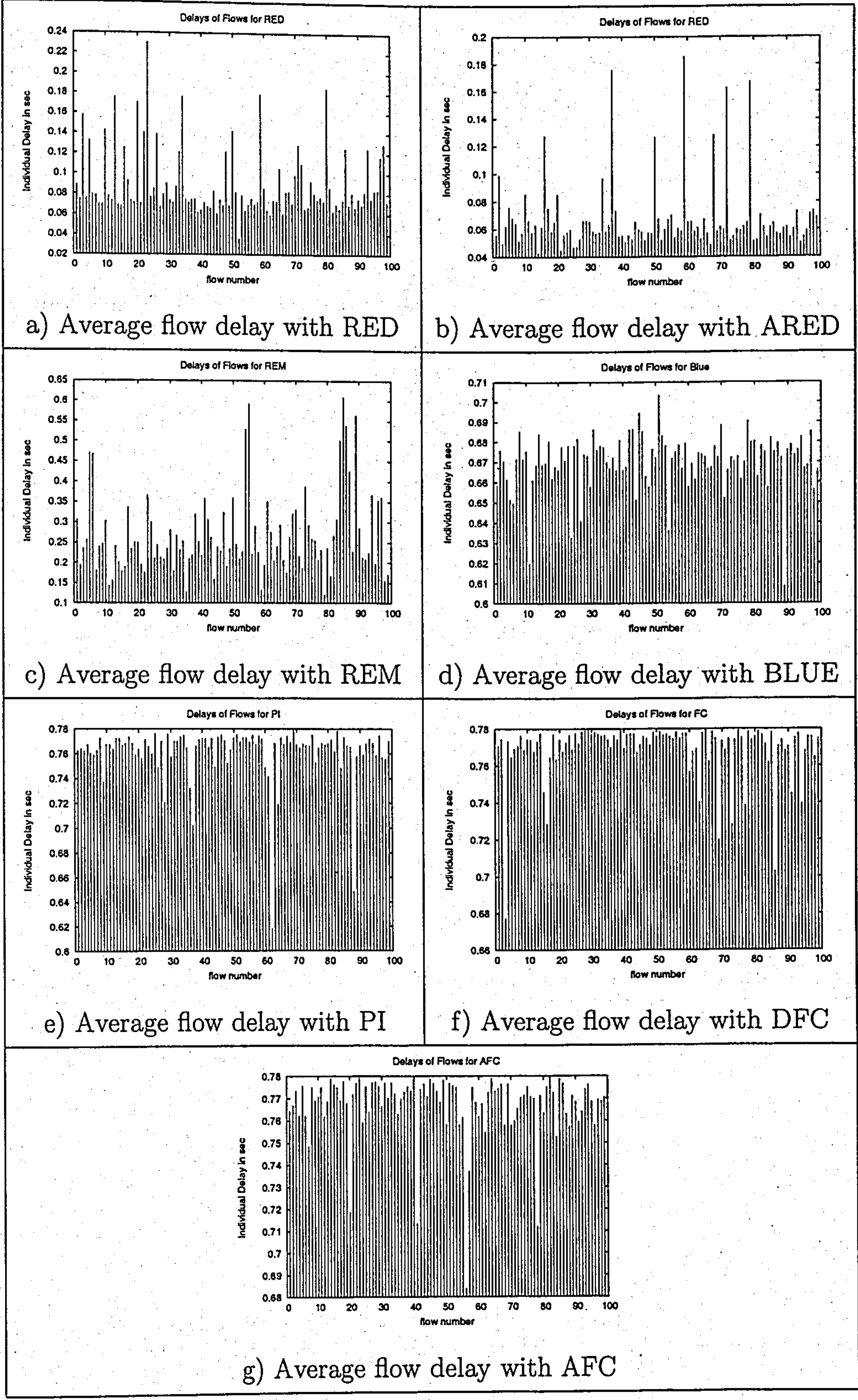


Figure 5.5. Comparison of average individual flow delays for FTP traffic only with $buffer = 200$, $nodes = 100$ and a bottleneck delay of 5 ms

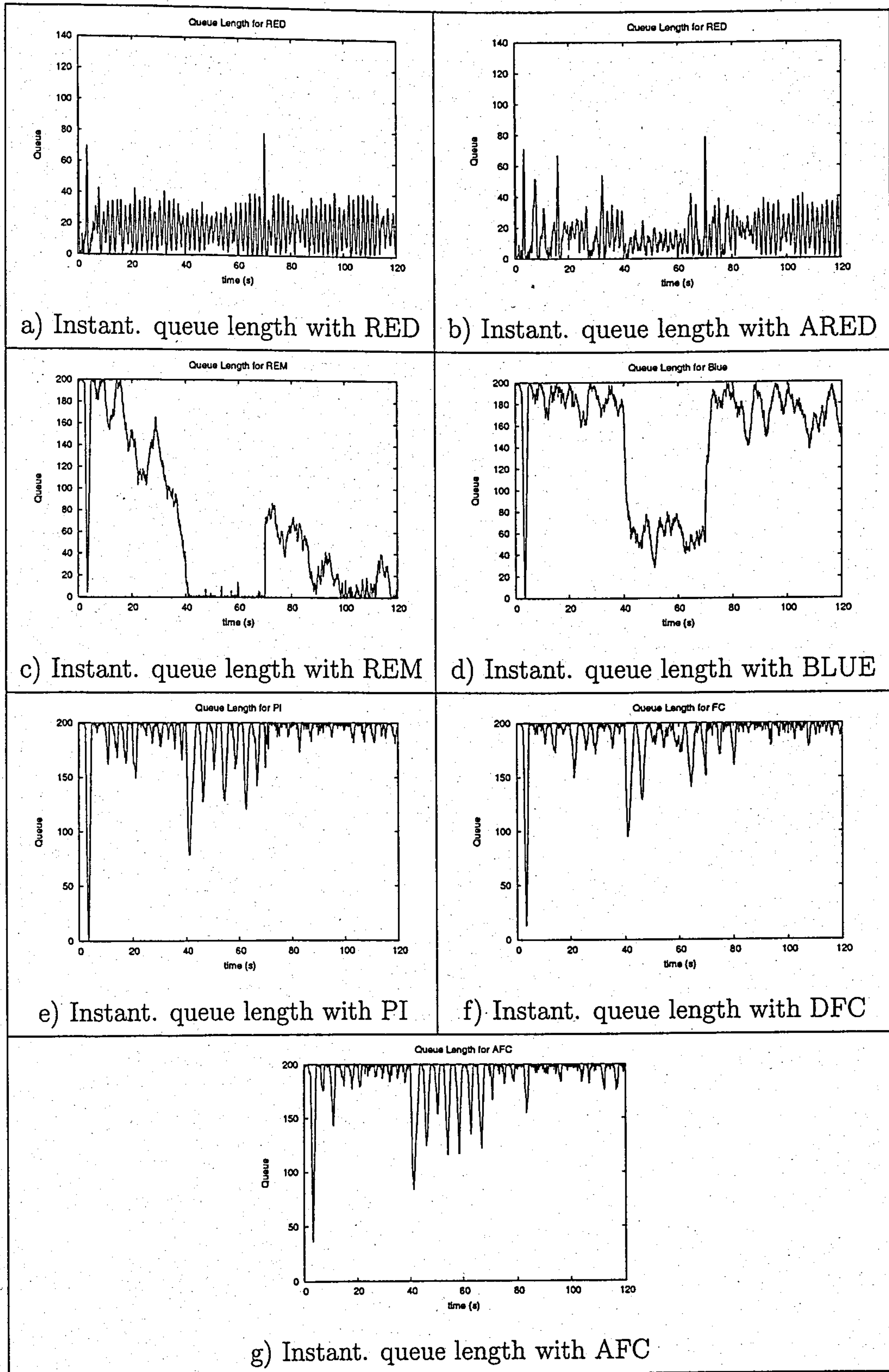


Figure 5.6. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic

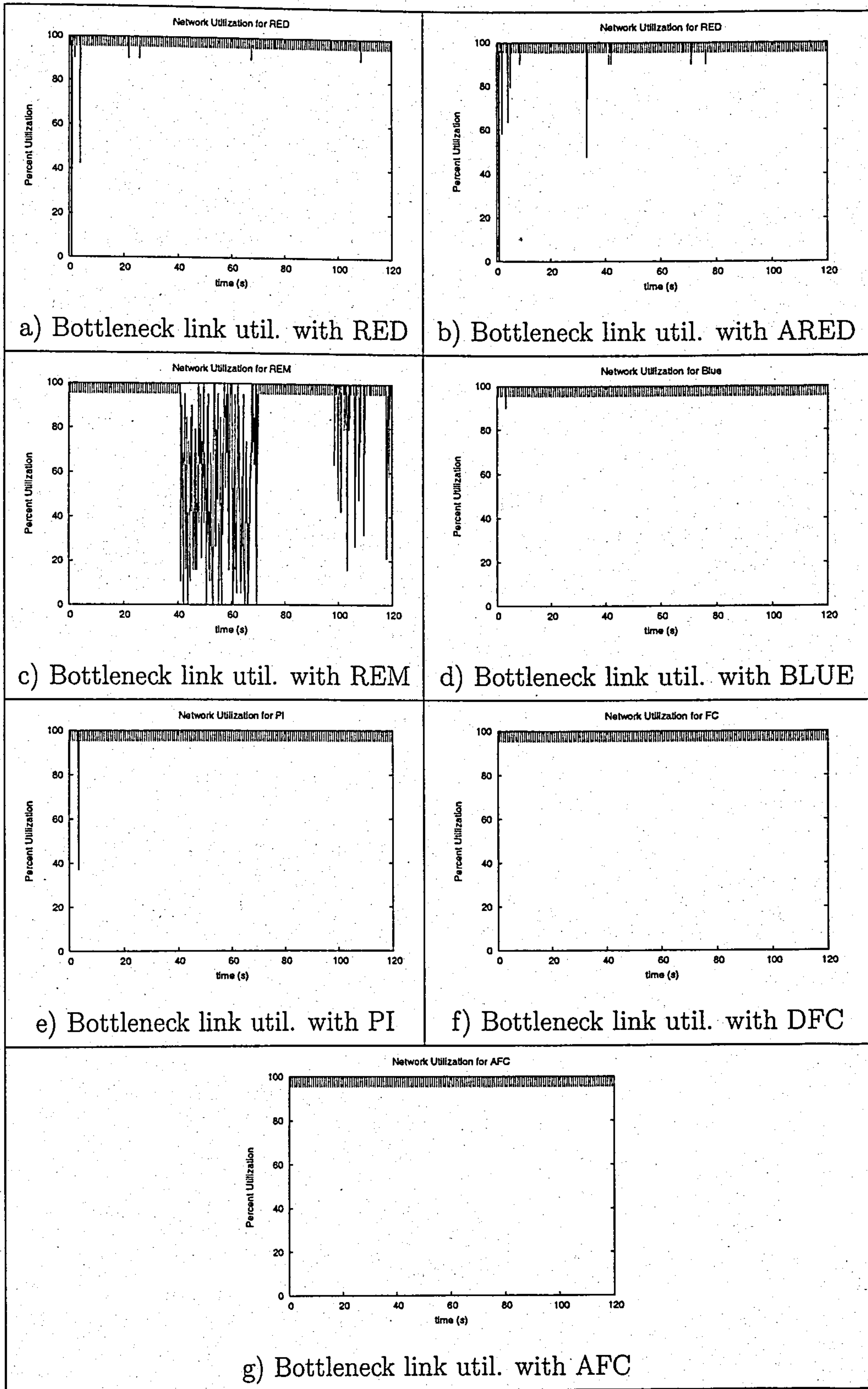
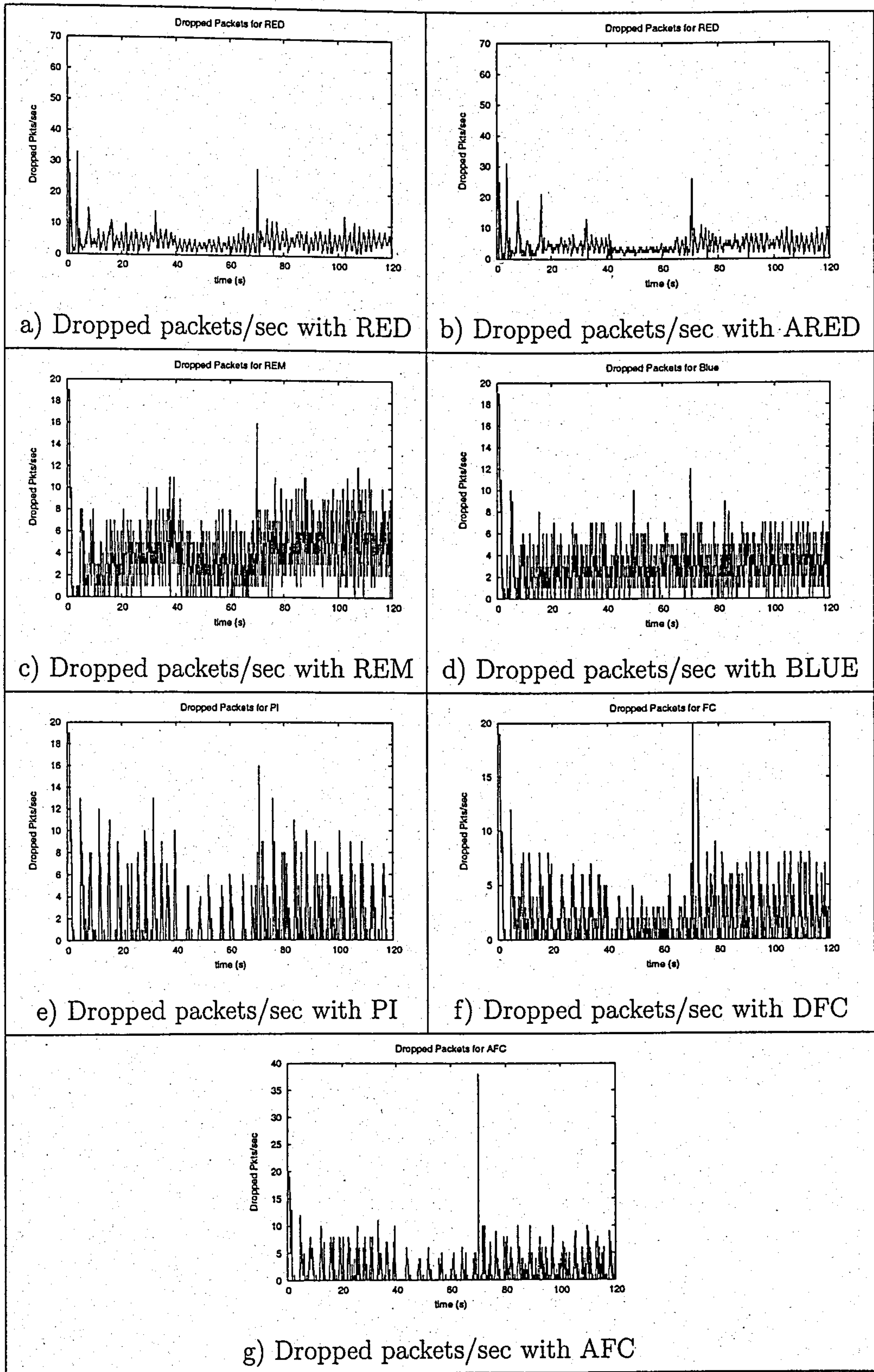


Figure 5.7. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.8. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic

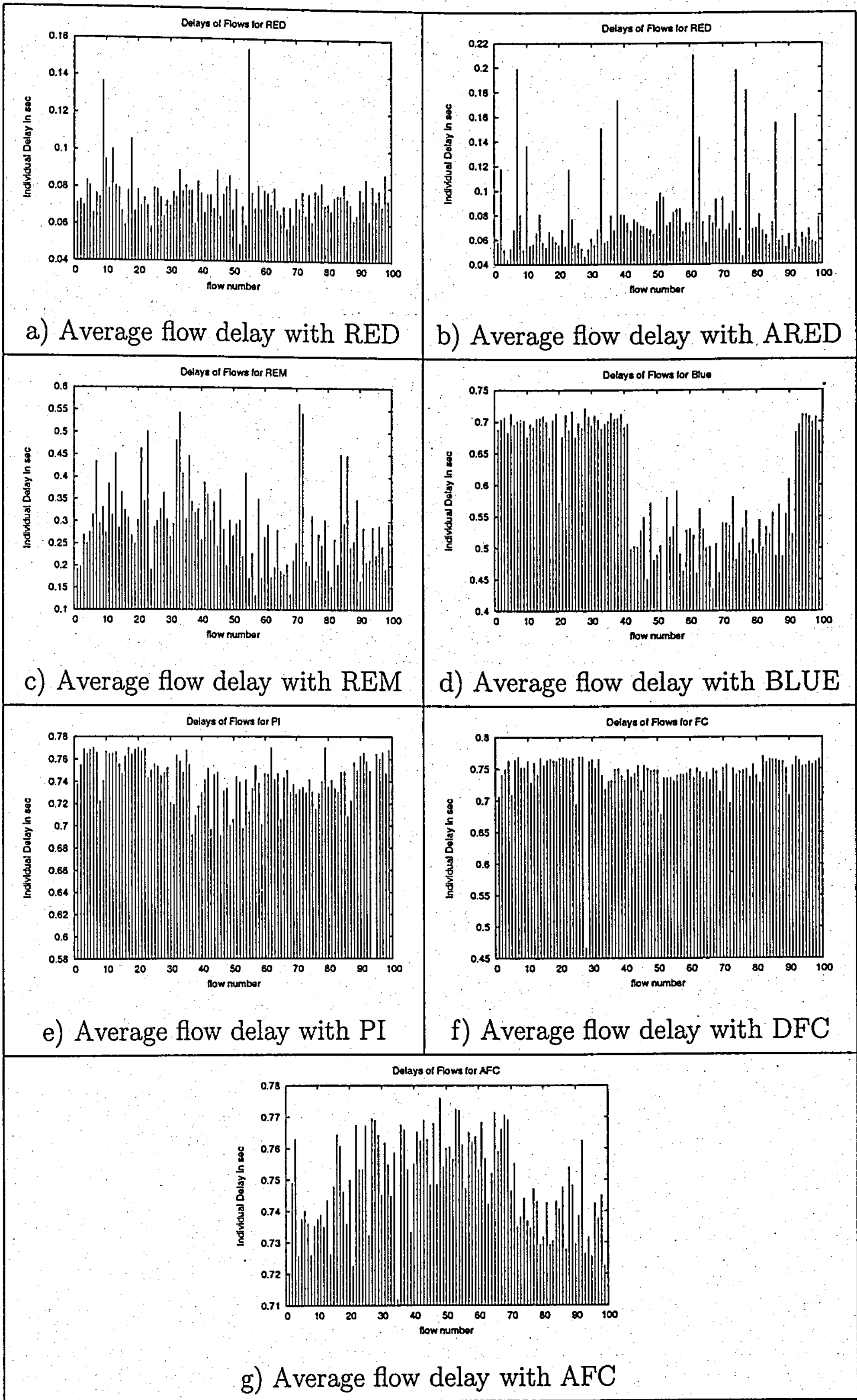


Figure 5.9. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic

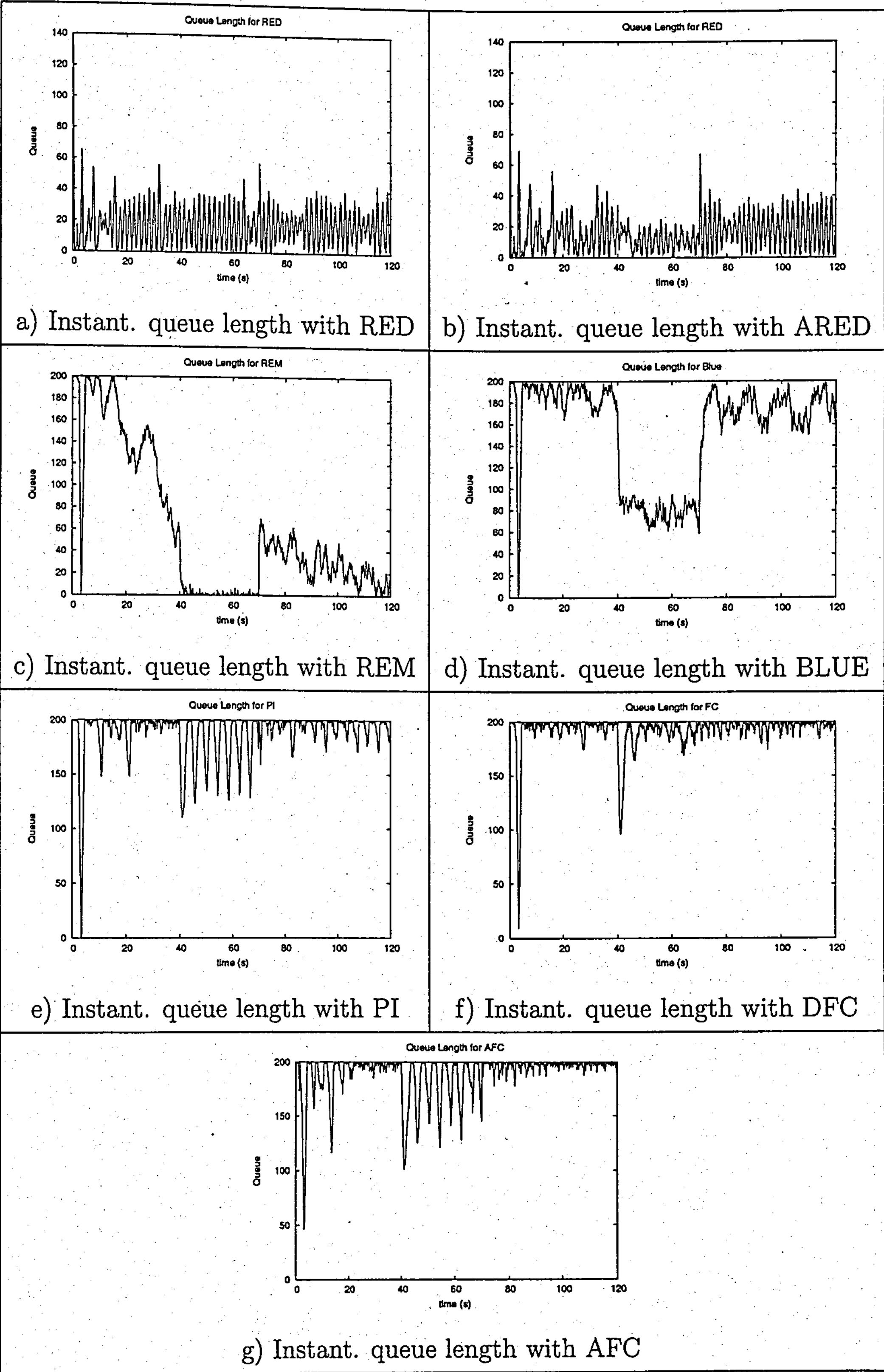


Figure 5.10. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set

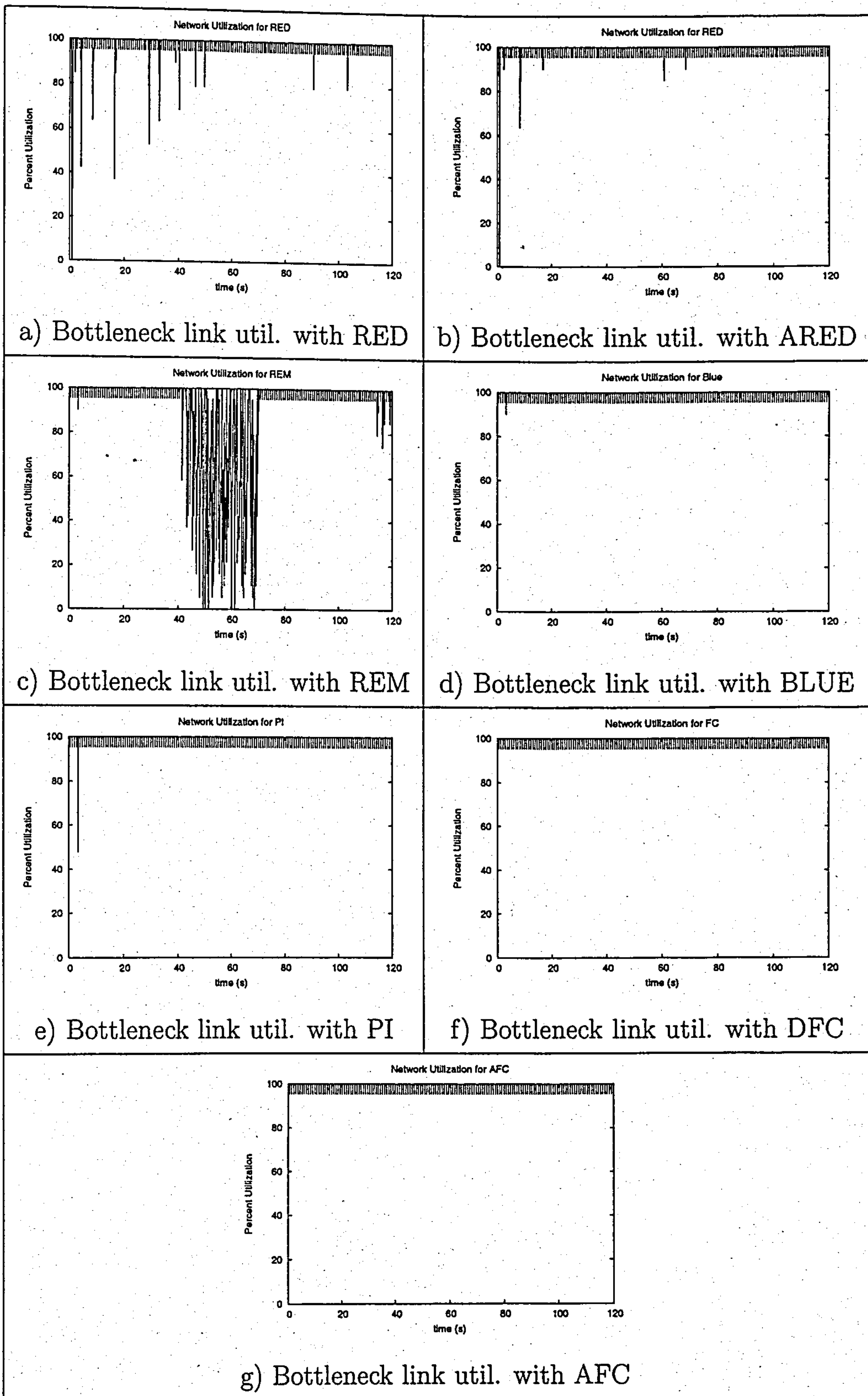
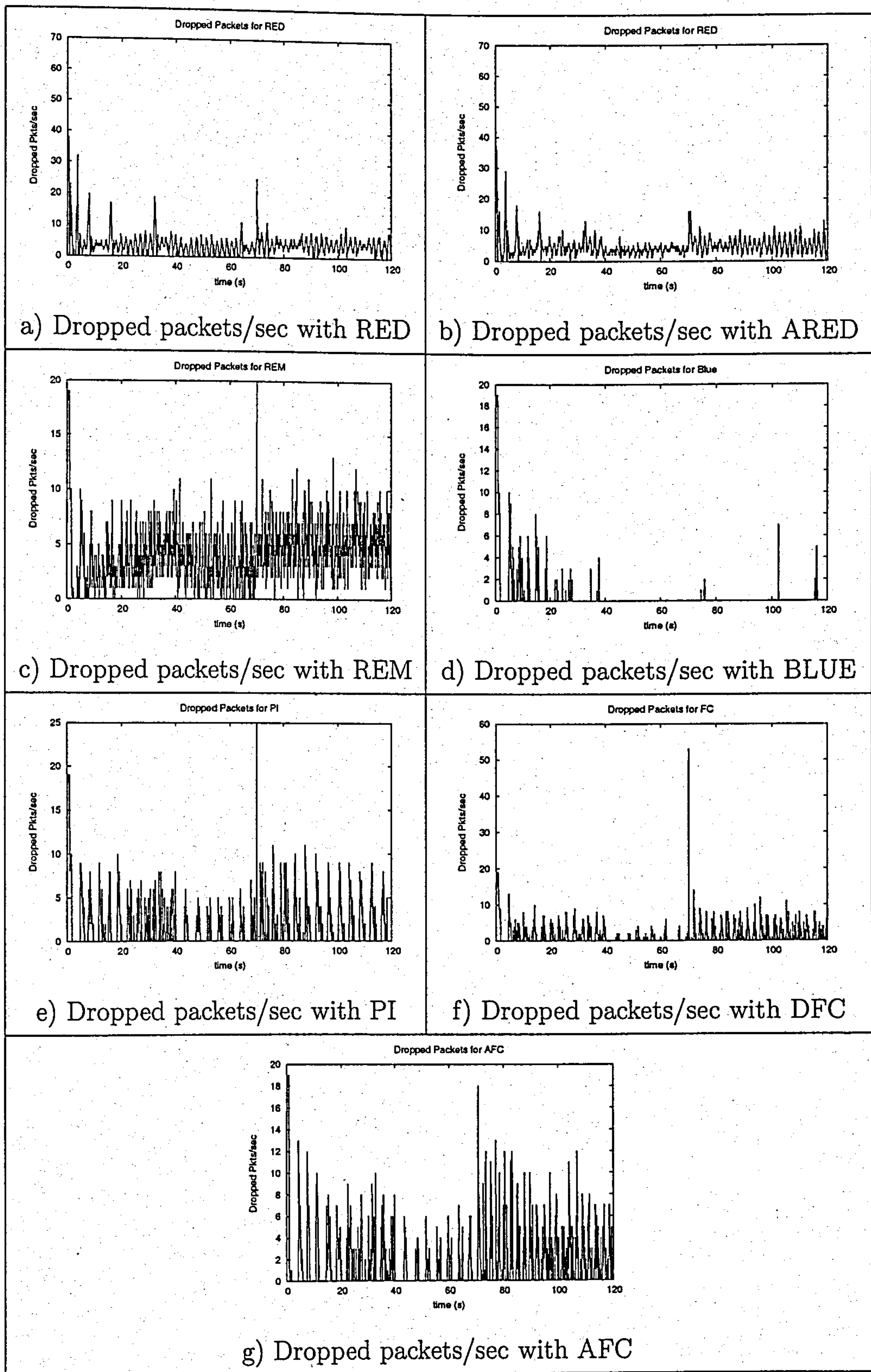


Figure 5.11. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

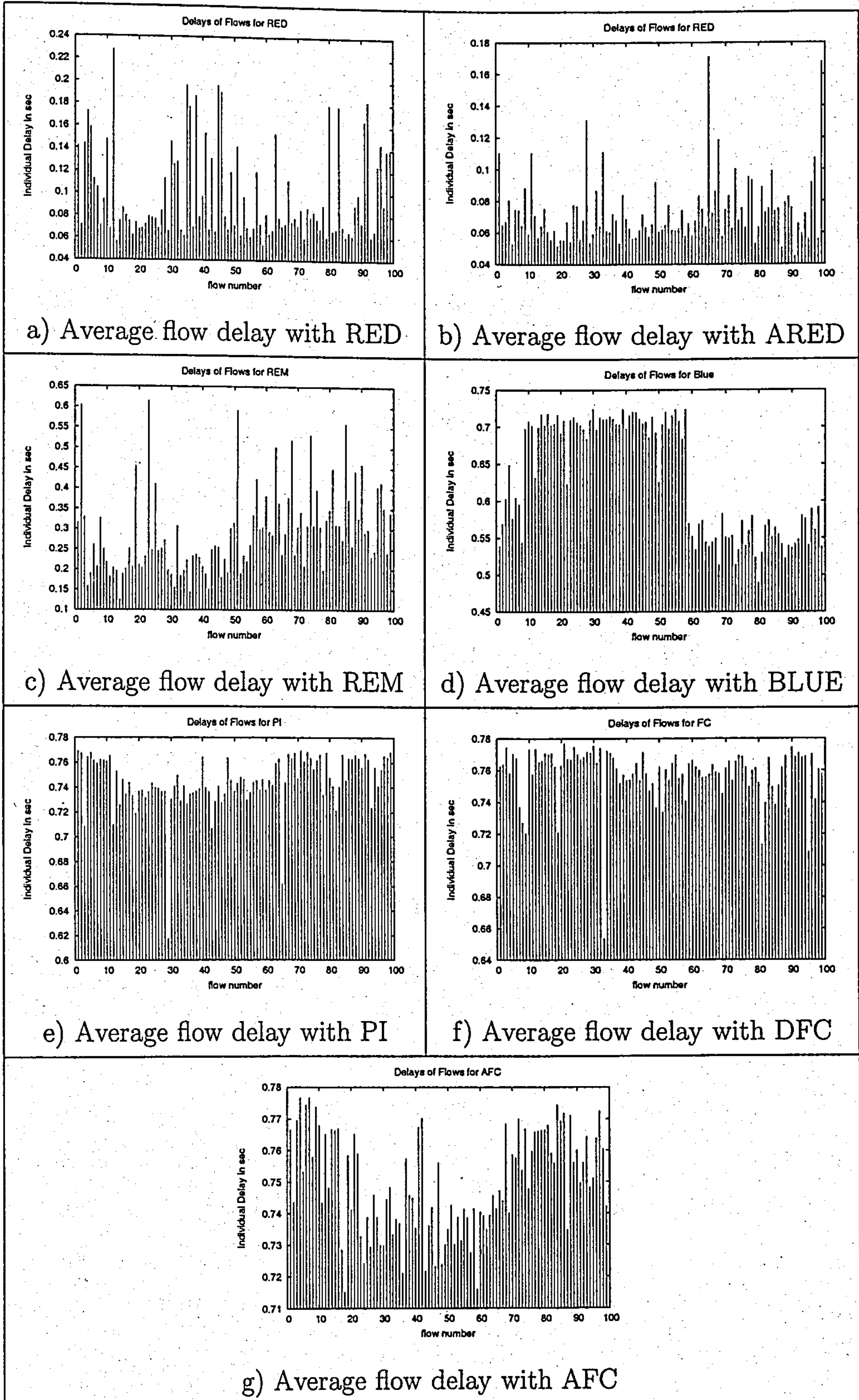
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.12. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

g) Average flow delay with AFC

Figure 5.13. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic, while ECN bit is set

5.2.2. Scenarios with Added Traffic and Bottleneck Delay of 5 ms

There are two types of traffic to be added, one being CBR (with a rate of 384 Kbps), which can simulate a low resolution video traffic as an example and a WEB traffic of 10 randomly started nodes with 10 randomly initiated sessions.

If we look at the effects of the added CBR traffic, we see that AFC keeps track of the reference value for the queue length better since it performs better during slow start, which is shown in Figure 5.14. BLUE acts according to the Jain's index more fair than the other AQM mechanisms. However, it performs worse than PI, DFC and AFC in terms of individual delay shown in Figure 5.17. PI, DFC and AFC show similar results with the delay variation, but Jain's index of AFC is higher among them. AFC performs also well in terms of dropped packets in dropping the least amount. Second is the DFC mechanism for the same metric. BLUE performs worse than the case without CBR traffic in terms of queue length tracking, packet drops and network utilization. RED, ARED and REM show more fluctuations in the queue length, REM being the worst among them.

When ECN bit is set, the effects are similar to the "FTP only" case. DFC and BLUE seem to show much better performance among all mechanisms compared to the case, where ECN bit is not set, which is mostly observable for BLUE for queue length tracking, shown in Figure 5.18. However, for this case with very high load on the congested router, DFC and AFC drop the least packets respectively. BLUE loses its advantage with the ECN bit in this type of highly loaded cases. The reason of it that the buffer start to overflow before the ECN bit performs its intended action.

RED and ARED settles down to a reference value for both cases, although this is not the reference set. REM performs worse than two also in terms of bottleneck link utilization, shown in Figure 5.15 and in Figure 5.19. The queue length of REM drops to zero and stays there most of time for the cases studied.

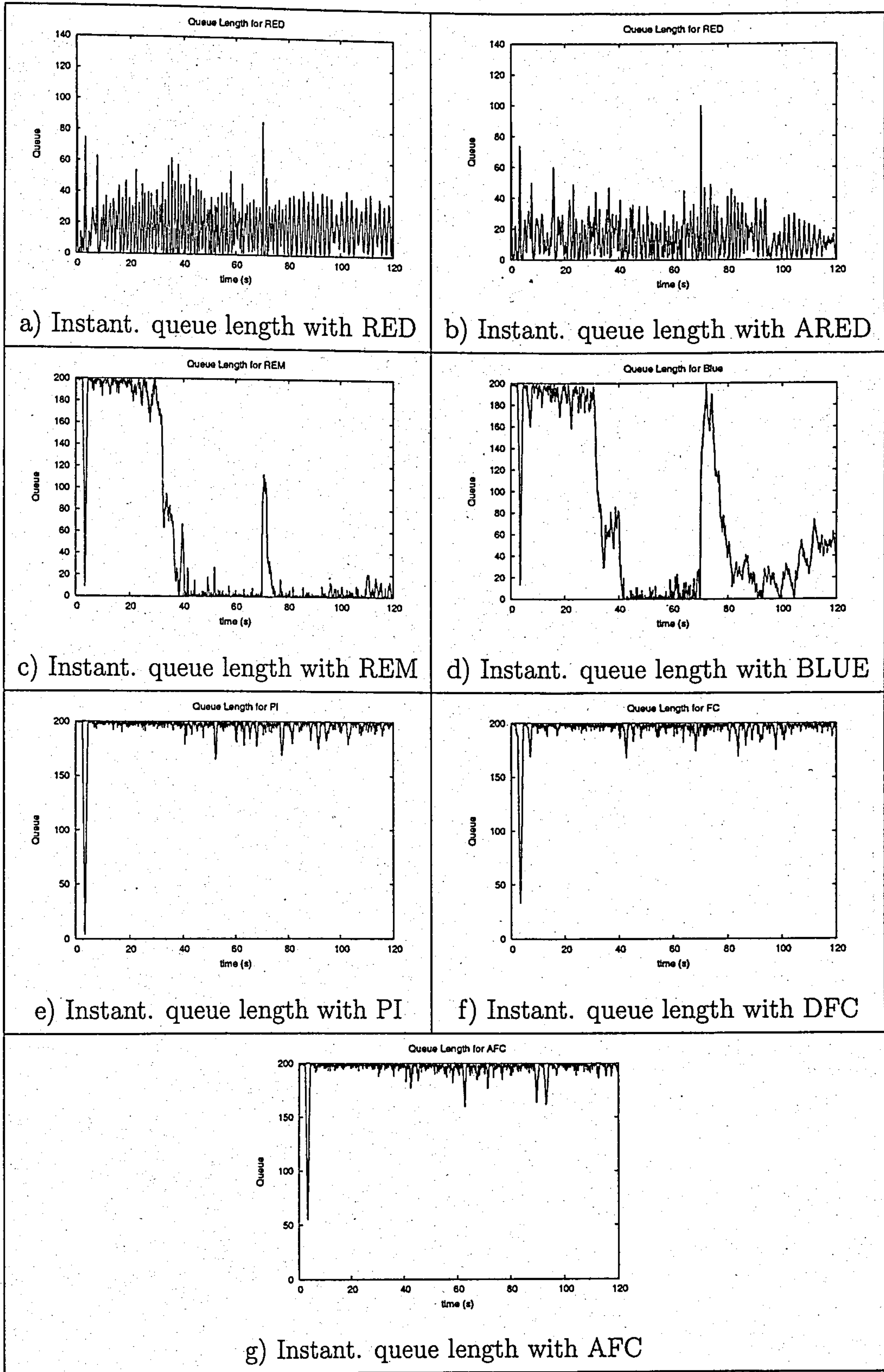


Figure 5.14. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic

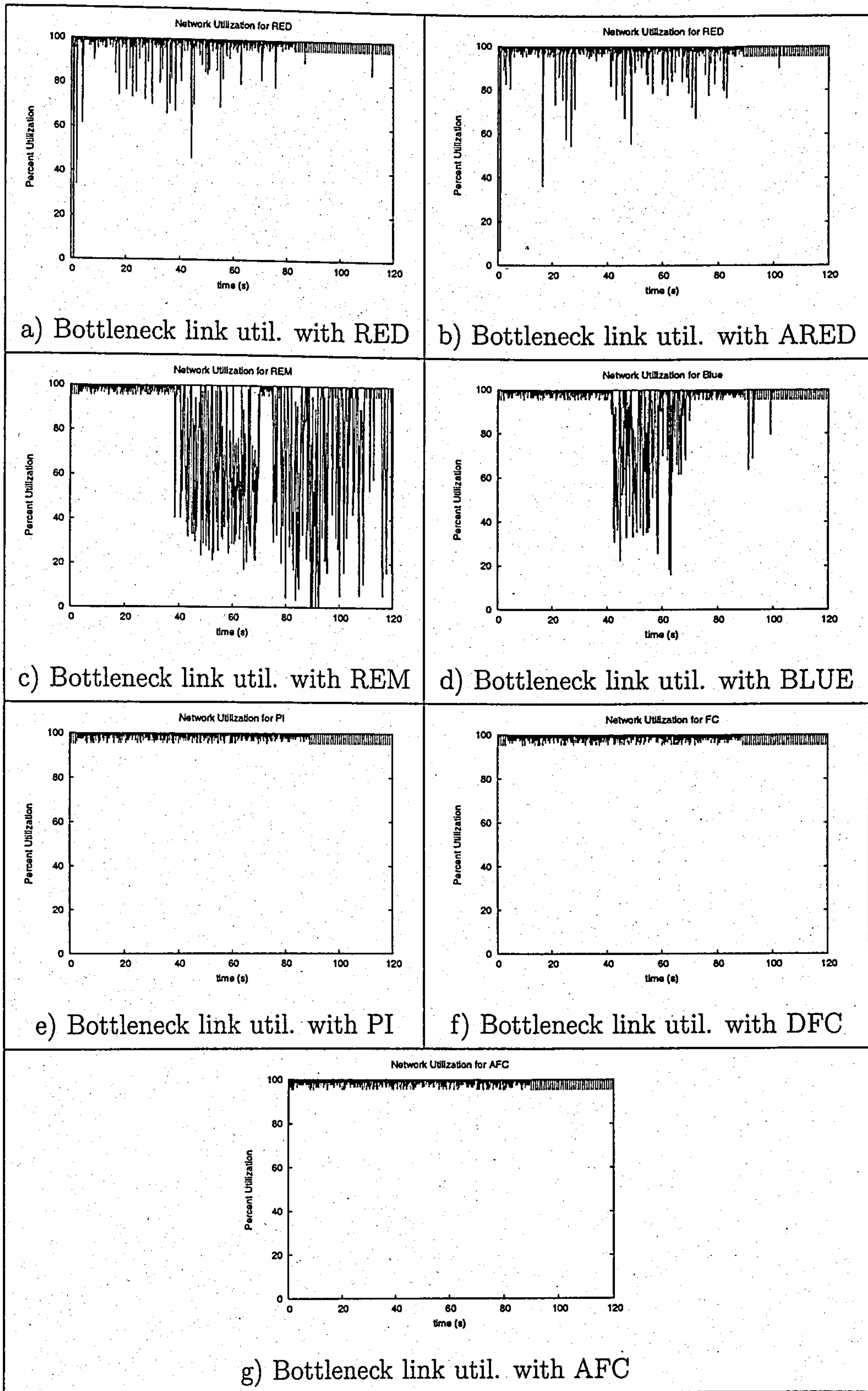
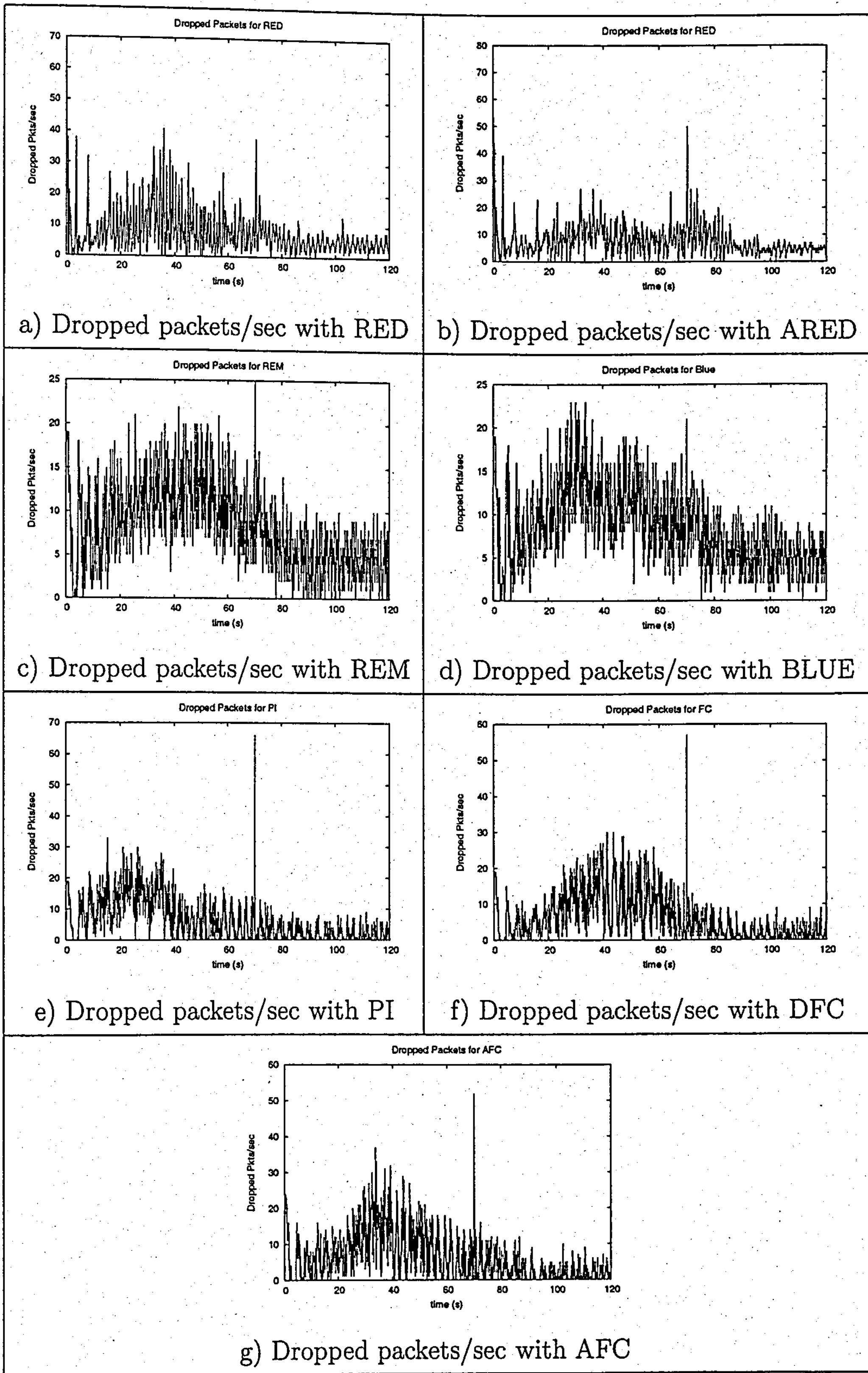


Figure 5.15. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.16. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic

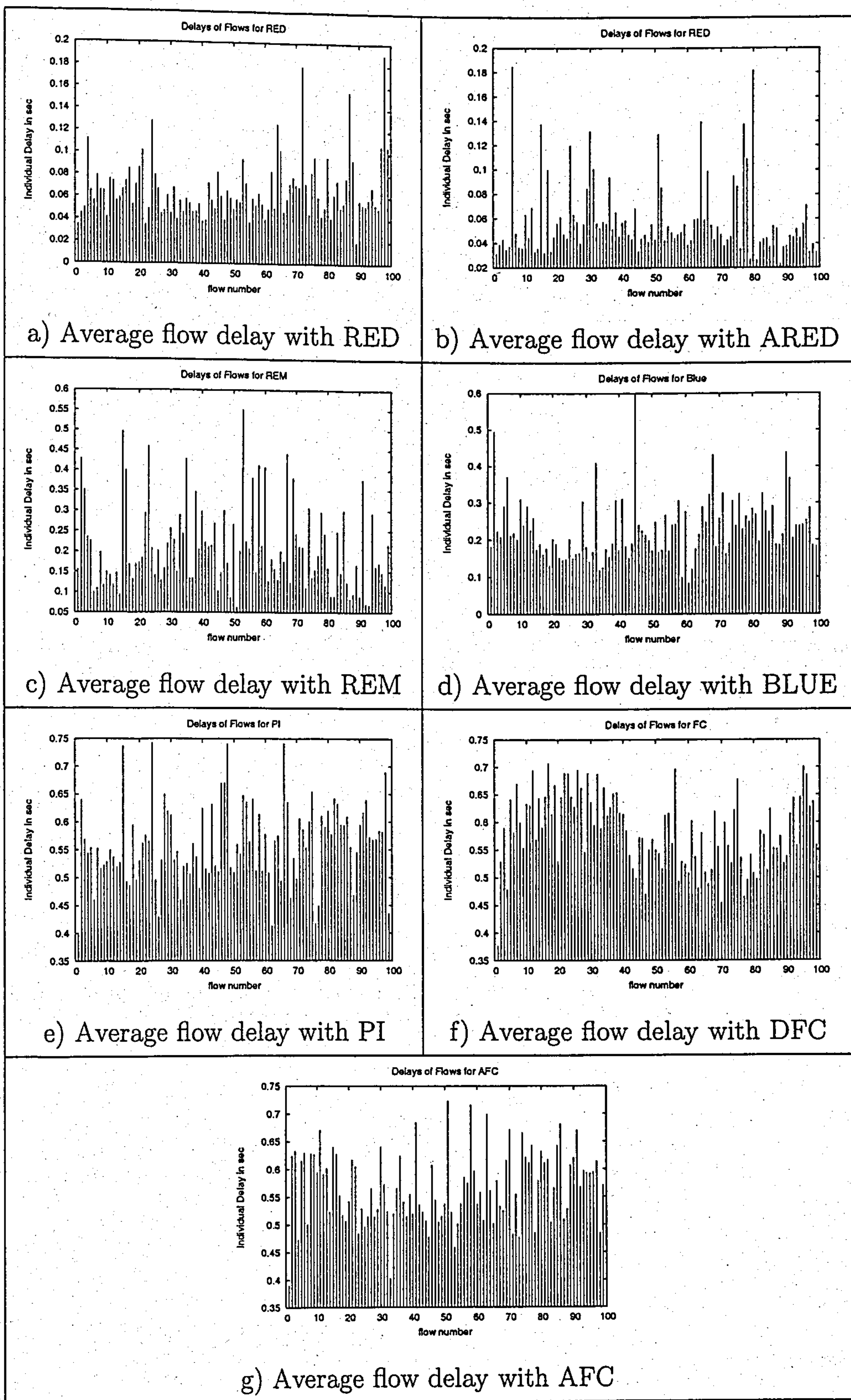


Figure 5.17. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic

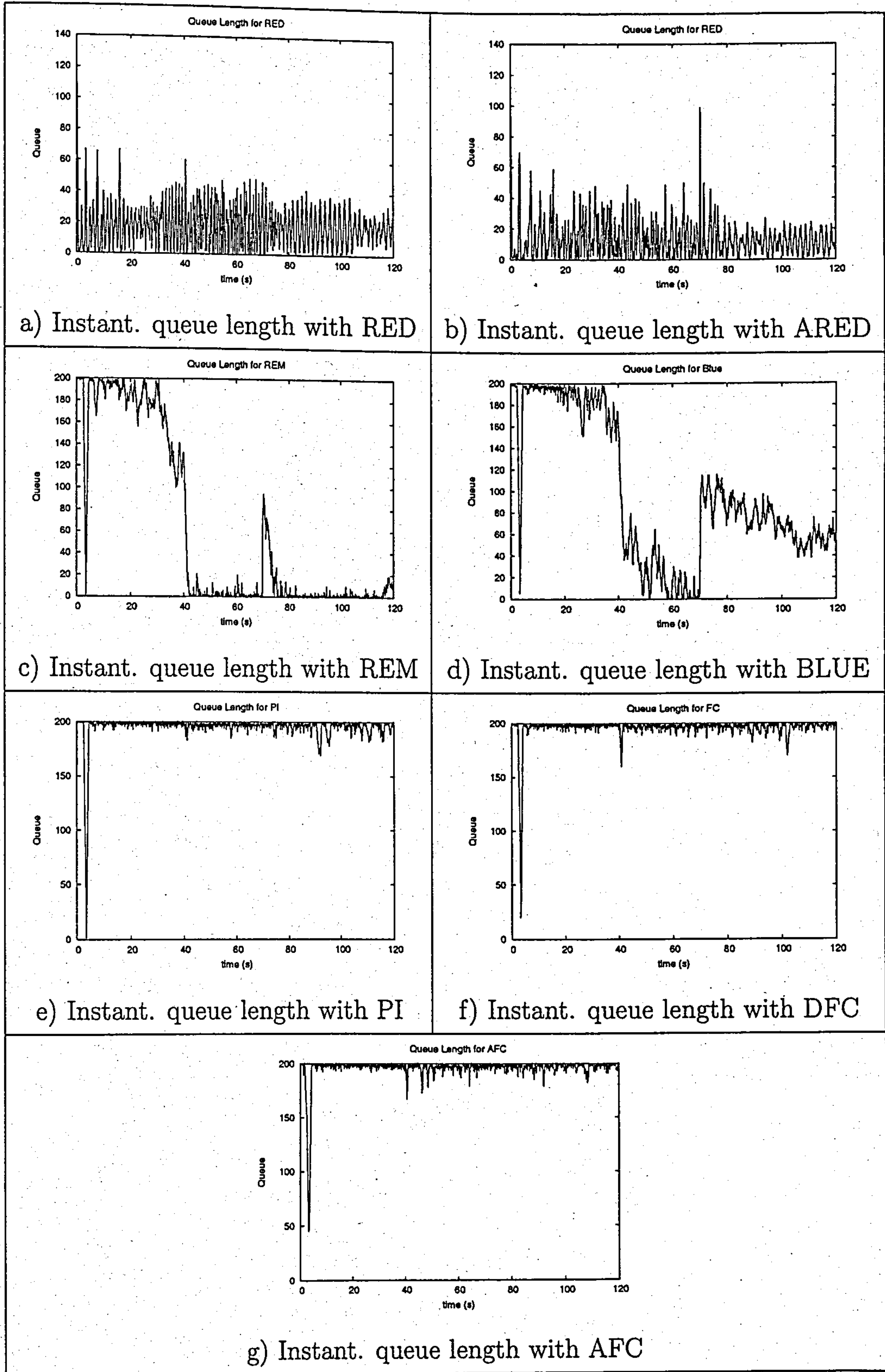
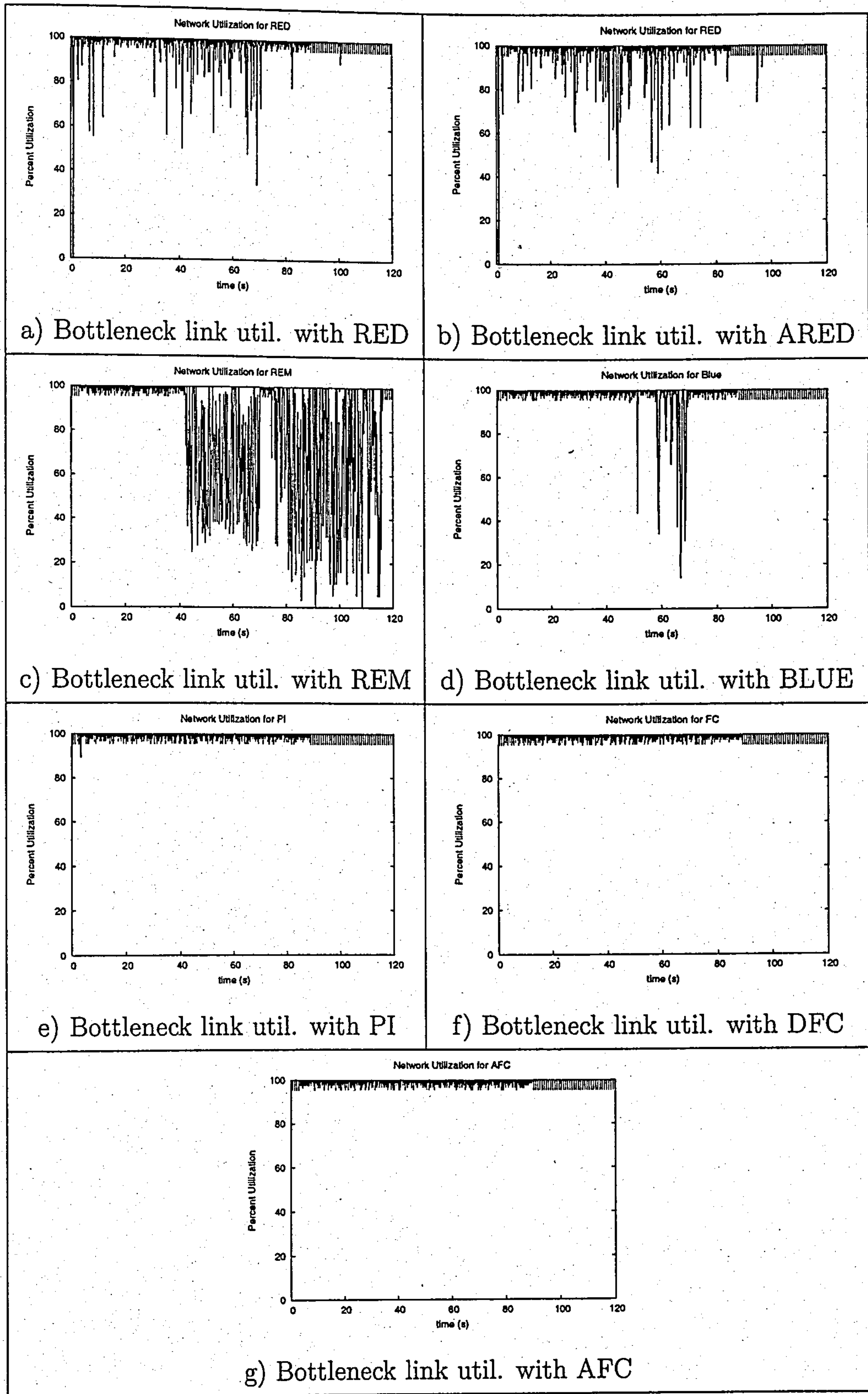


Figure 5.18. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.19. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set

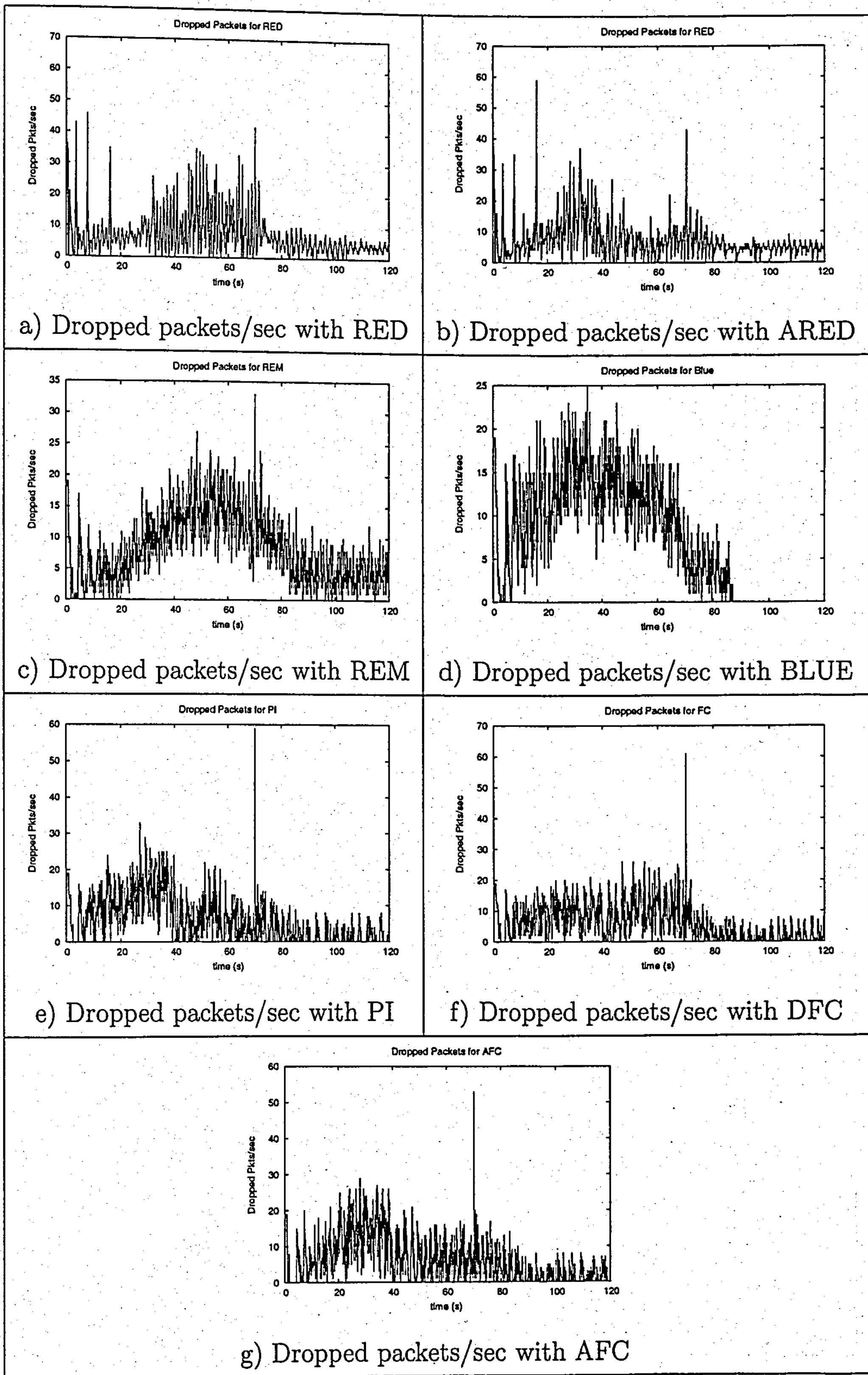
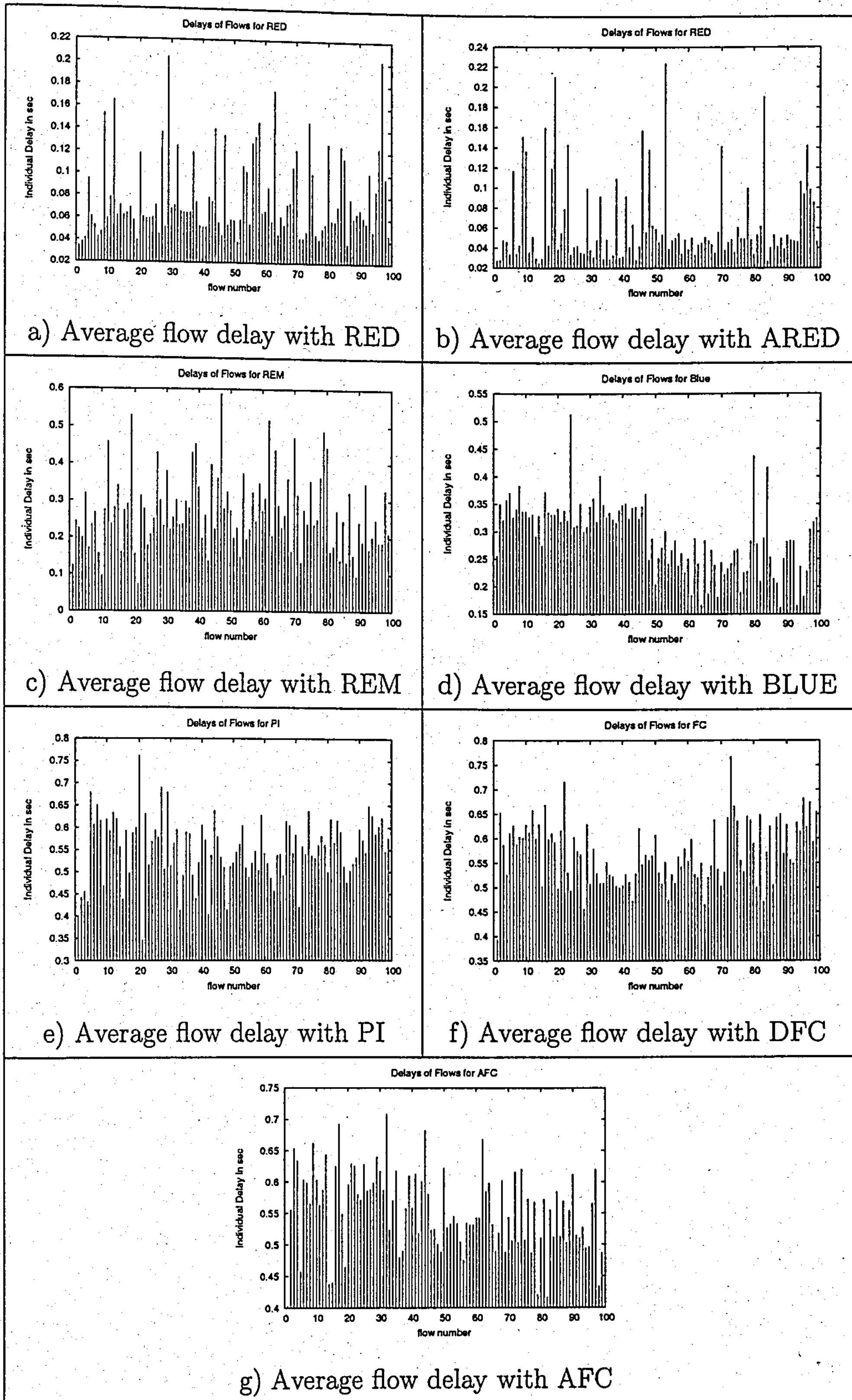


Figure 5.20. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

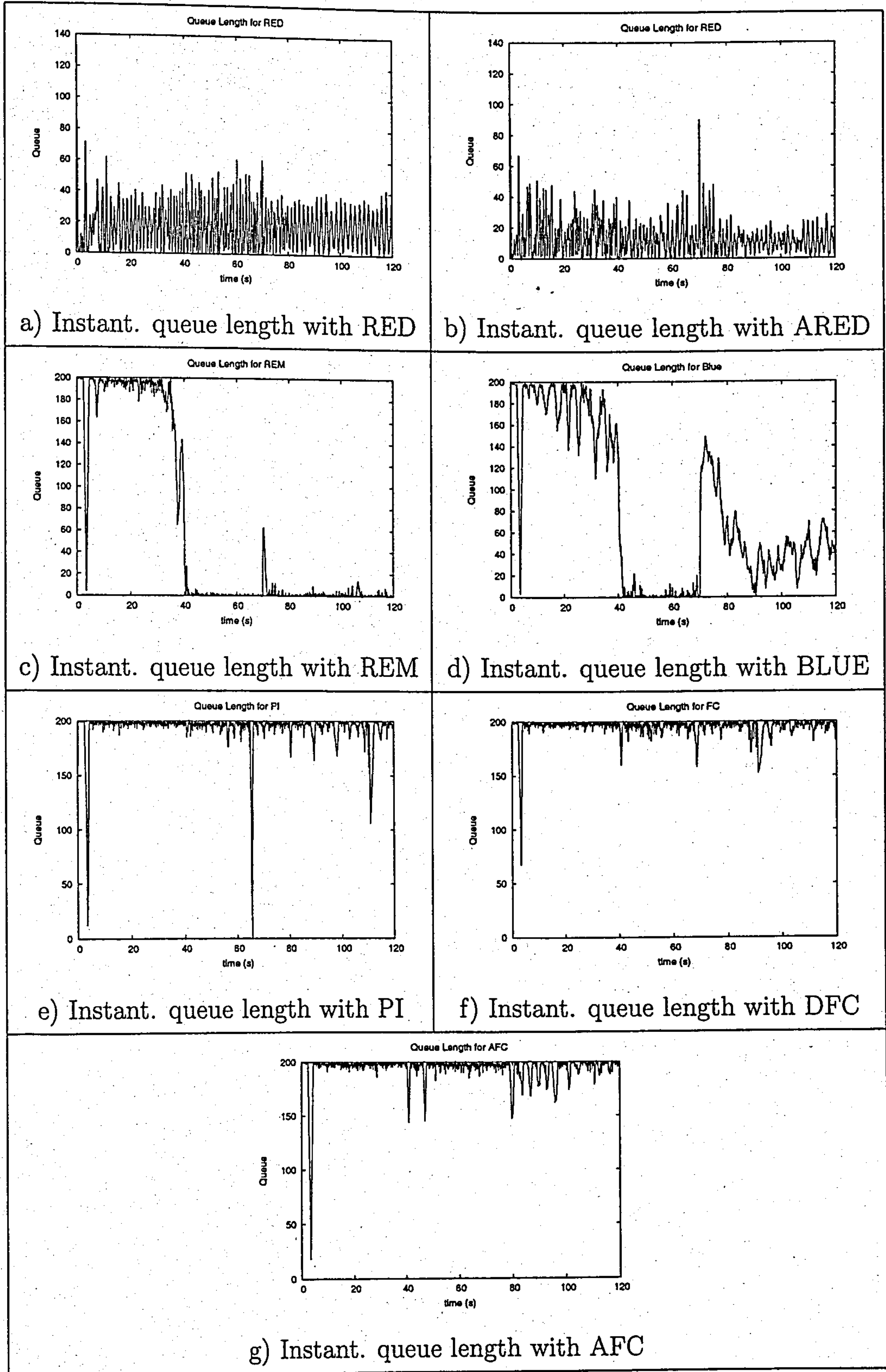
g) Average flow delay with AFC

Figure 5.21. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set

For the added WEB traffic, if ECN bit is not set, we observe that RED, REM and ARED perform similar to the case without the WEB traffic in terms of instantaneous queue length, shown in Figure 5.22. The drops of them are also comparable. BLUE seems to be only slightly affected. However, PI shows a worse tracking of the reference value, which is also clearly observable in Figure 5.22. Its bottleneck link utilization is also worse, which is demonstrated in Figure 5.23. The queue length tracking is best with the DFC and AFC and AFC drops the least packets, which can also be observed in Figure 5.24. Jain's fairness index is higher with BLUE, DFC and PI being the second and the third in respective order, however, from the traffic point of view, DFC and AFC perform better than the others, shown in Figure 5.25.

When ECN bit is set, BLUE and DFC benefit more than the others again. Figure 5.26 shows that AFC is again better than the others in tracking the reference value, since it performs better during slow start, however DFC and PI also perform similar to AFC. RED, REM and ARED perform also similar to the previous case of without WEB traffic.

As far as the fairness is concerned, Jain's index suggest that BLUE is the most fair, AFC and PI being the second among the AQM mechanisms studied. However, this is not the case from the delay variation point of view, shown in Figure 5.29. The delay variation is similar for DFC, PI and AFC, but BLUE shows worse results.



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.22. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic

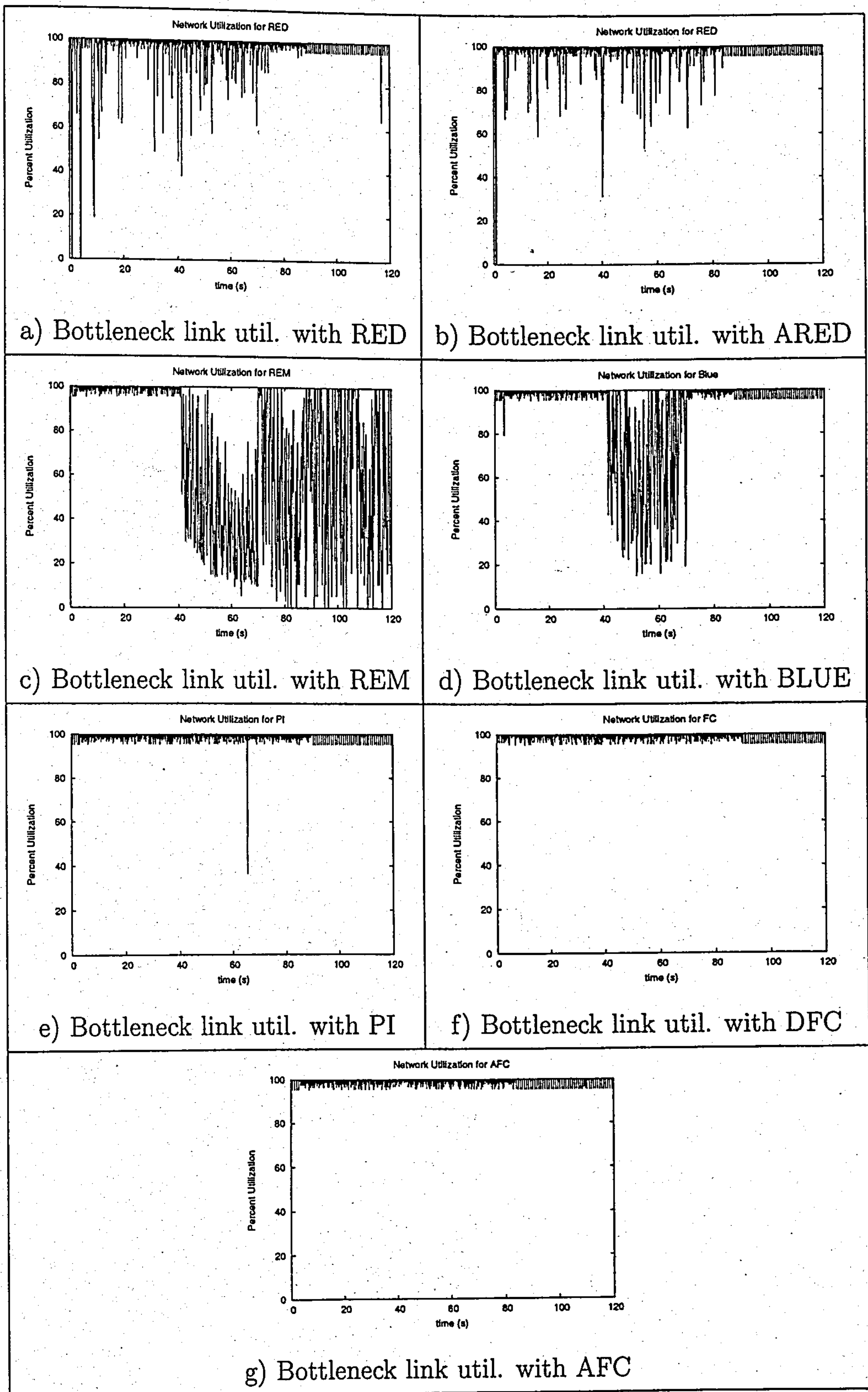
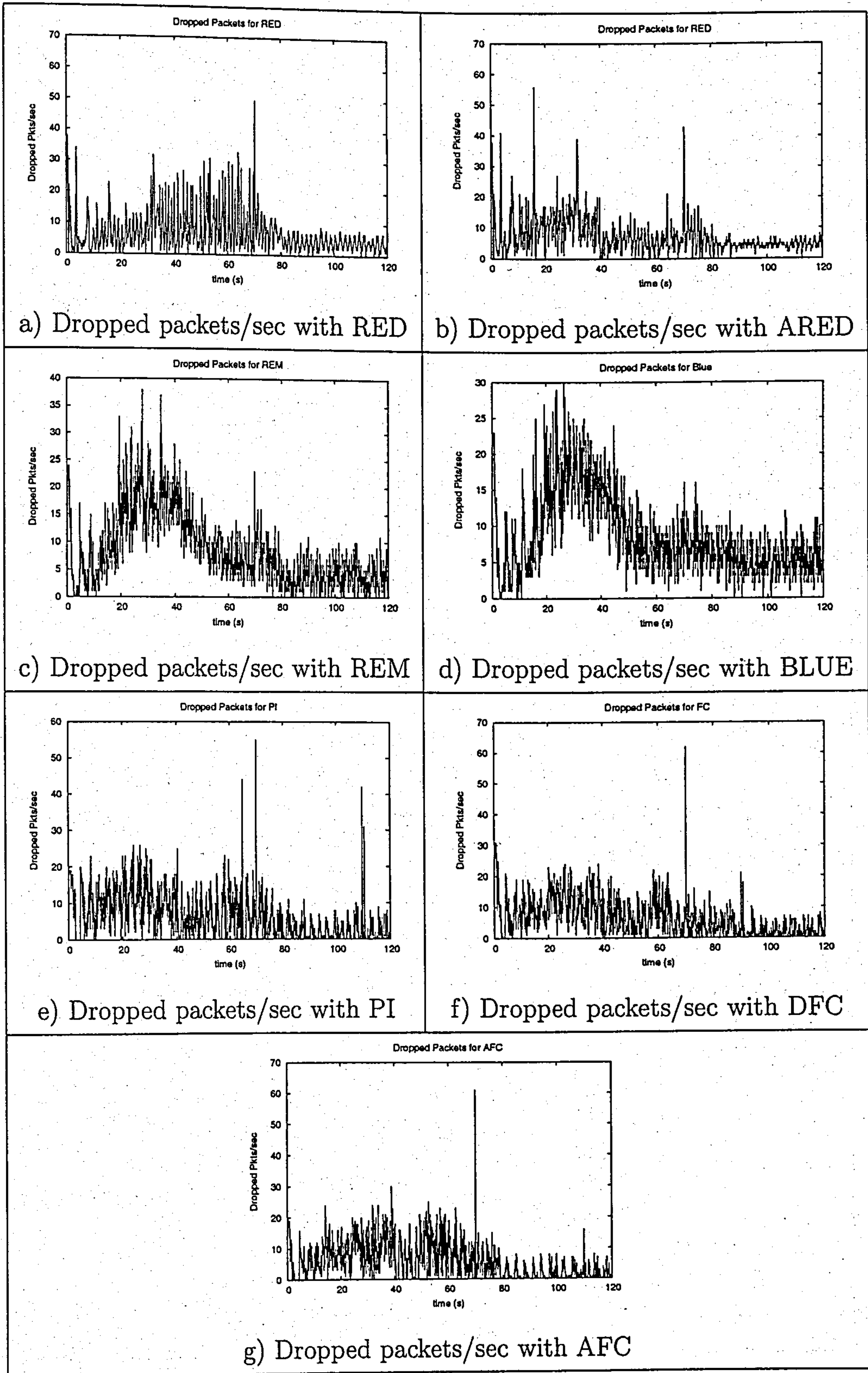
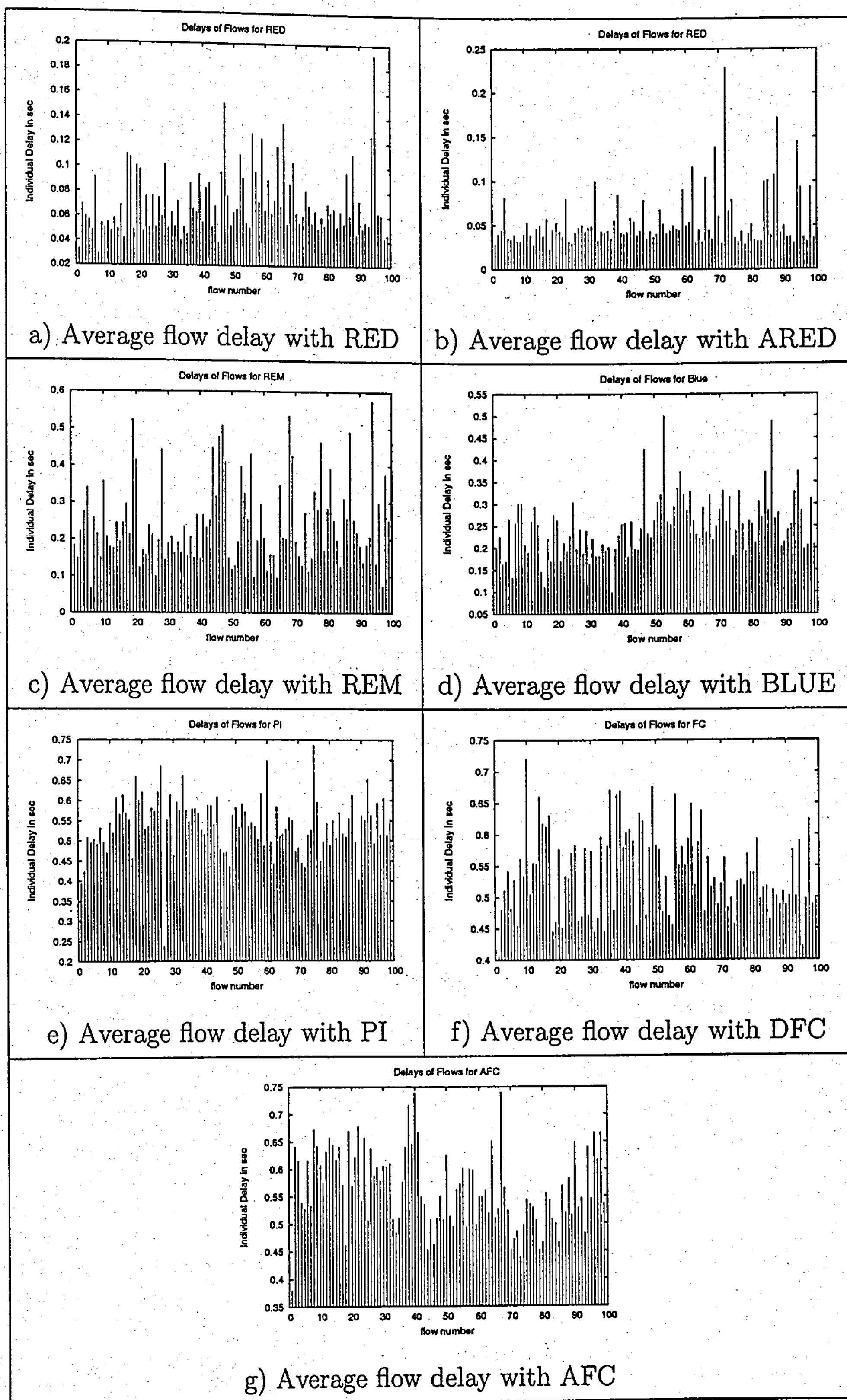


Figure 5.23. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Dropped packets/sec with RED
b) Dropped packets/sec with ARED
c) Dropped packets/sec with REM
d) Dropped packets/sec with BLUE
e) Dropped packets/sec with PI
f) Dropped packets/sec with DFC
g) Dropped packets/sec with AFC

Figure 5.24. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

g) Average flow delay with AFC

Figure 5.25. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic

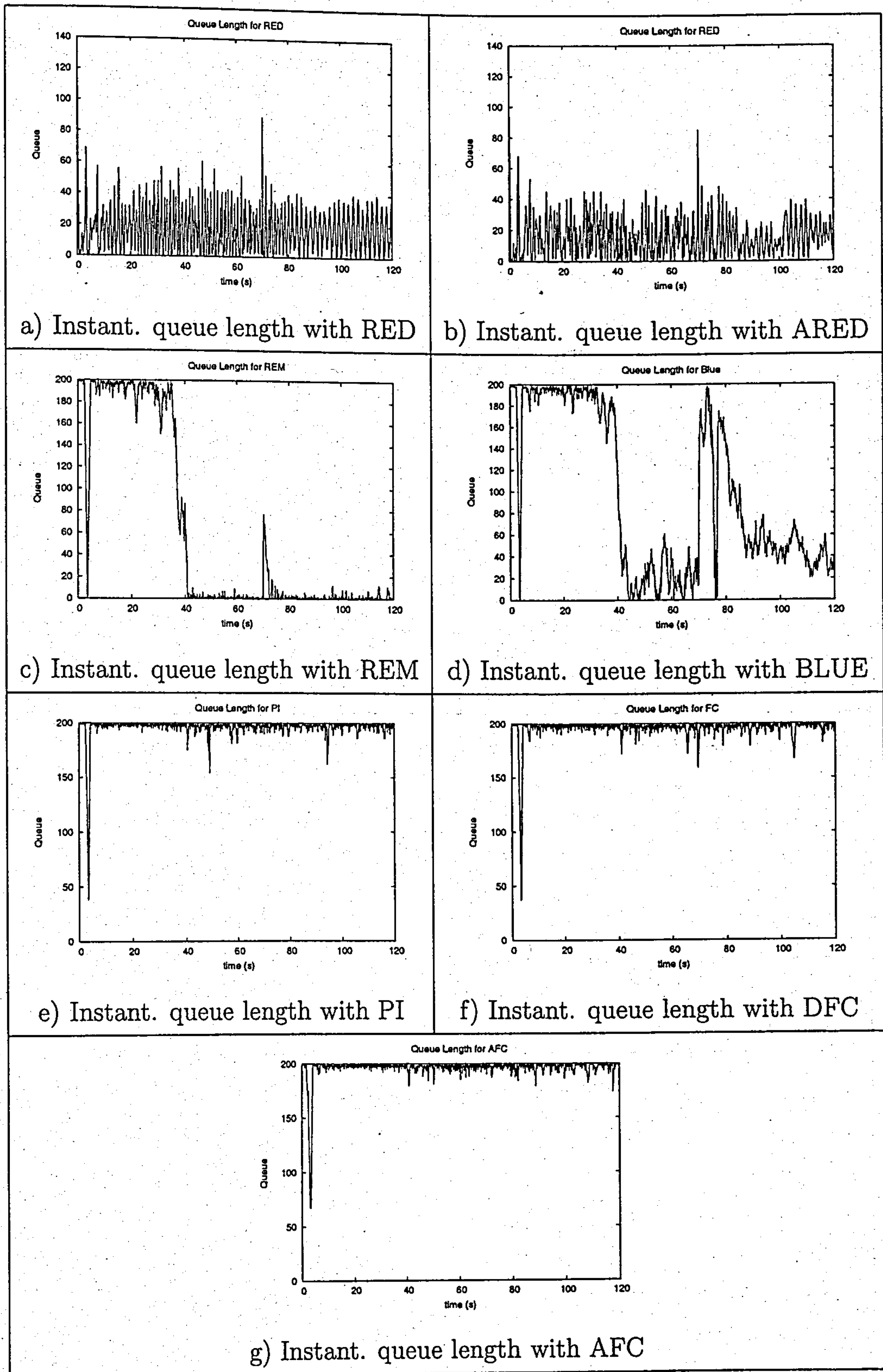
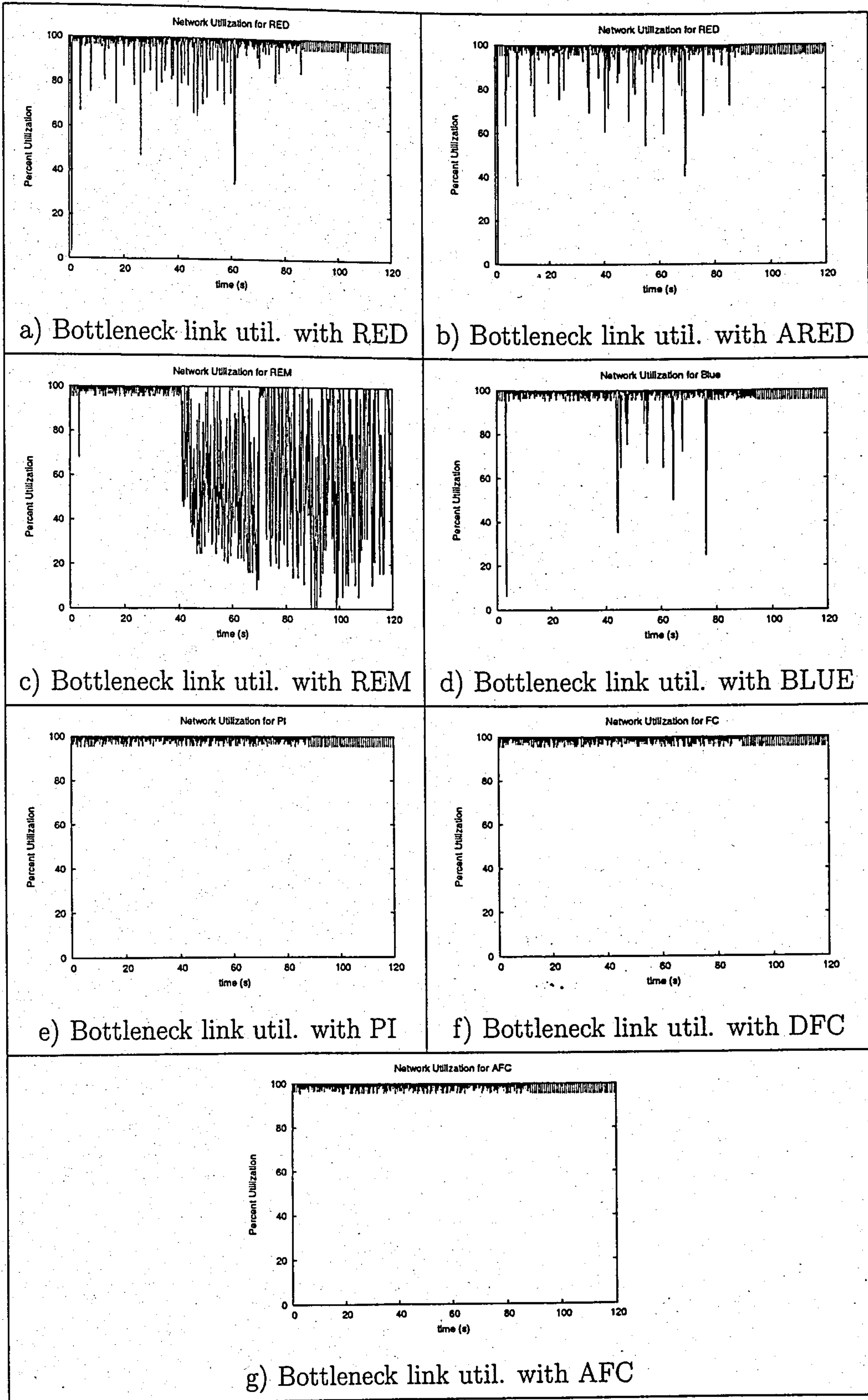


Figure 5.26. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

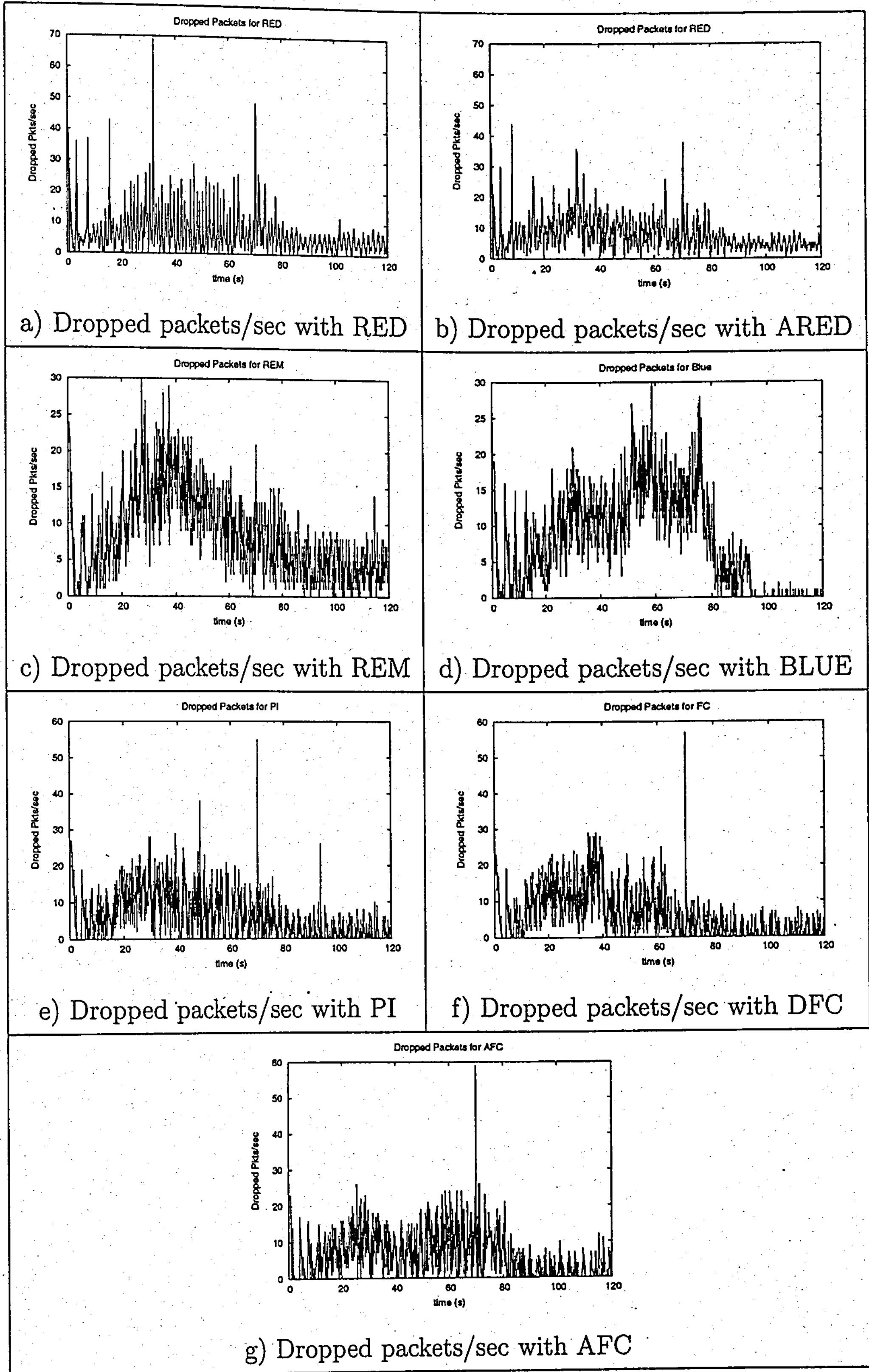
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.27. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

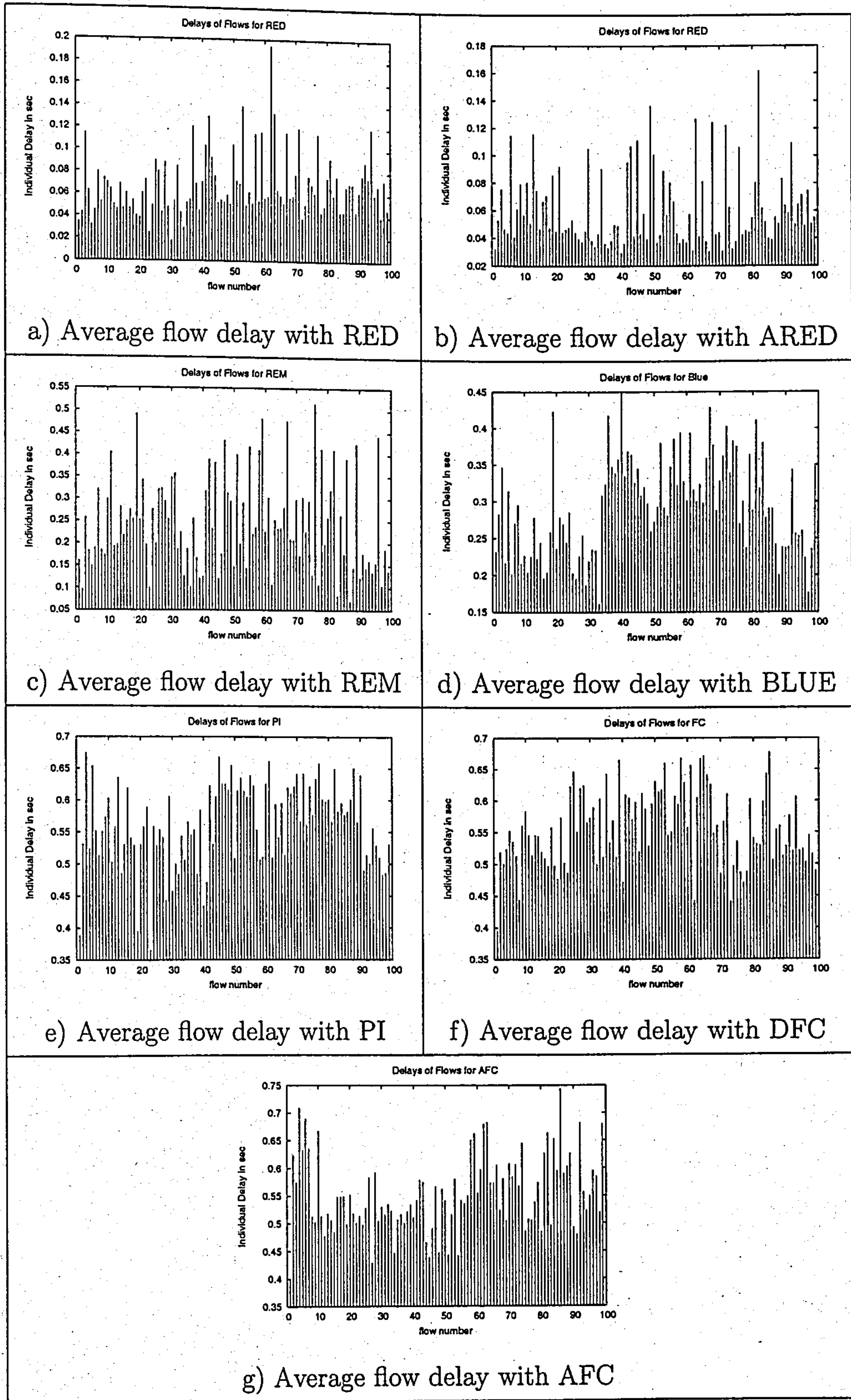
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.28. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

g) Average flow delay with AFC

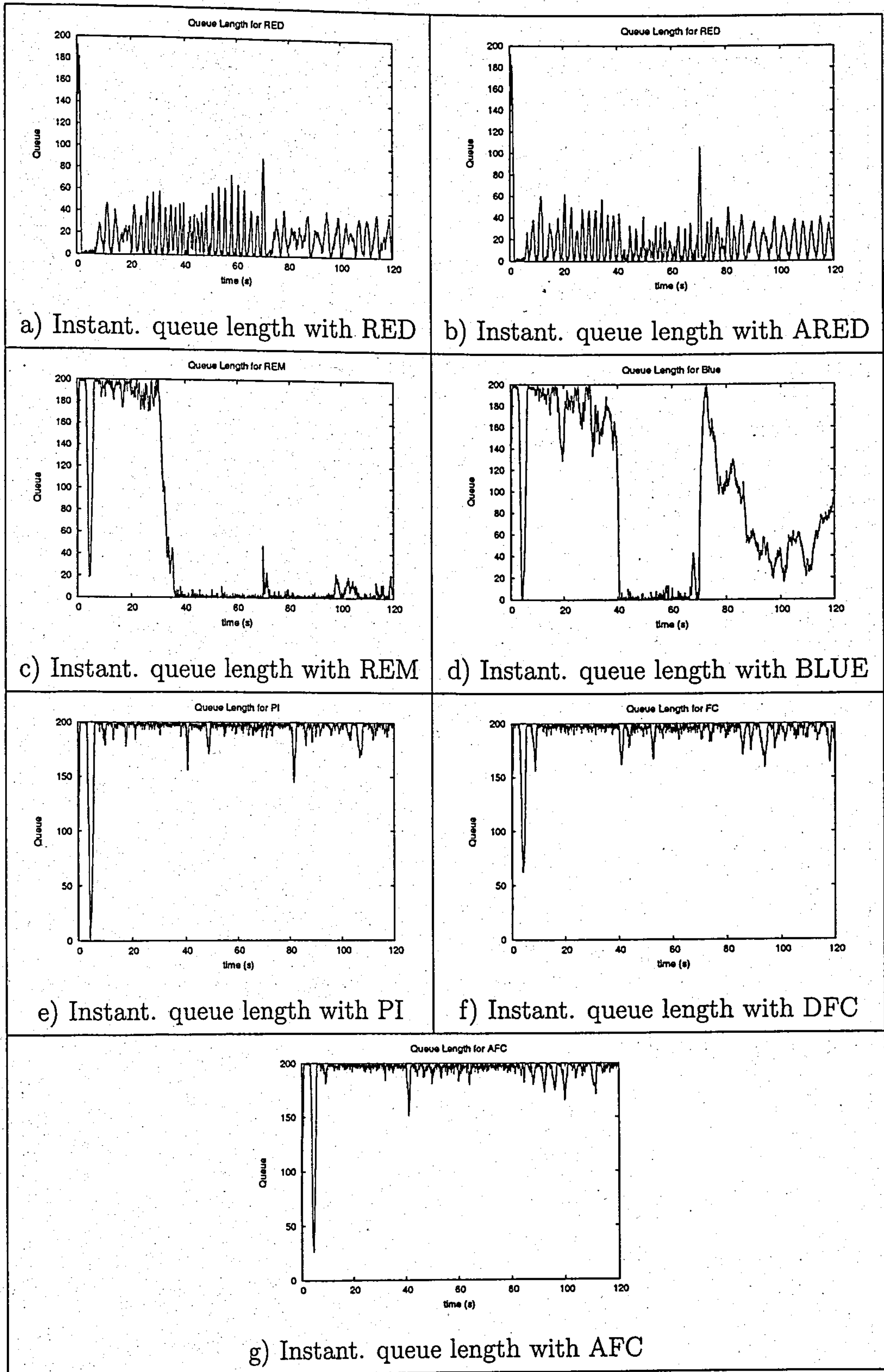
Figure 5.29. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic, while ECN bit is set

5.2.3. Scenarios with Changed Bottleneck Delay of 125 ms

The scenarios with $buffer = 200$ and bottleneck delay of 125 ms are divided into two, one having dynamic FTP traffic and CBR traffic, examined with either ECN bit is set or not, and the other one having WEB traffic added onto the first one and the ECN bit is not set. The delay 125 ms is examined to simulate MEO satellite links in order to compare the AQM mechanisms examined in such an environment.

If we look into the results, DFC and AFC performs less fluctuations in the queue length and they keep the reference value. PI also performs similar with slightly more fluctuations, which is shown in Figure 5.30. Since there is less traffic in the same simulation time because of the higher delay, there are less drops for all AQM mechanisms. RED, REM and ARED perform similar to the previous cases, however bottleneck link is more utilized for them compared to the "5 ms" case, shown in Figure 5.31. Jain's fairness index is this time the highest with PI and the second one is AFC, however PI has a value of 0.87 and AFC has a value of 0.86 for the fairness index, which is not a great difference. The third one is BLUE with 0.81. From the delay variation shown in Figure 5.33, PI, AFC and DFC seem to behave more fair towards the connections. Another important point is that AFC has the least drops among all mechanisms. The second one is DFC, third one is the PI controller with a difference of nearly 500 packets.

If ECN bit is set, RED, REM and ARED perform more drops because of the early buffer overflow. Drops are again the least with AFC, DFC being the second and PI being the third with a difference of nearly 800 packets. The queue length fluctuation shown in Figure 5.34 is similar to the previous cases, however there is a slightly drop in bottleneck link utilization for a short moment with DFC, which can be observed in Figure 5.35 Also note that, the queue length drops to zero in the dynamic region with BLUE and it stays near zero with REM for most of the time.



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

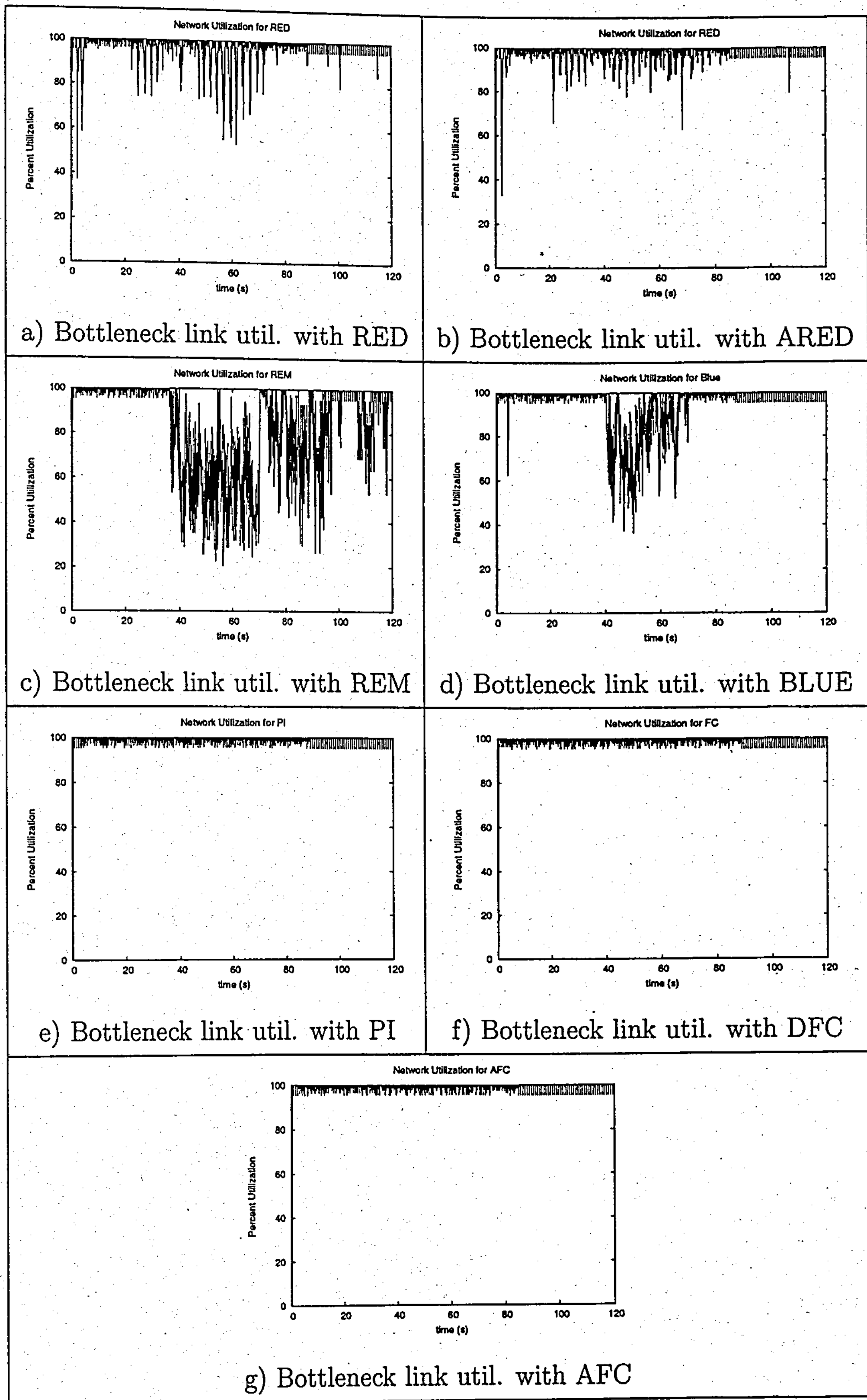
d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.30. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.31. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic

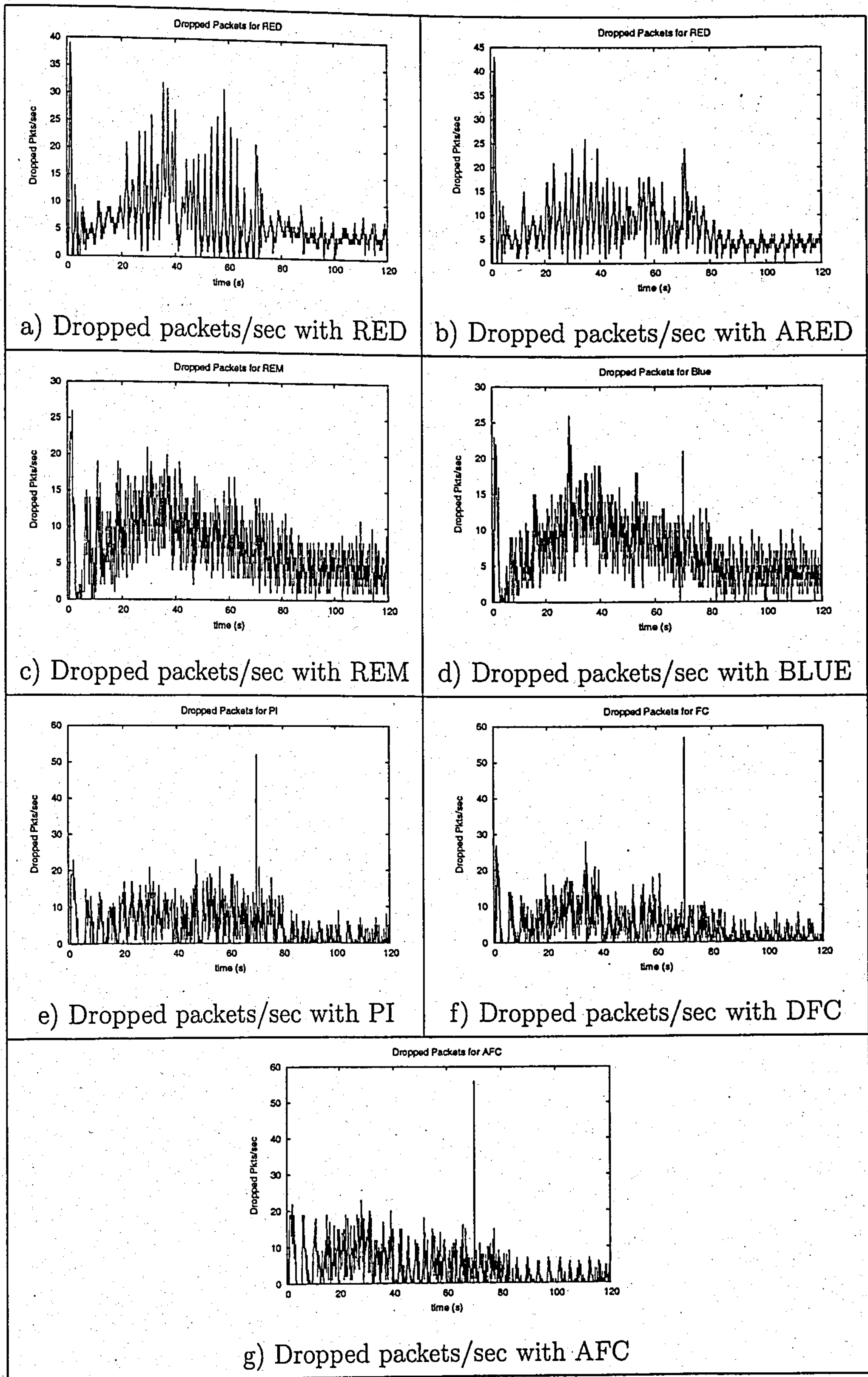


Figure 5.32. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic

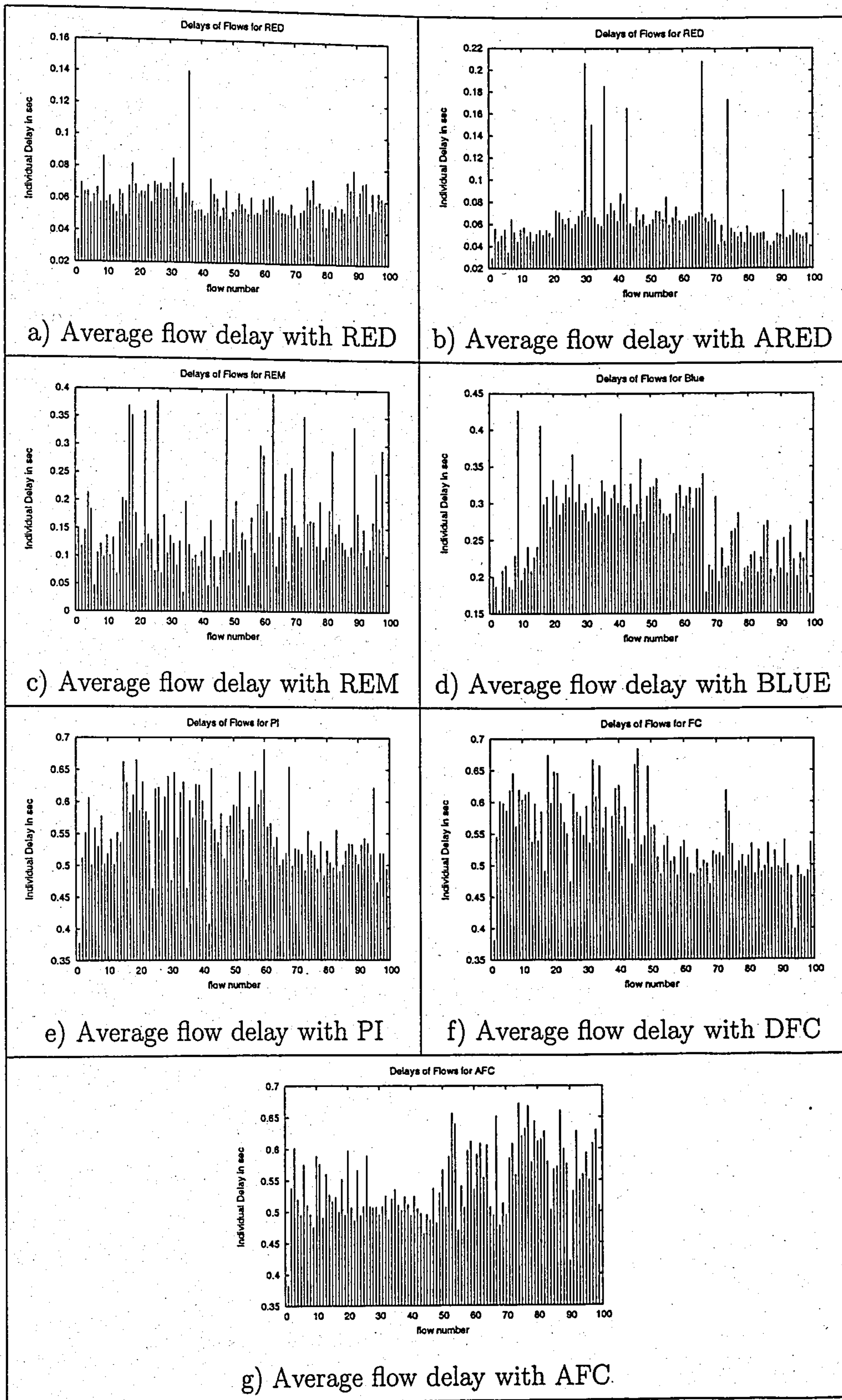
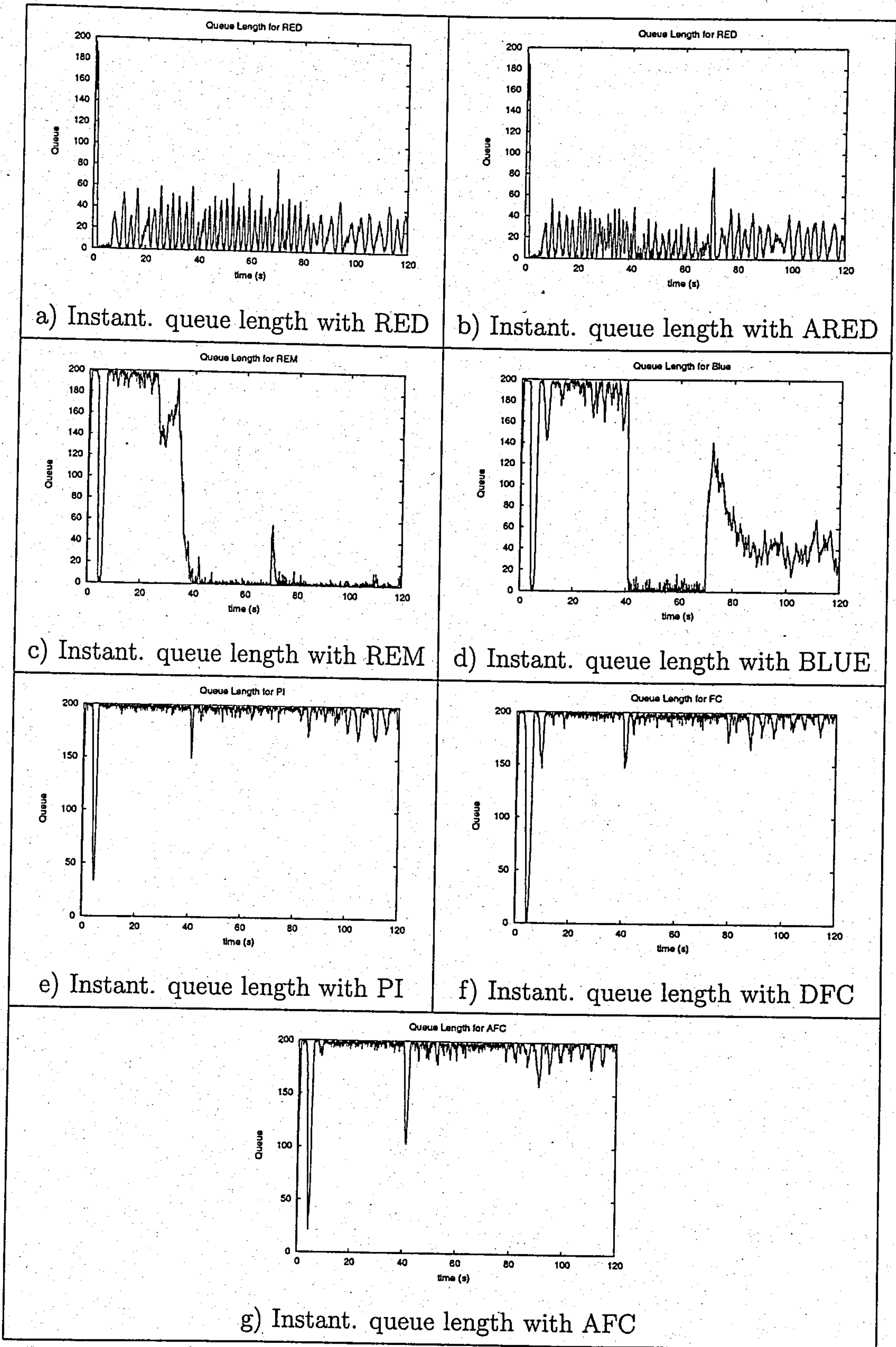


Figure 5.33. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

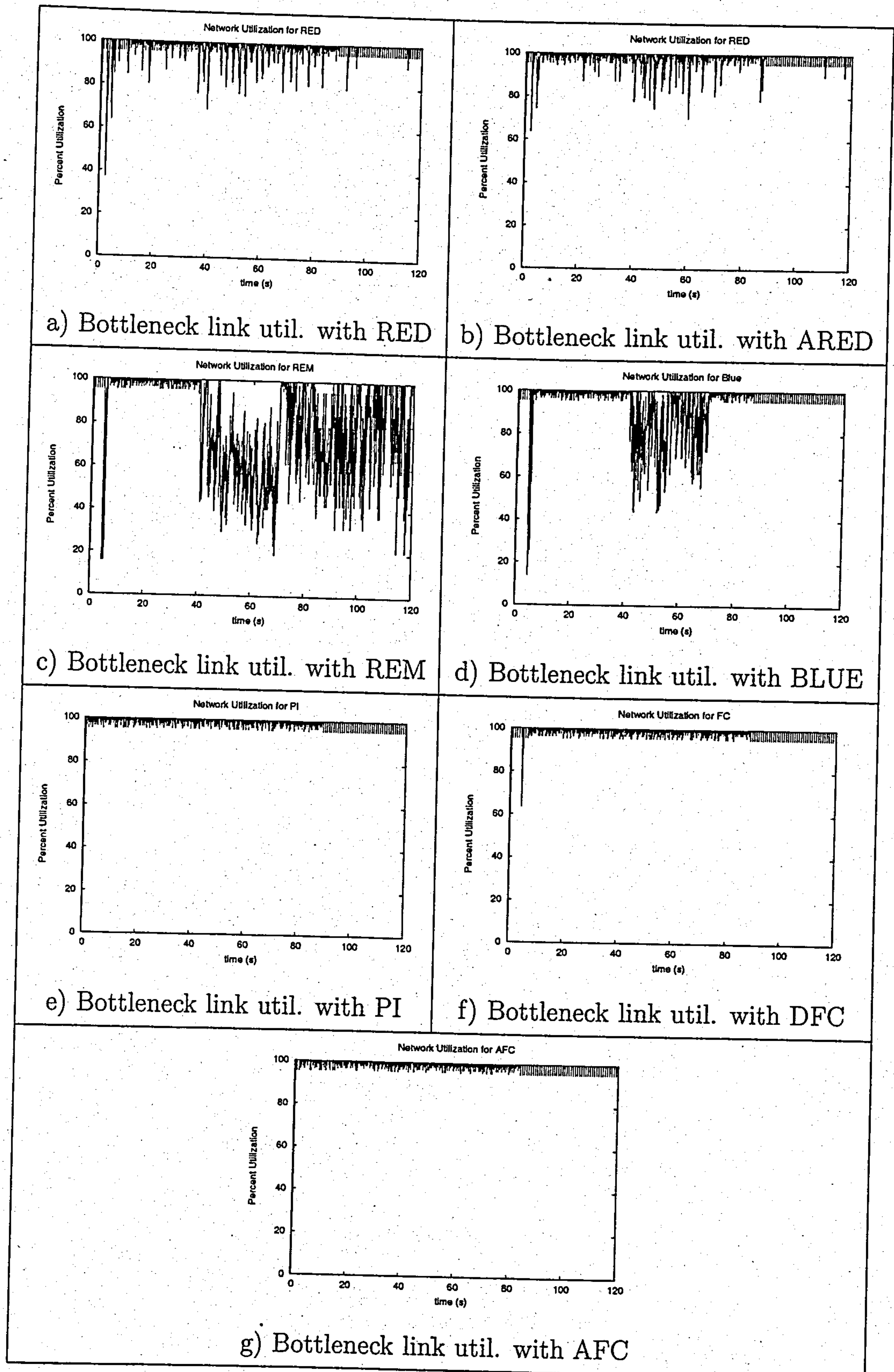
d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

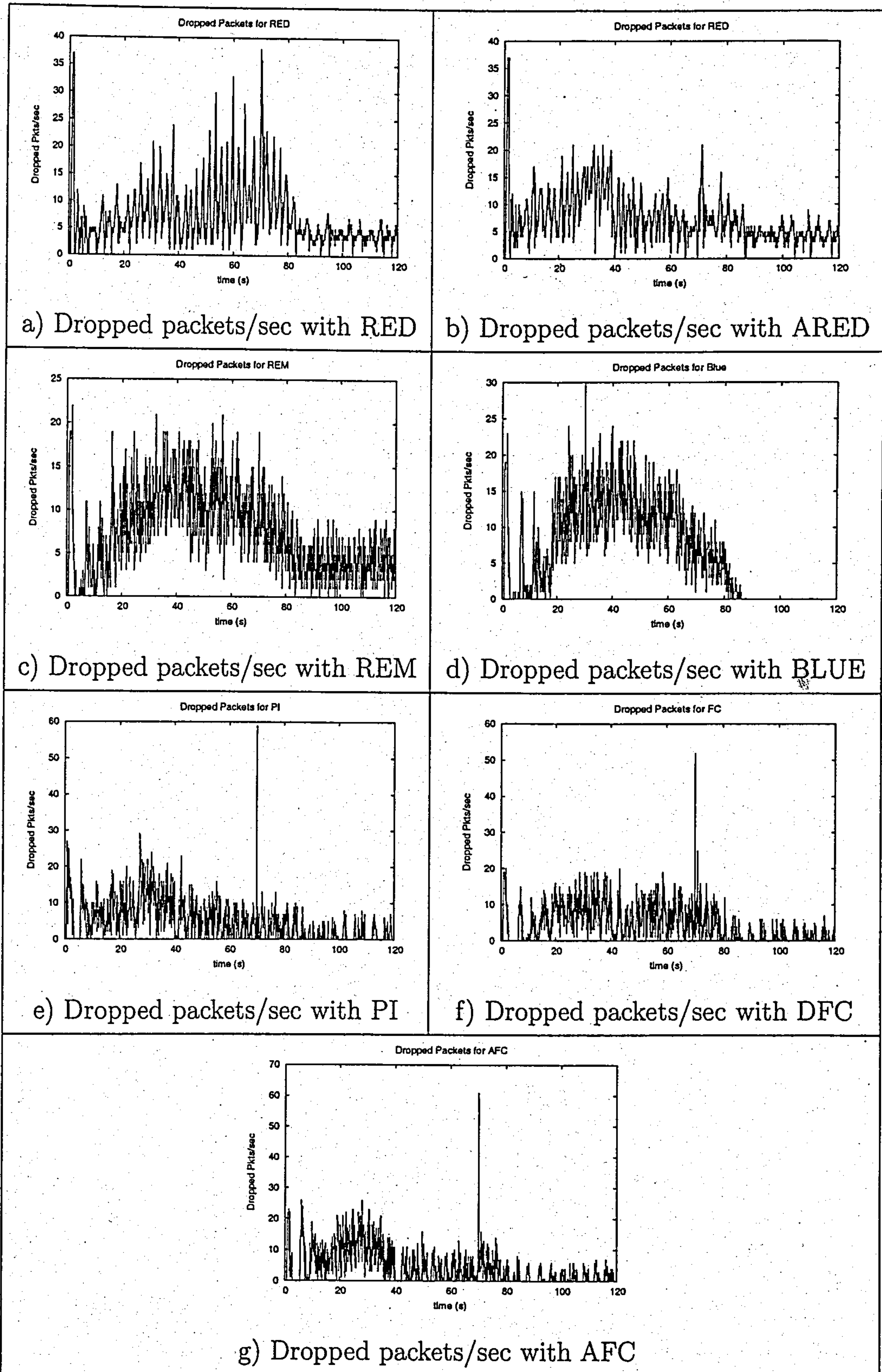
g) Instant. queue length with AFC

Figure 5.34. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set



a) Bottleneck link util. with RED
b) Bottleneck link util. with ARED
c) Bottleneck link util. with REM
d) Bottleneck link util. with BLUE
e) Bottleneck link util. with PI
f) Bottleneck link util. with DFC
g) Bottleneck link util. with AFC

Figure 5.35. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.36. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set

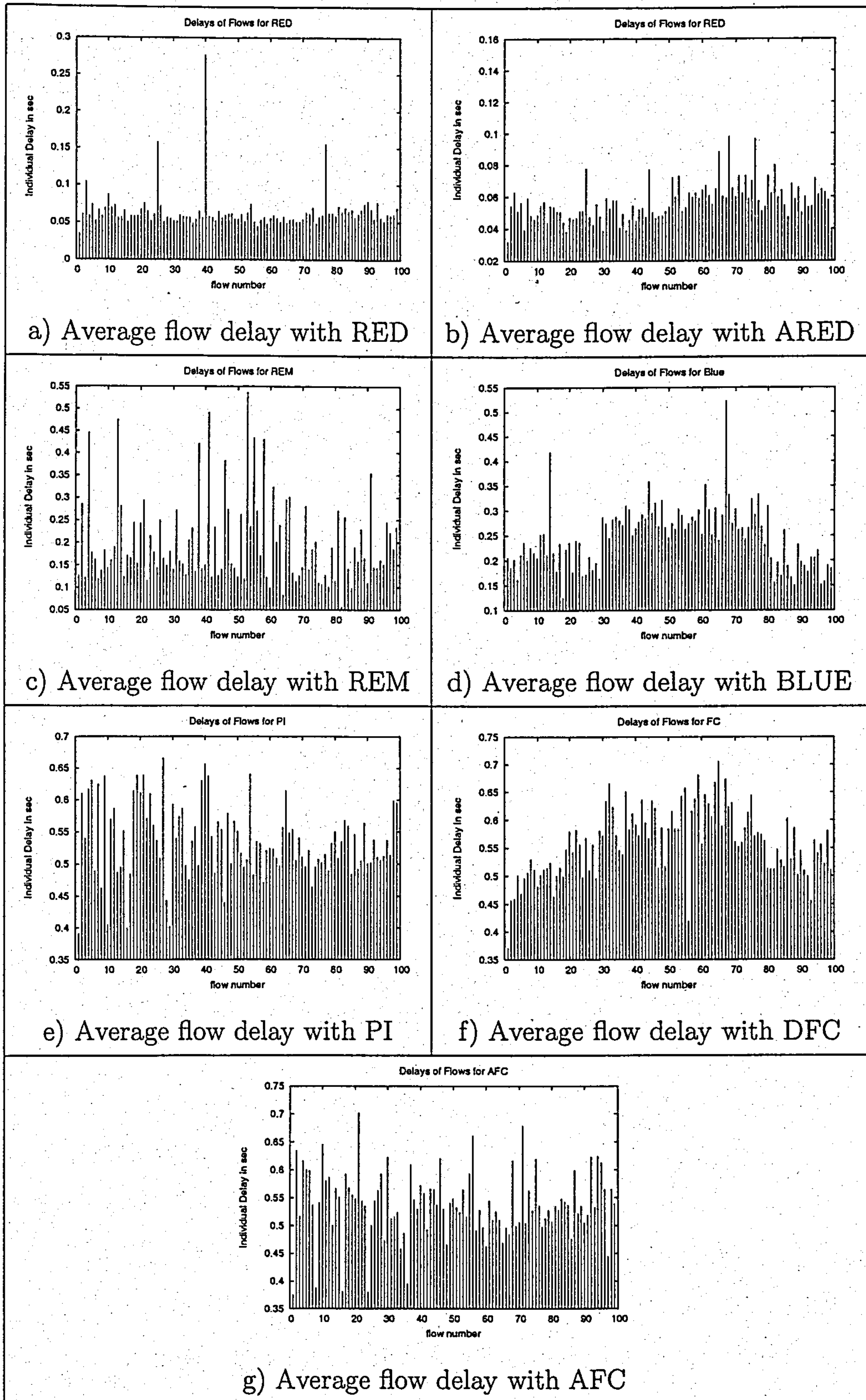


Figure 5.37. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic, while ECN bit is set

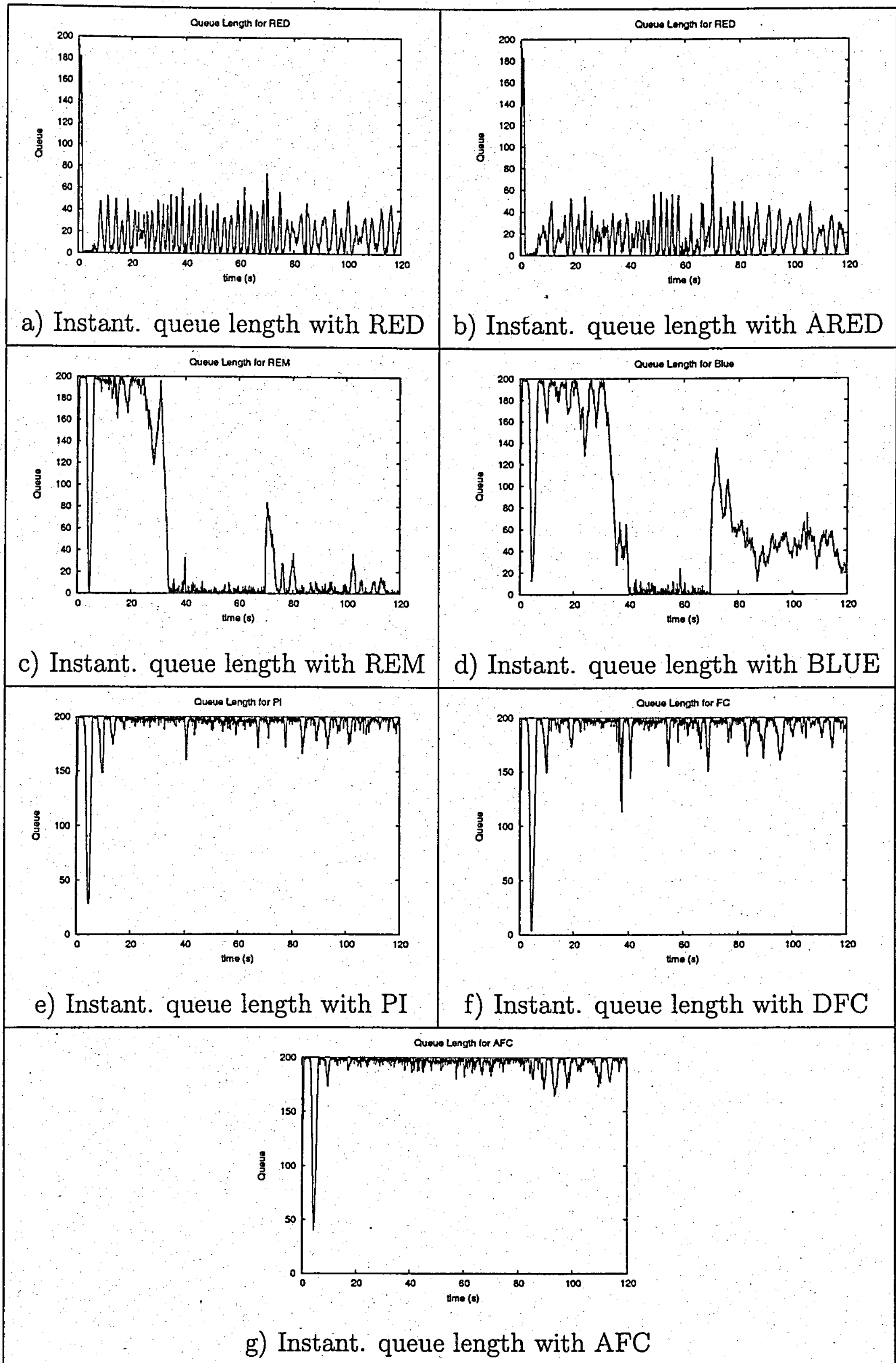
If WEB traffic added and ECN bit is again set to zero, we see in Figure 5.38 that AFC performs the least fluctuations and PI is performing similar to it. PI and AFC drop the lest number of packets with a difference of four. ARED has the highest Jain's index with a value of 0.80. In some situations, RED, REM and ARED have high fairness indexes. This means that those algorithms give nearly equal amount of resource to the connections sharing bottleneck link. However, since the bottleneck link is not fully utilized when those algorithms are used, the high value of the fairness index is not so important. In this particular scenario, the second highest fairness index belongs to RED and BLUE with a value of 0.77 and the third highest index is 0.75 with AFC.

5.2.4. Scenarios with Changed Number of Nodes

For those cases, dynamic FTP, CBR and WEB traffic remain, number of nodes are changed to 60 and simulations are performed for 5 ms and 125 ms bottleneck delays. Since the number of nodes are diminished, there is less traffic on the congested router.

If we look at the case, where the delay is 125 ms, it can be seen that AFC has the least queue fluctuation, which is shown in Figure 5.42. PI and DFC perform similar to each other, but they are worse than AFC. Network utilization, shown in Figure 5.43 is the highest with PI and AFC. The utilization is similar for RED, ARED and REM compared to the previous cases. AFC and BLUE have the highest fairness index with a value of 0.80, DFC and RED being the second with 0.79. From the delay variation shown in Figure 5.45, it is seen that AFC and PI have the least variation.

If the delay is changed to 5 ms, then we can observe from Figure 5.46 that AFC is the best keeping the track of the reference value for the queue length with the least fluctuations. Changing the delay to 5 ms brings more queue fluctuations for RED and ARED. The slow start drawback of the PI algorithm can be seen in Figure 5.47. Actually, only AFC manages to almost fully utilize the bottleneck link. DFC performs a bit worse than AFC. AFC has the highest fairness index of 0.80 and Figure 5.49 shows that it has less fluctuations among mechanisms and hence it is the most fair one.



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.38. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic

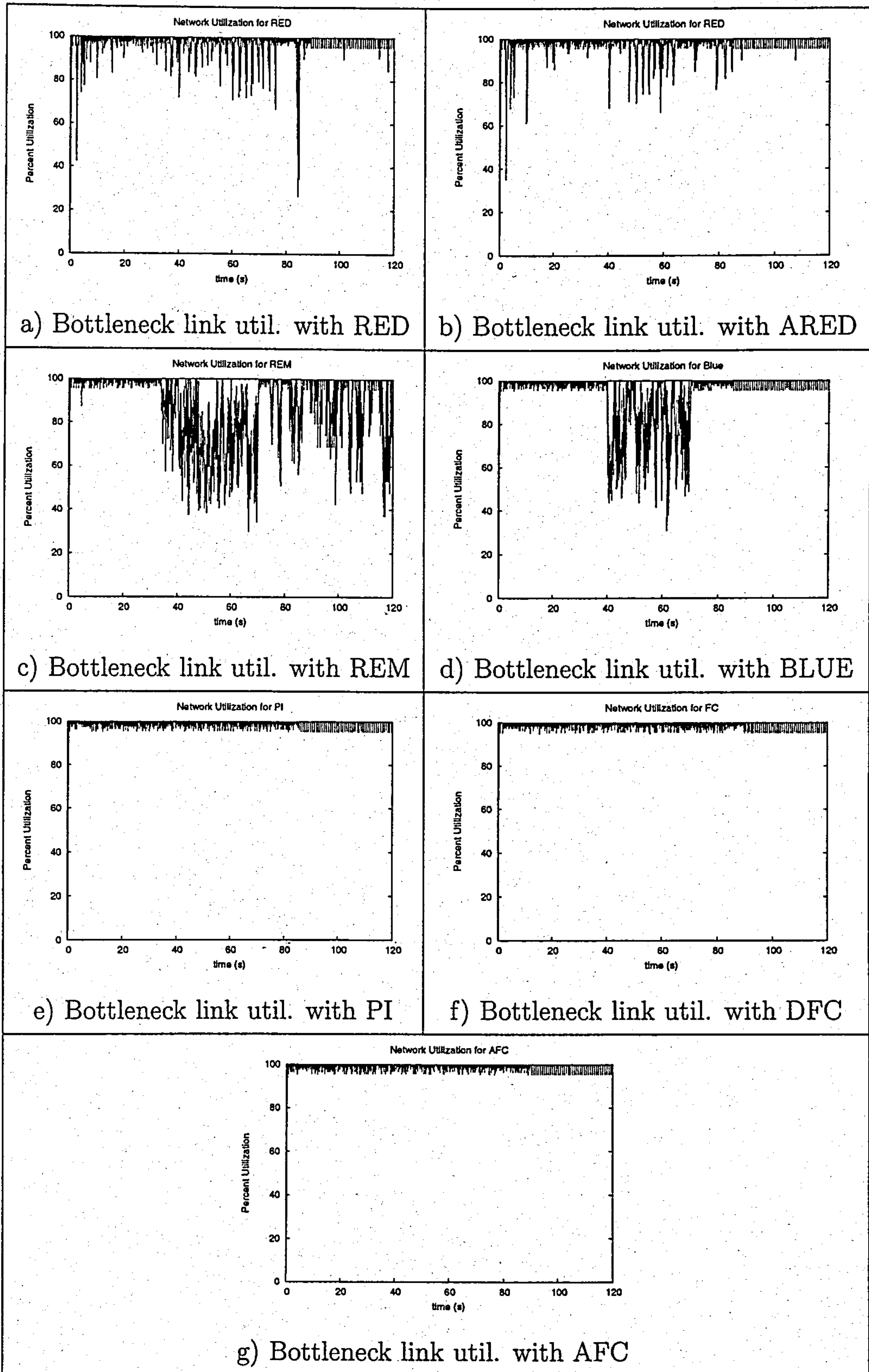
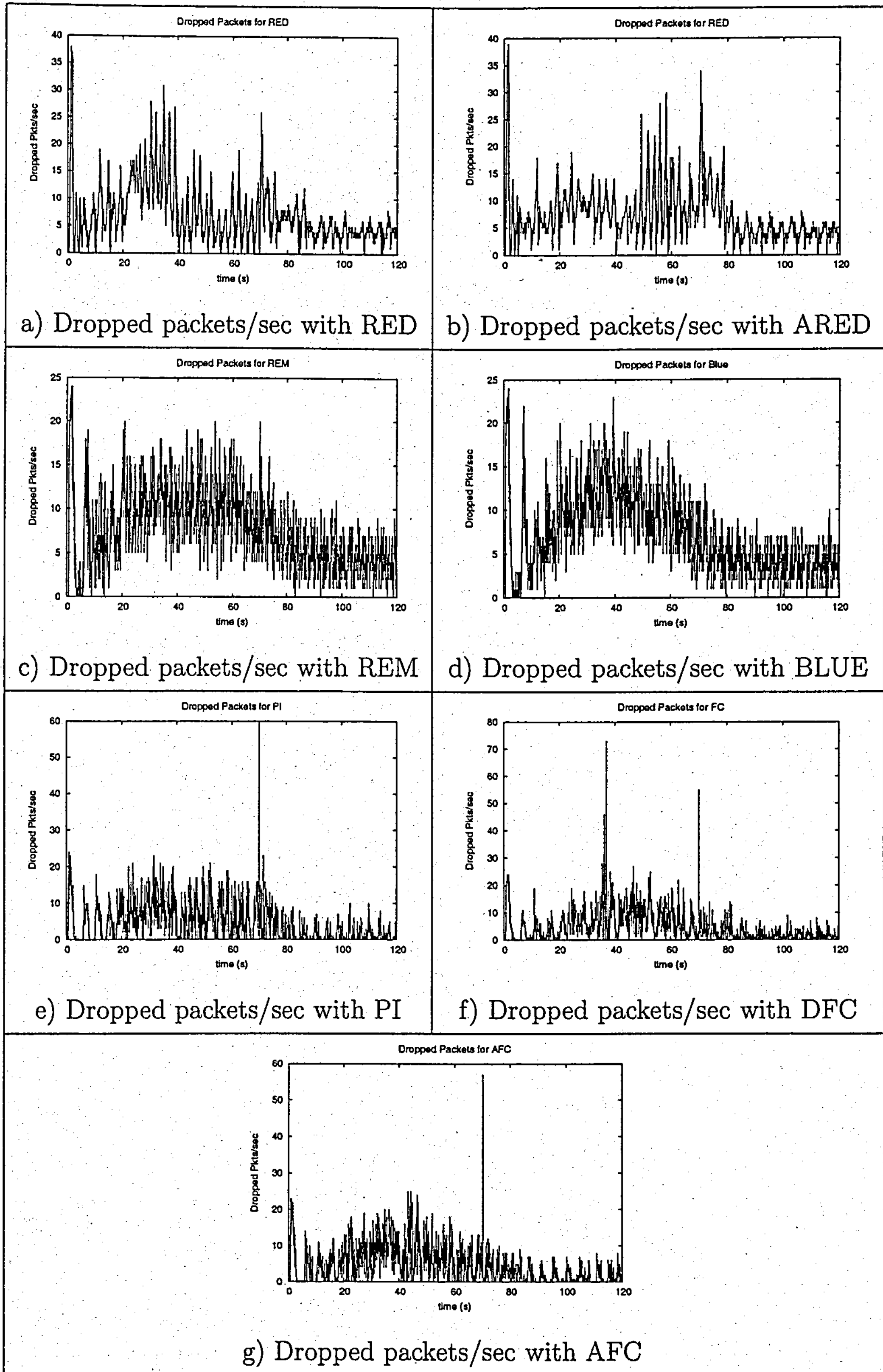


Figure 5.39. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

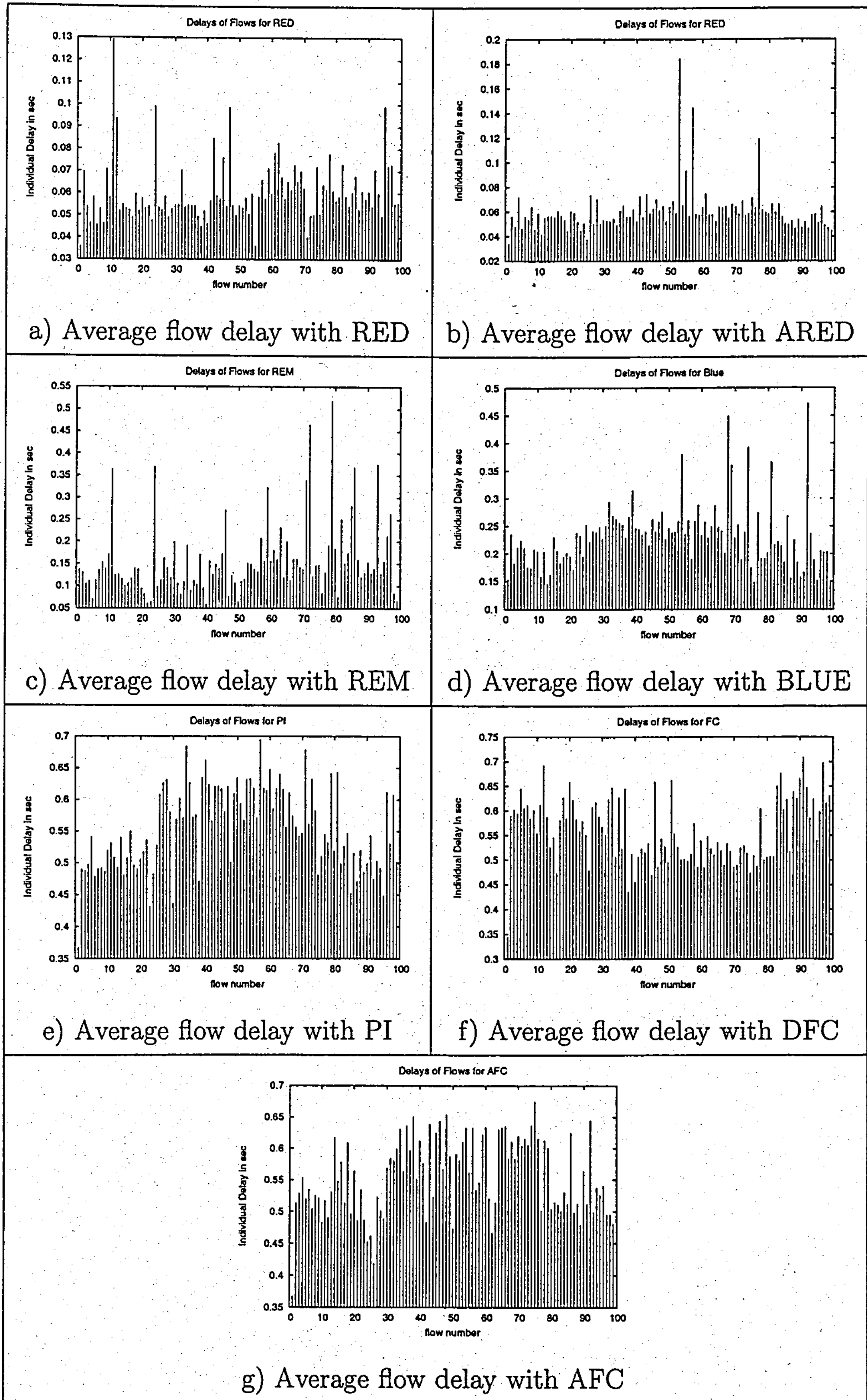
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.40. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

g) Average flow delay with AFC

Figure 5.41. Comparison of average individual flow delays for $buffer = 200$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic

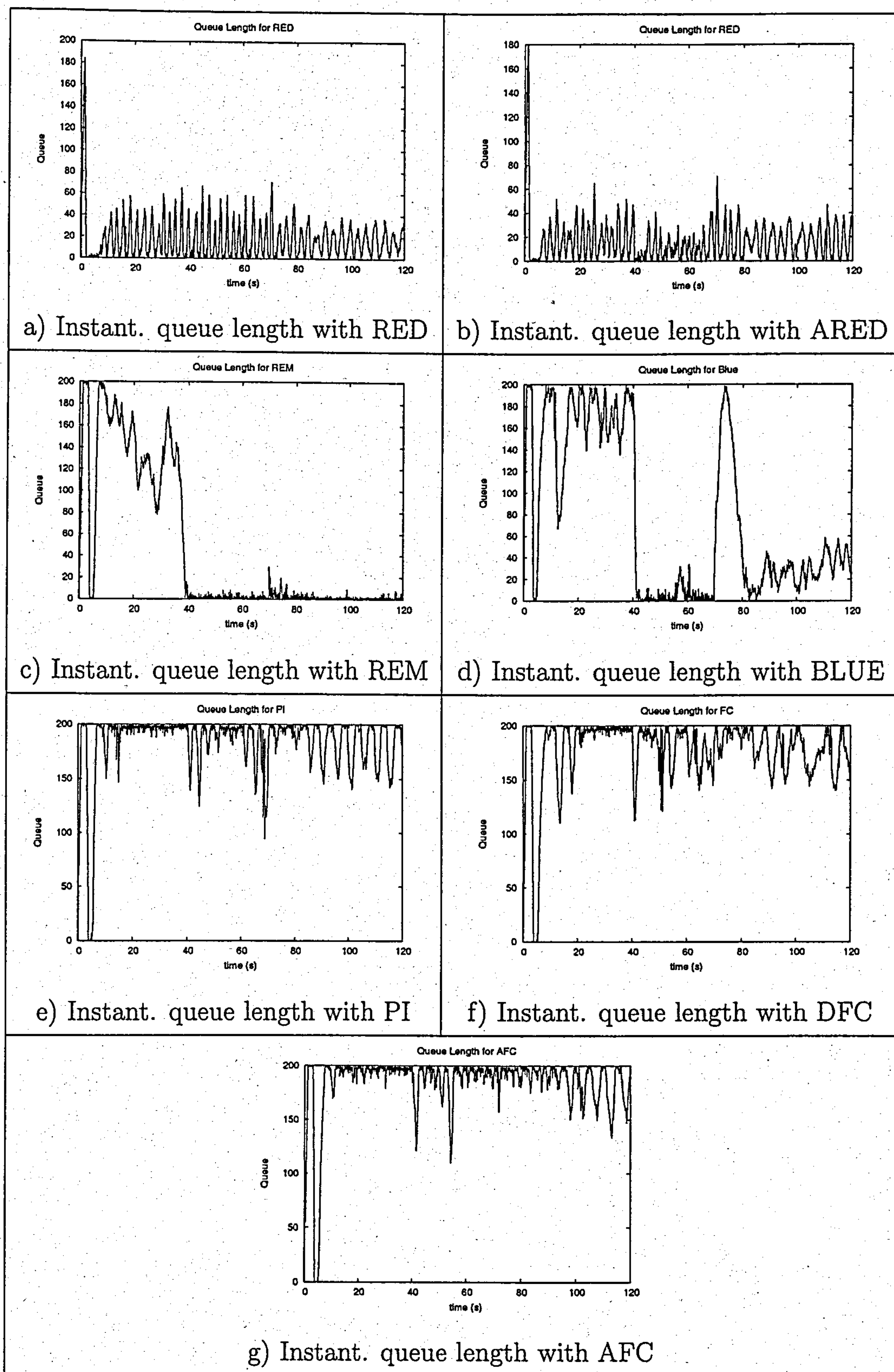
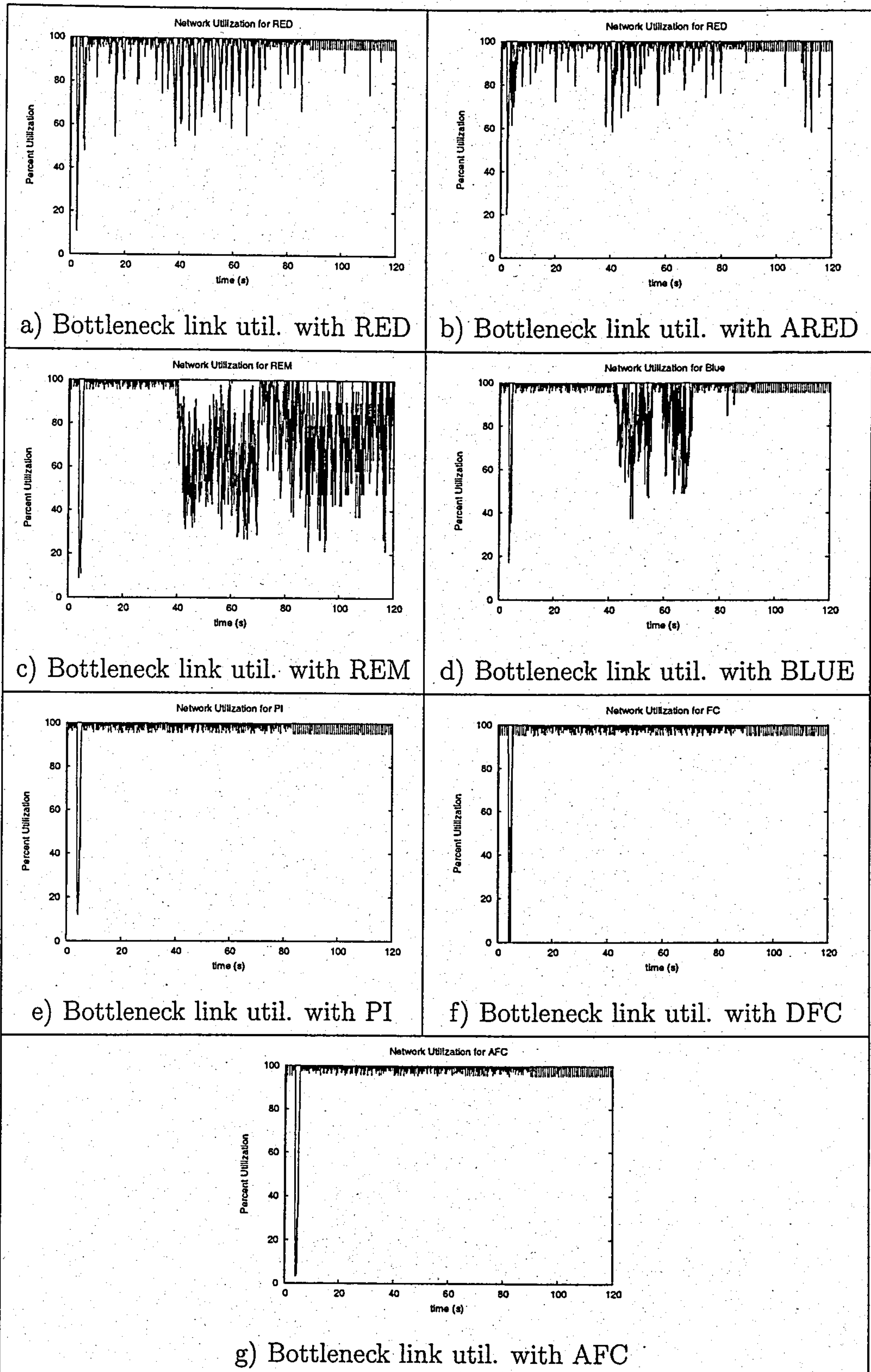


Figure 5.42. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

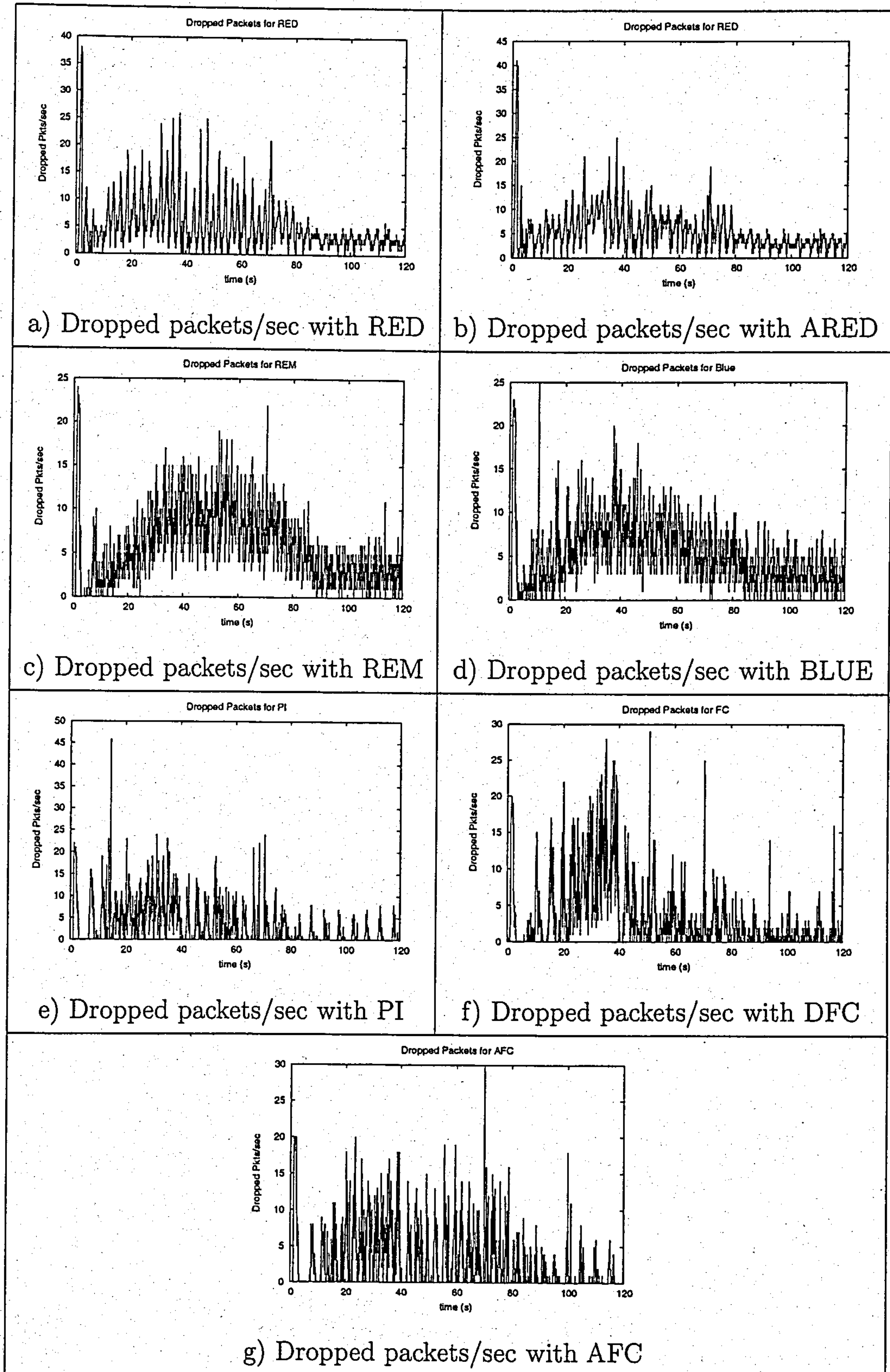
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.43. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.44. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic

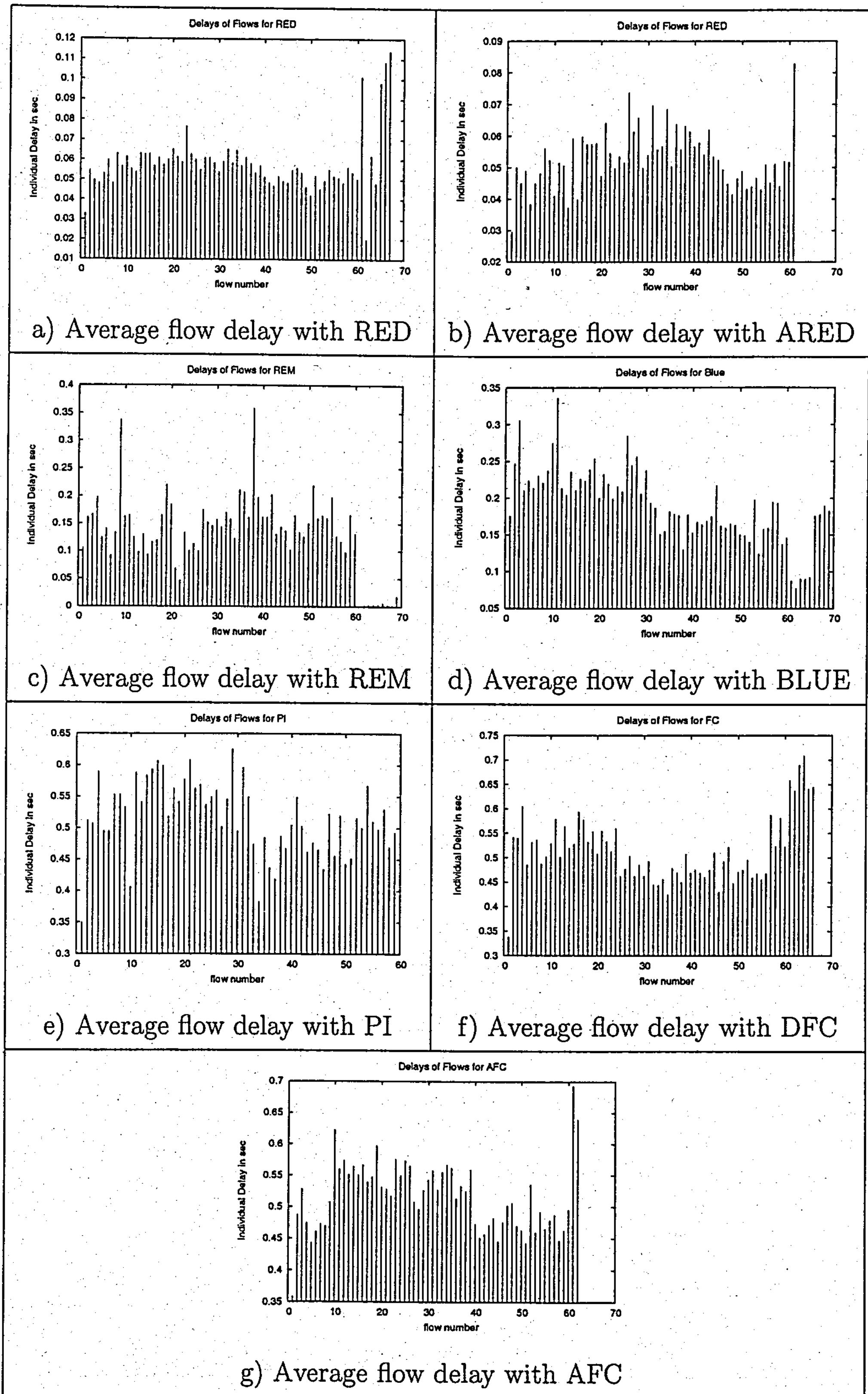
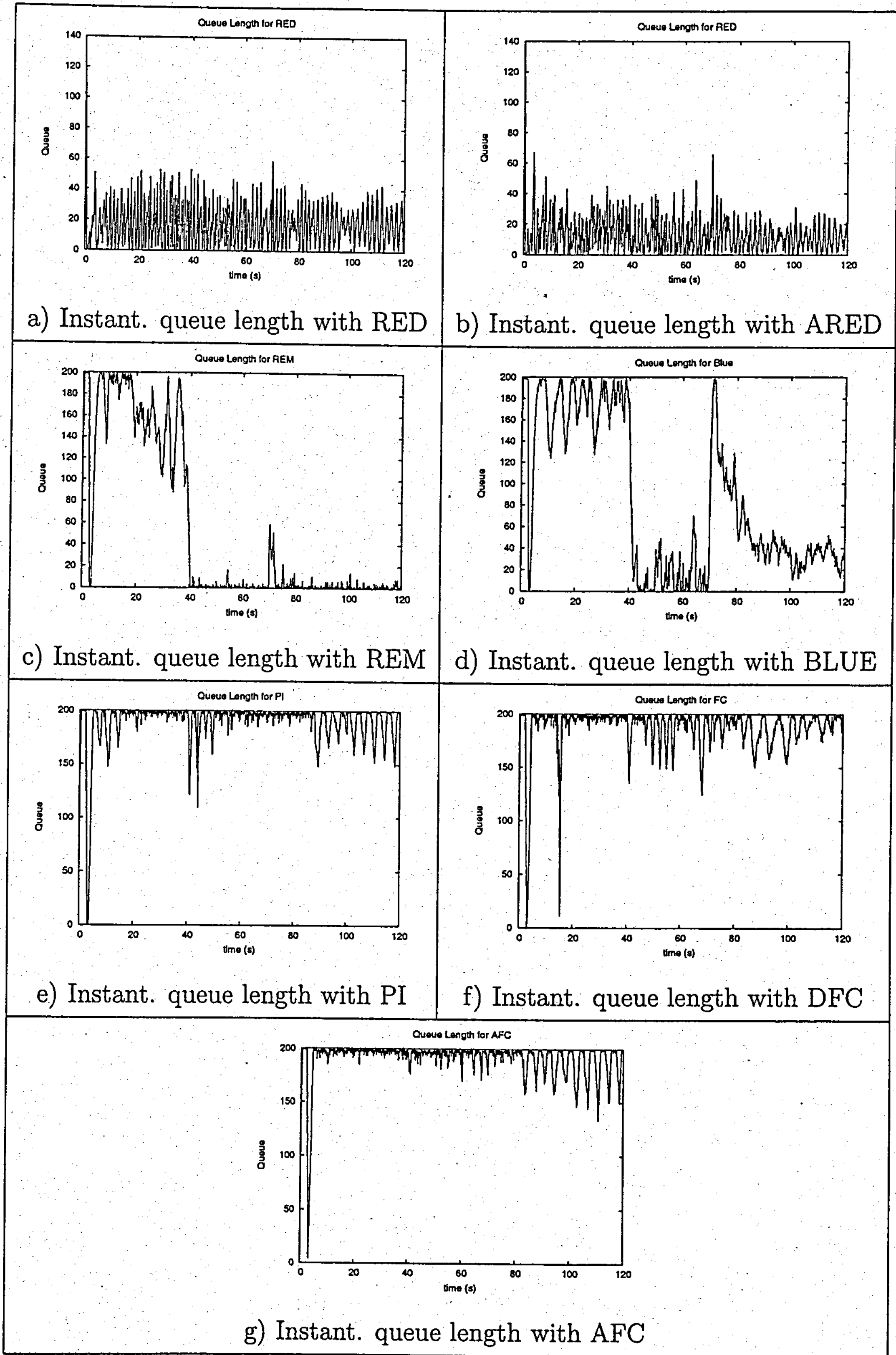


Figure 5.45. Comparison of average individual flow delays for $buffer = 200$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.46. Comparison of instantaneous queue lengths for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic

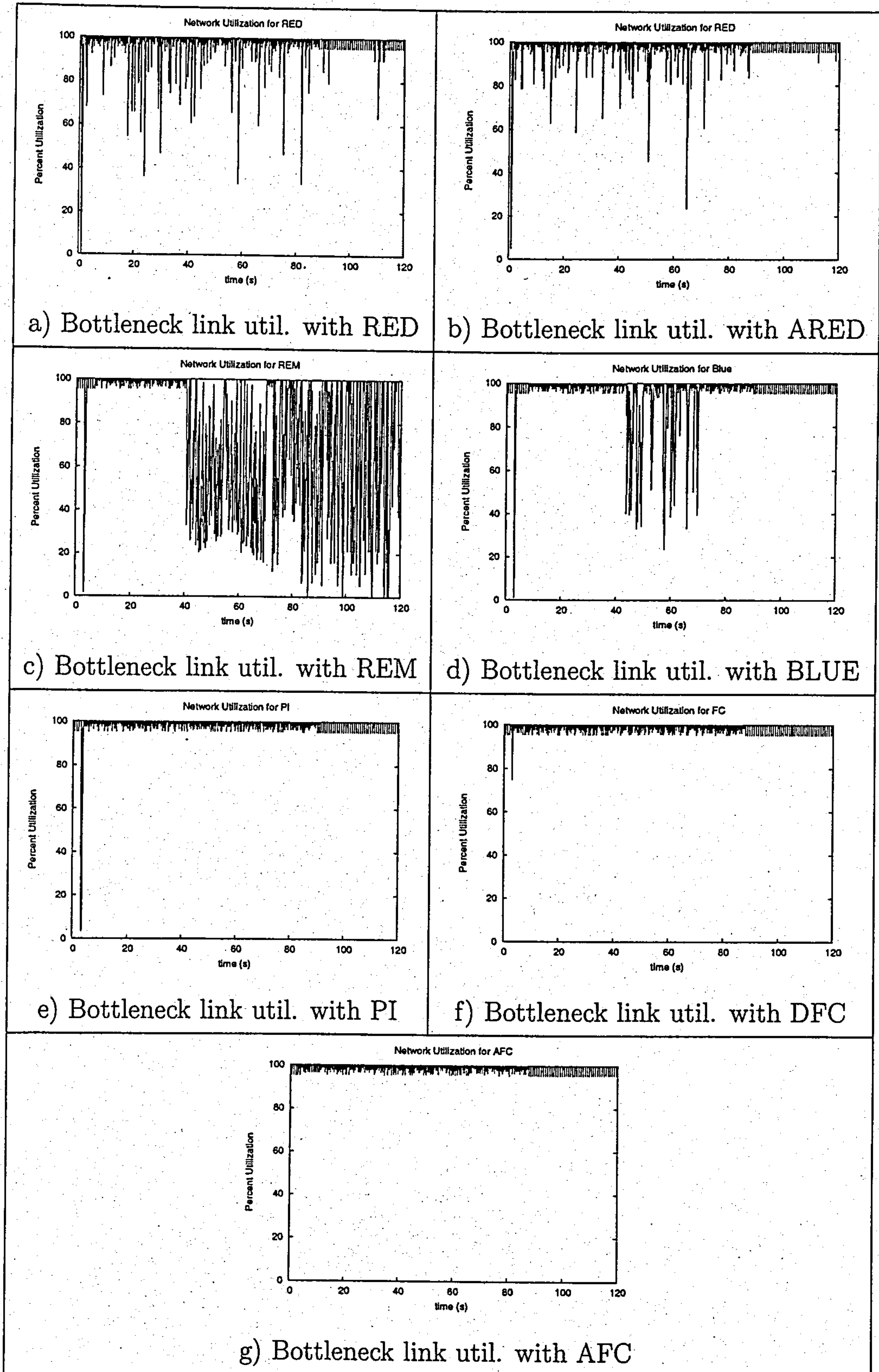
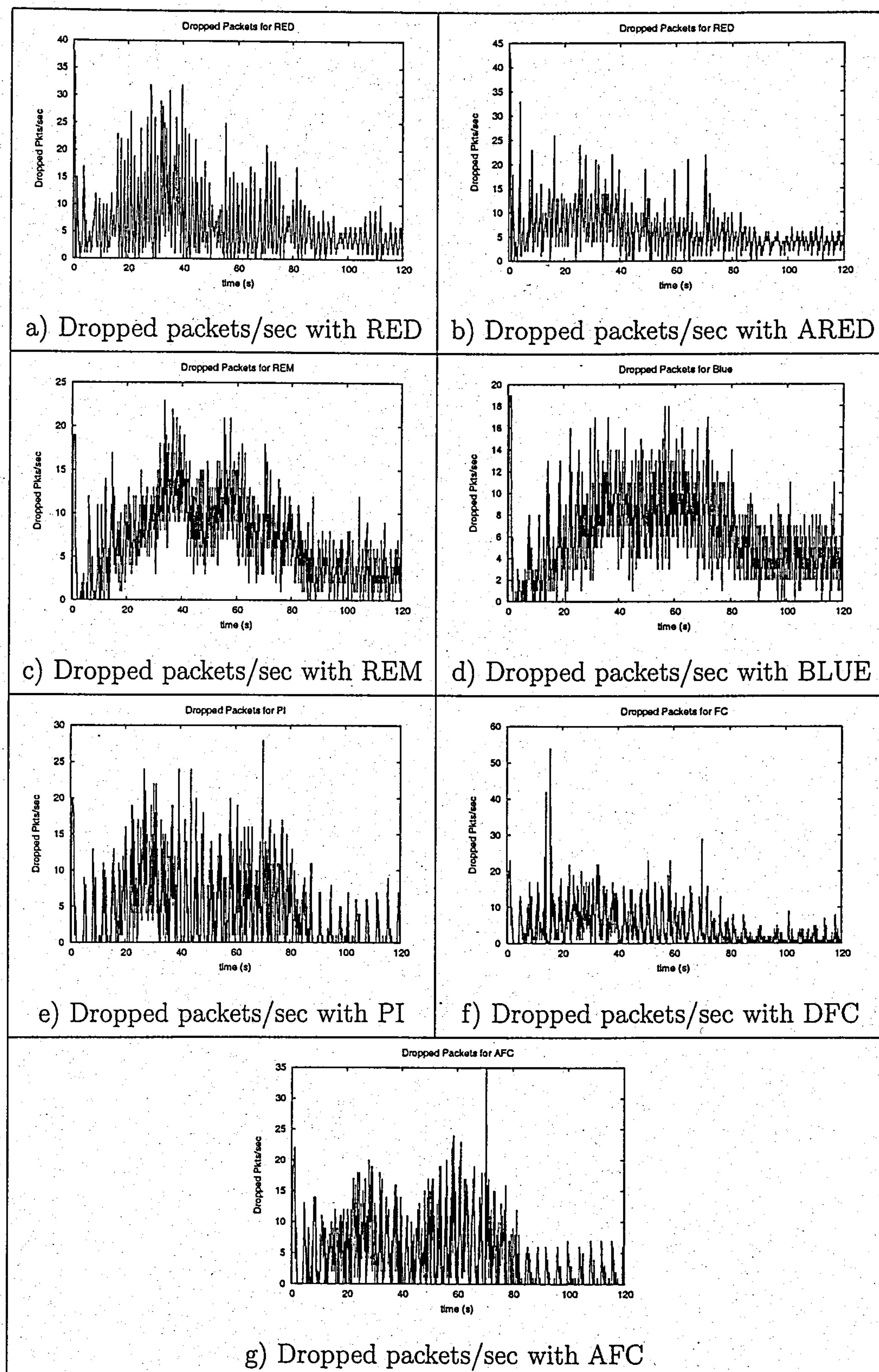


Figure 5.47. Comparison of bottleneck link utilization for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

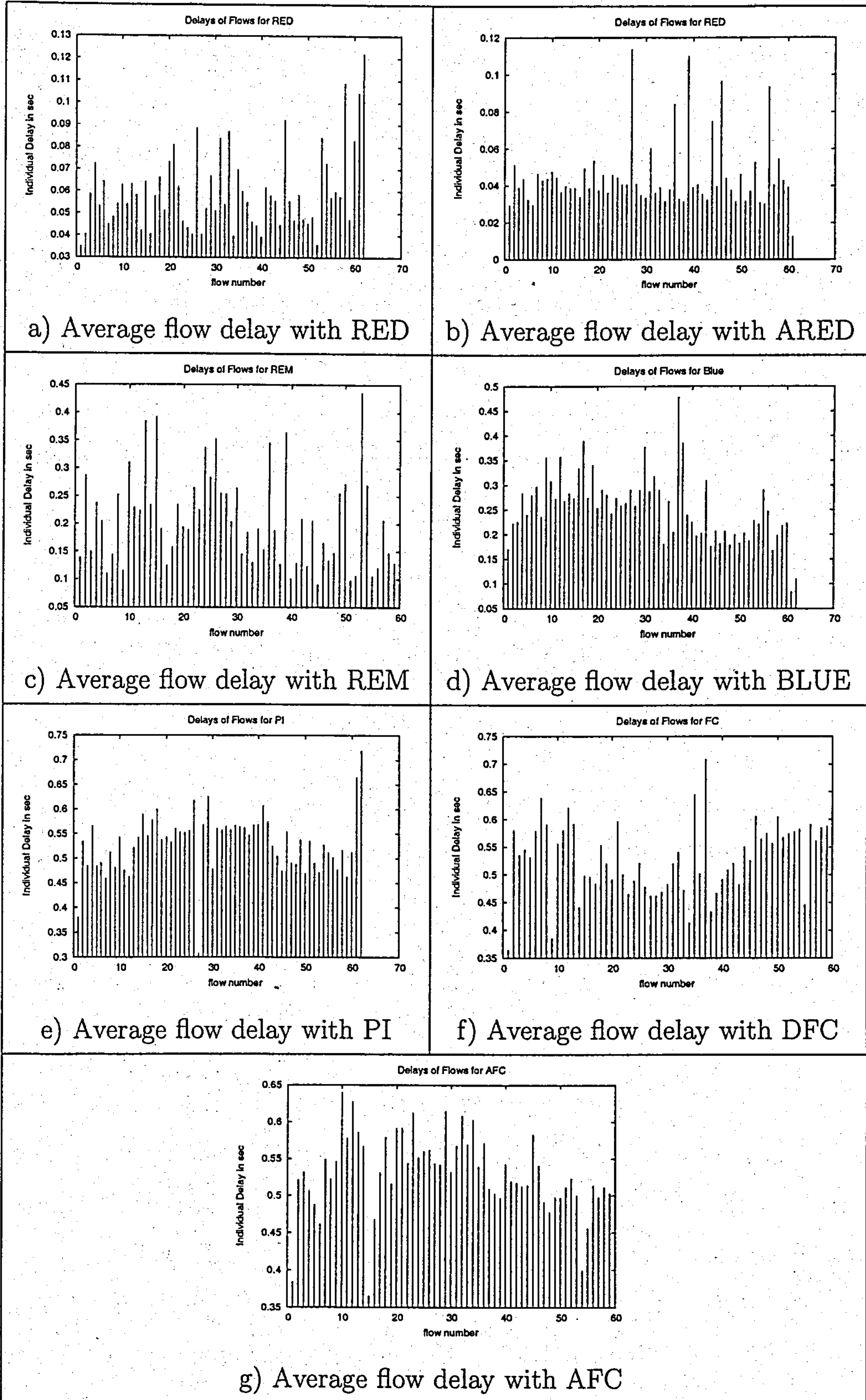
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.48. Comparison of dropped packets/sec for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic



a) Average flow delay with RED
b) Average flow delay with ARED
c) Average flow delay with REM
d) Average flow delay with BLUE
e) Average flow delay with PI
f) Average flow delay with DFC
g) Average flow delay with AFC

Figure 5.49. Comparison of average individual flow delays for $buffer = 200$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic, CBR traffic and WEB traffic

5.2.5. Scenarios with Changed Buffer Size

The effect of changing buffer size to 150 means to drop more packets for the AQM mechanisms to keep track of the reference value for the queue length.

The first scenario is performed with 100 nodes and 5 ms of bottleneck delay. This case is also divided into two, one having only dynamic FTP traffic and the other one having CBR traffic added onto it.

If we focus on the "dynamic FTP only" case, it would be better to compare it with the same scenario and the only difference being $buffer = 200$. Figure 5.50 shows that all mechanisms behave similar compared to the case, where buffer size is 200. For RED and ARED, there is seemingly no difference, REM and DFC perform slightly better, PI and BLUE slightly worse and AFC has less fluctuations at the beginning of the simulation, but a bit more towards the end. Bottleneck link utilization, which can be observed in Figure 5.51 is also similar for both buffer sizes, maximum utilization is achieved with DFC and AFC mechanisms. The slow start drawback of PI can again be observed. For the fairness criterion, Jain's index and the delay variation for individual flows shown in Figure 5.53 suggest that the most fair approach being DFC. In this scenario, PI drops the least packets and AFC is the second one.

If CBR traffic is added, it can be observed from Figure 5.54 that RED and ARED behave similar and BLUE and REM perform slightly better in terms of reference value tracking compared to the case, where $buffer = 200$. DFC has similar fluctuations and performs well, AFC and PI show slightly more fluctuations, but AFC is performing better than PI. Figure 5.55 also demonstrates the slow start problem of PI, as far as utilization is concerned, DFC and AFC are the best achieving maximum utilization. The fairness metric again favors DFC mechanism, both with Jain's index and delay variation shown in Figure 5.57, however this time the least amount of packets are dropped by AFC.

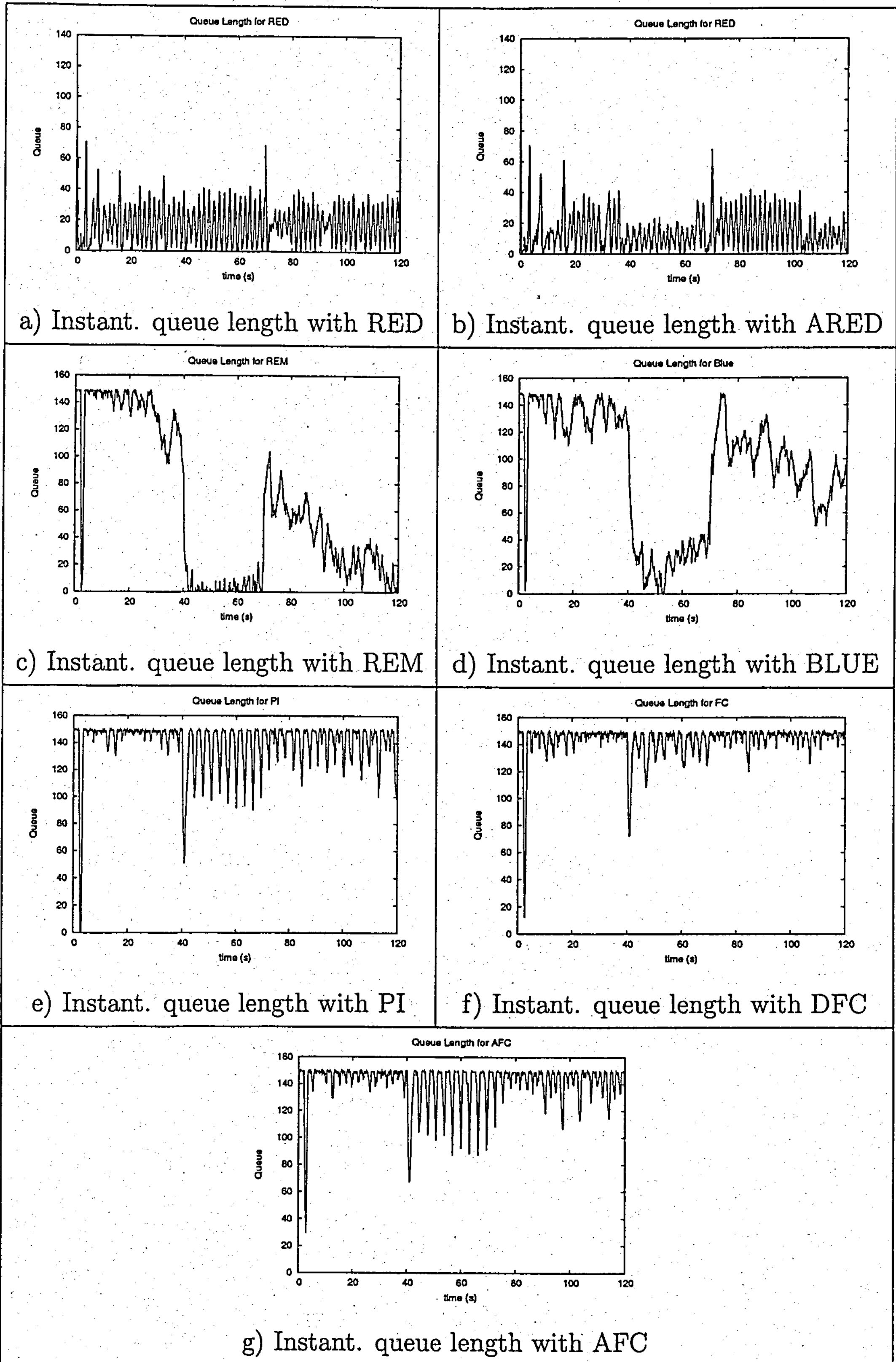
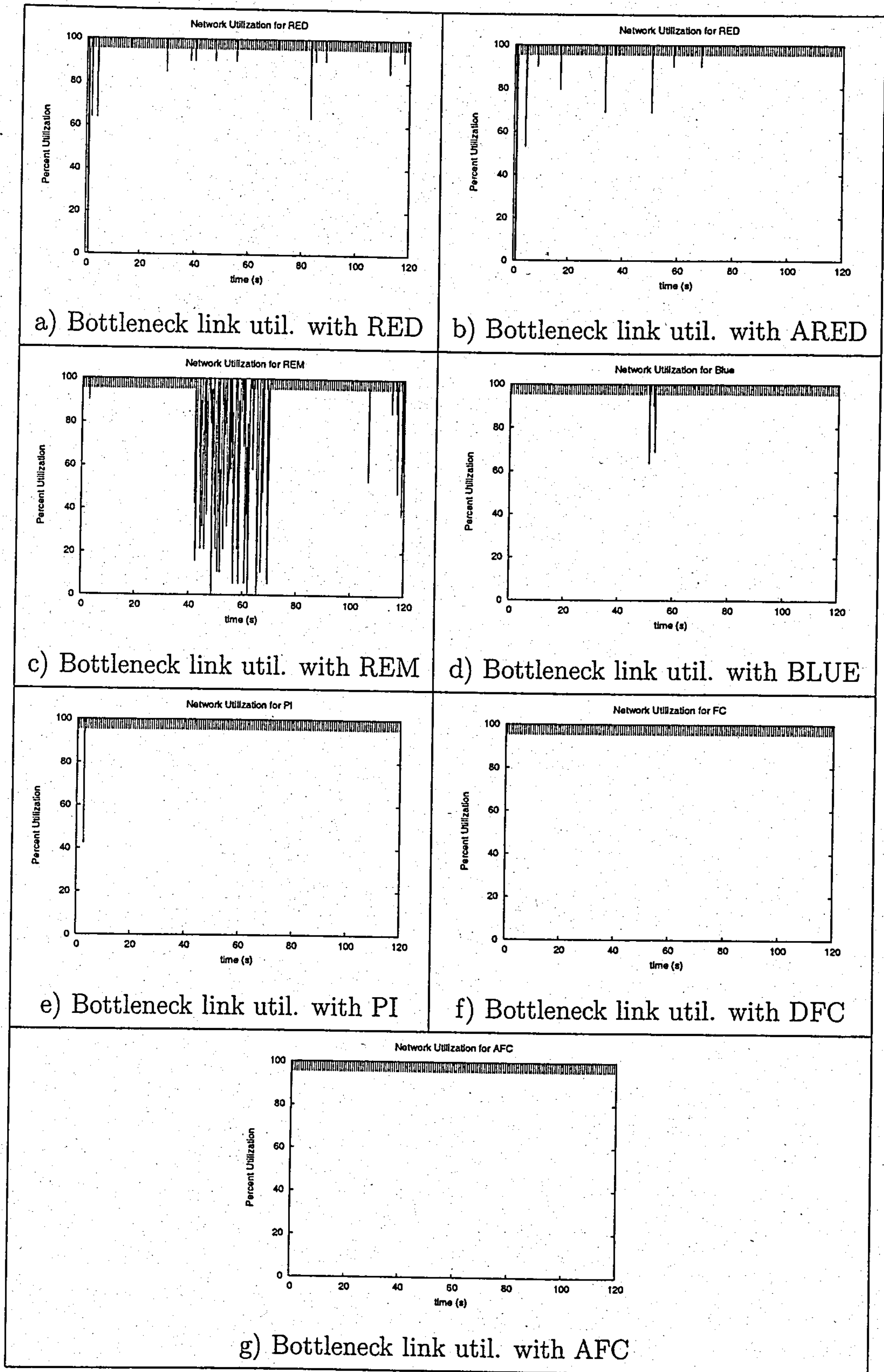


Figure 5.50. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

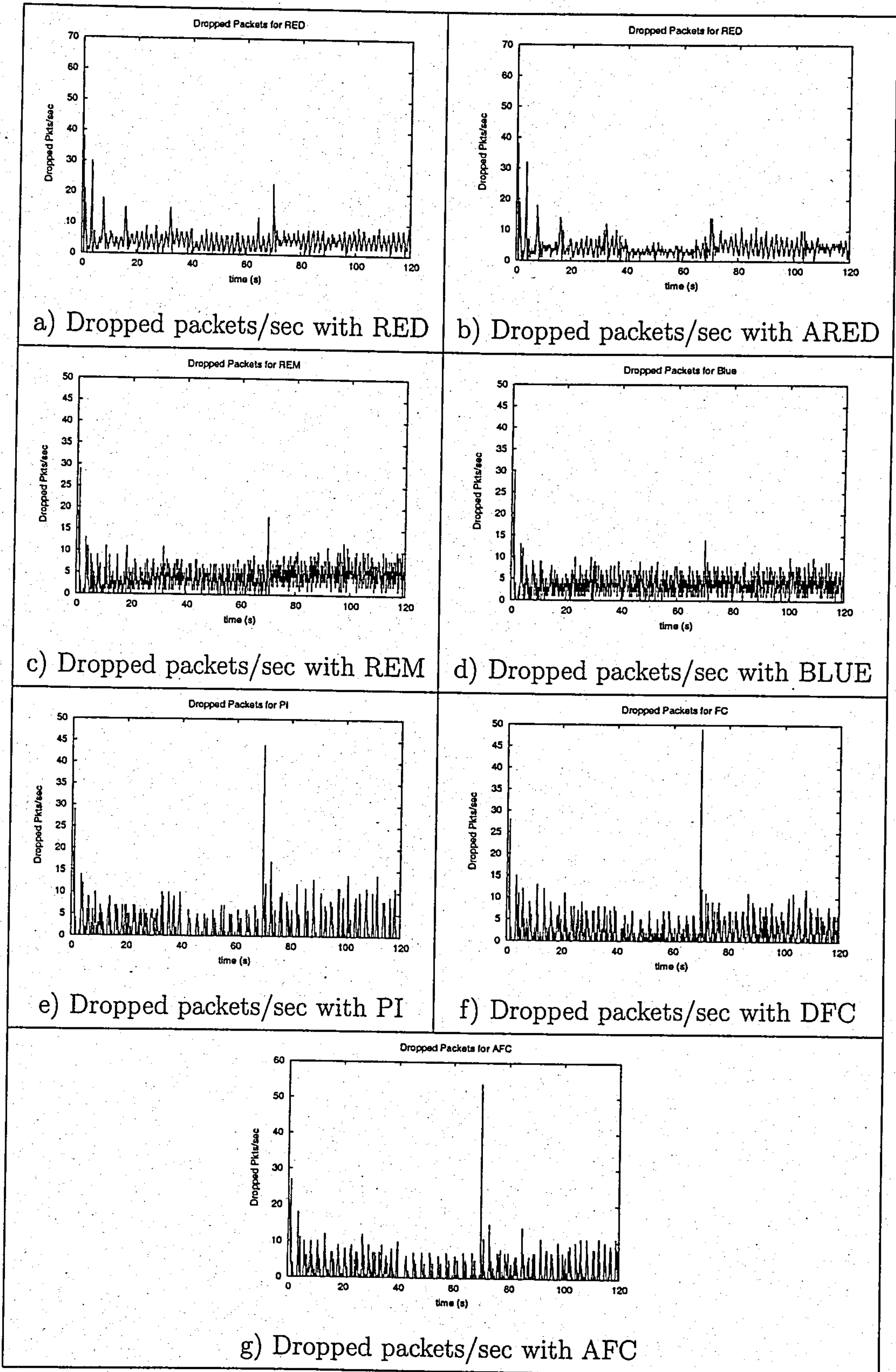
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.51. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

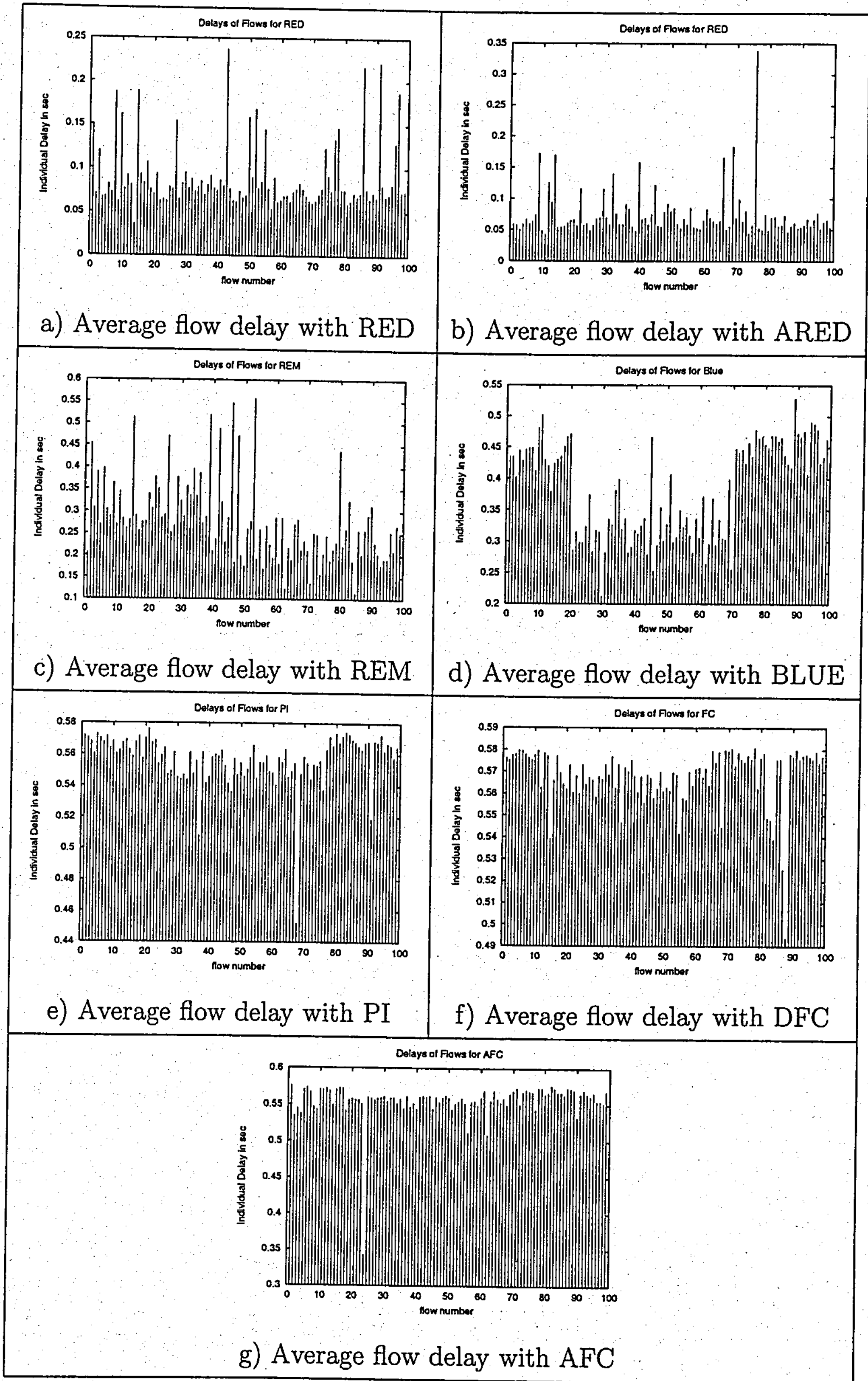
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.52. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic



a) Average flow delay with RED
b) Average flow delay with ARED
c) Average flow delay with REM
d) Average flow delay with BLUE
e) Average flow delay with PI
f) Average flow delay with DFC
g) Average flow delay with AFC

Figure 5.53. Comparison of average individual flow delays for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms and dynamic FTP traffic

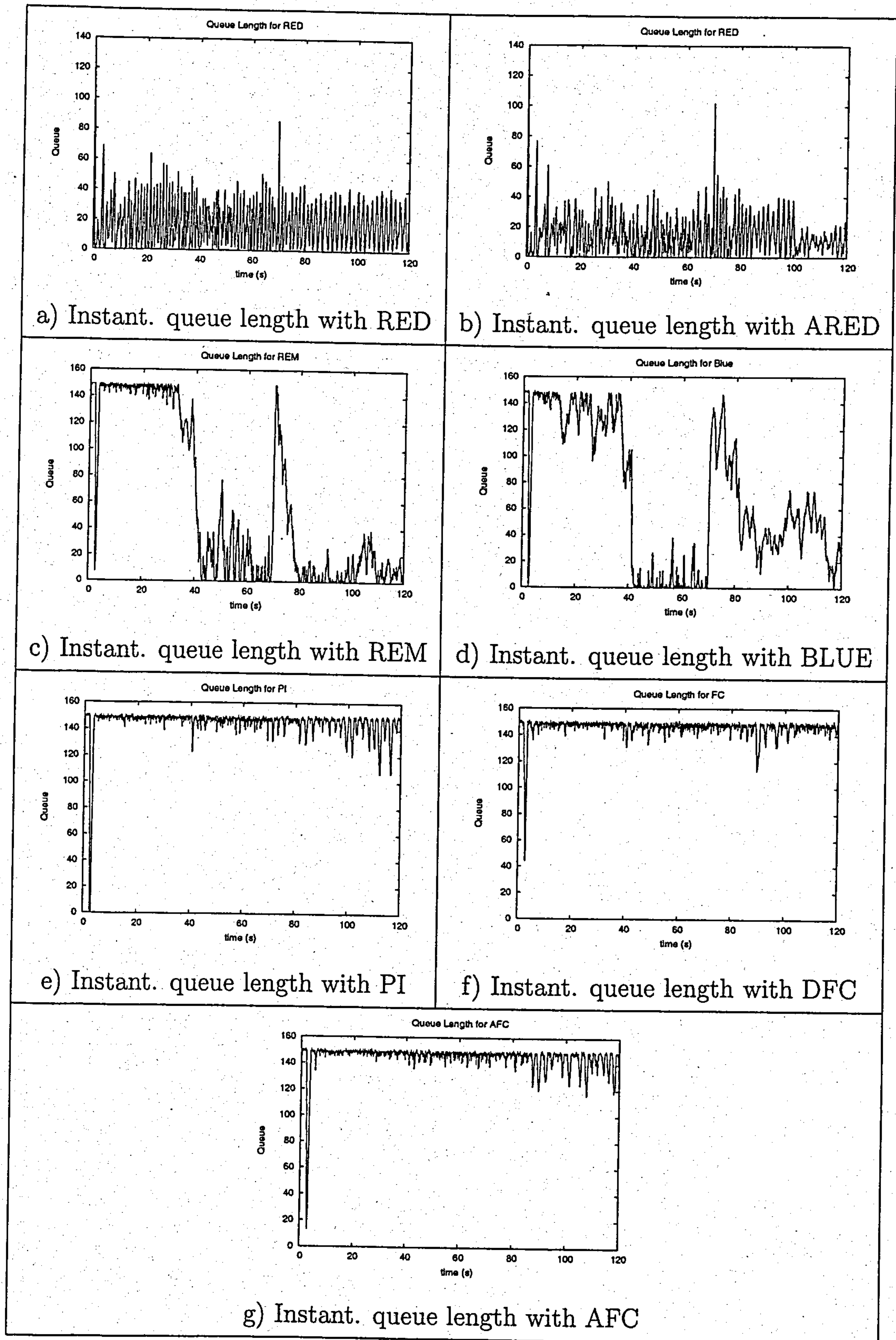


Figure 5.54. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic

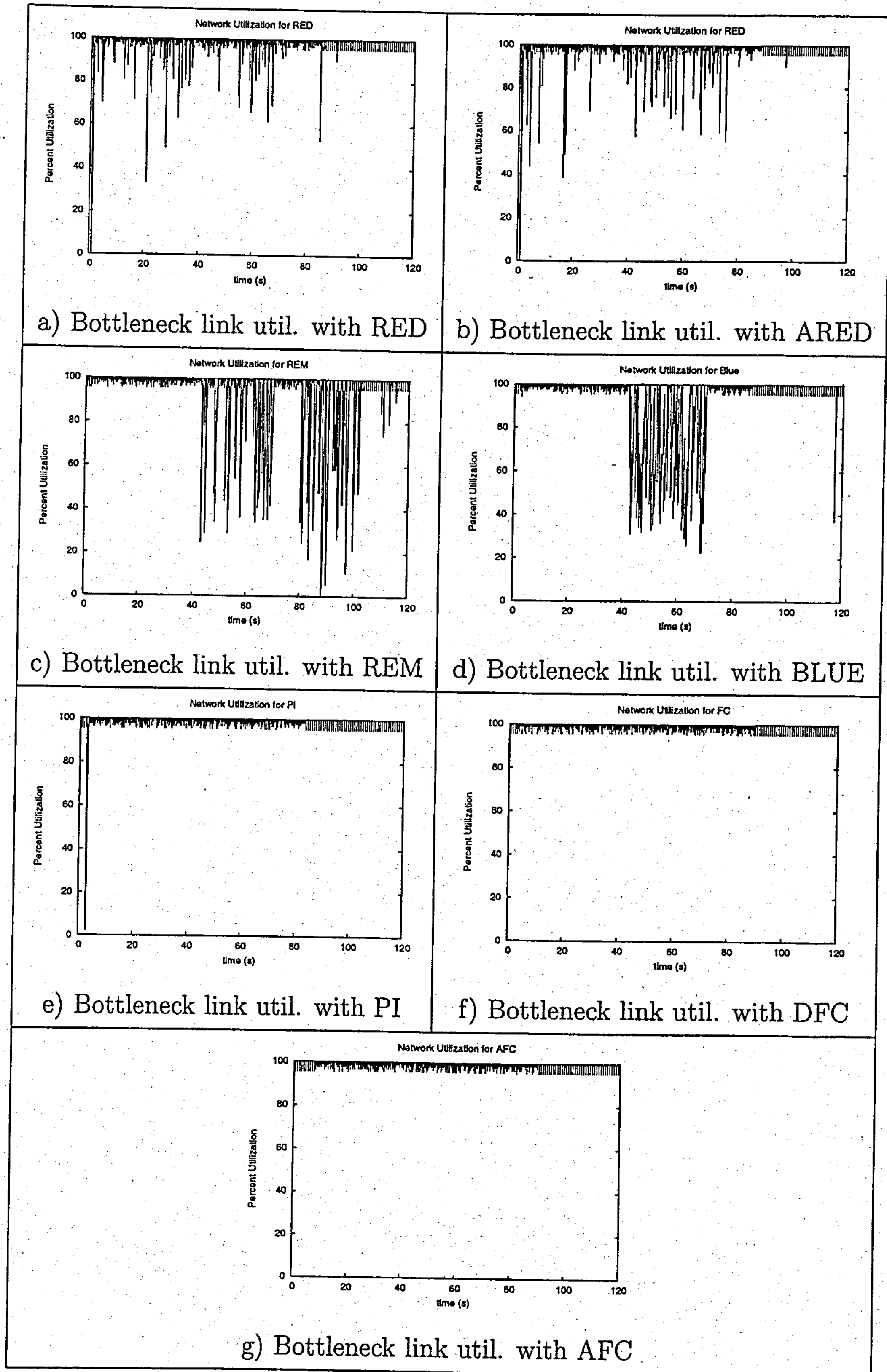
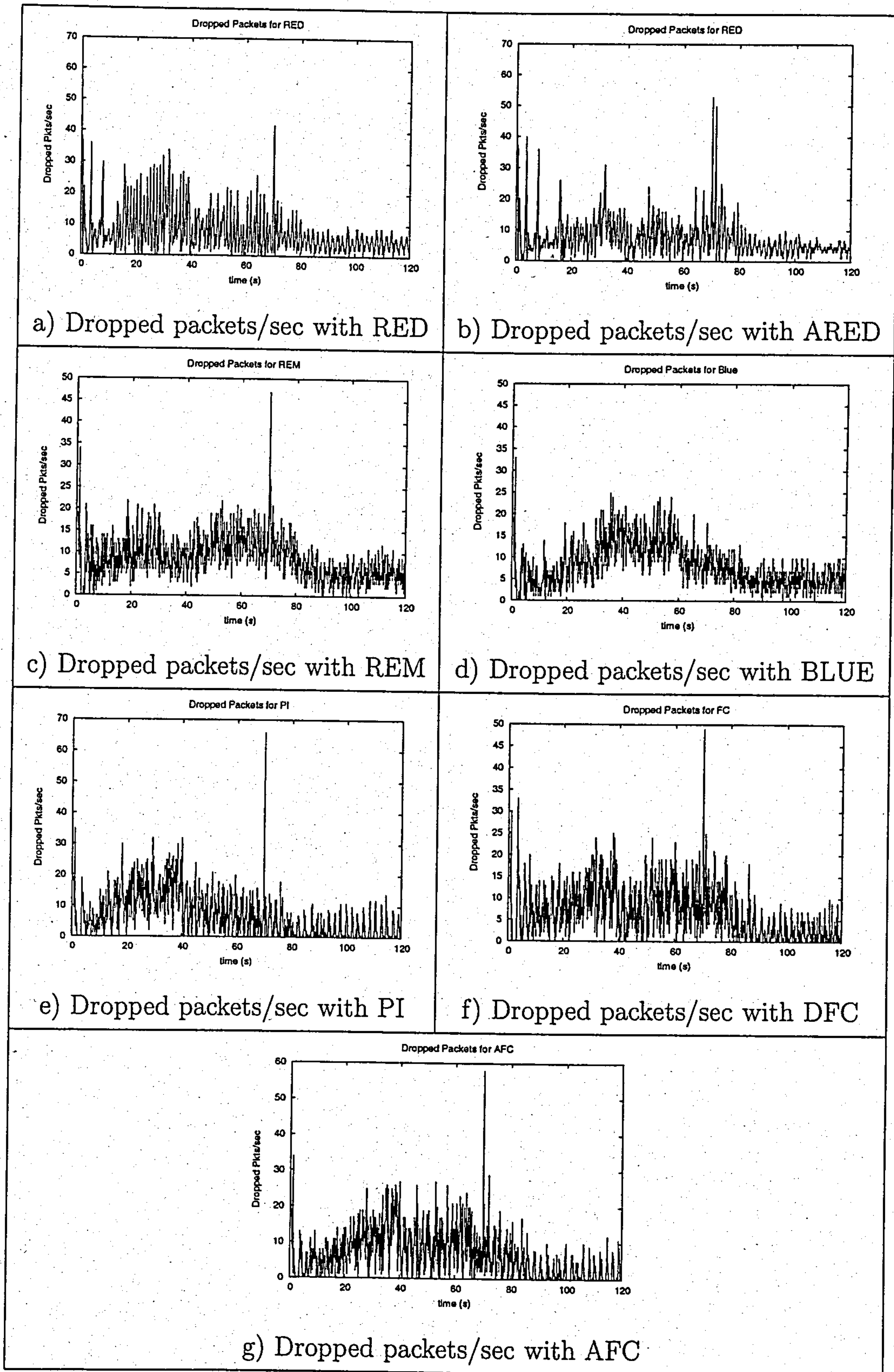


Figure 5.55. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.56. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic

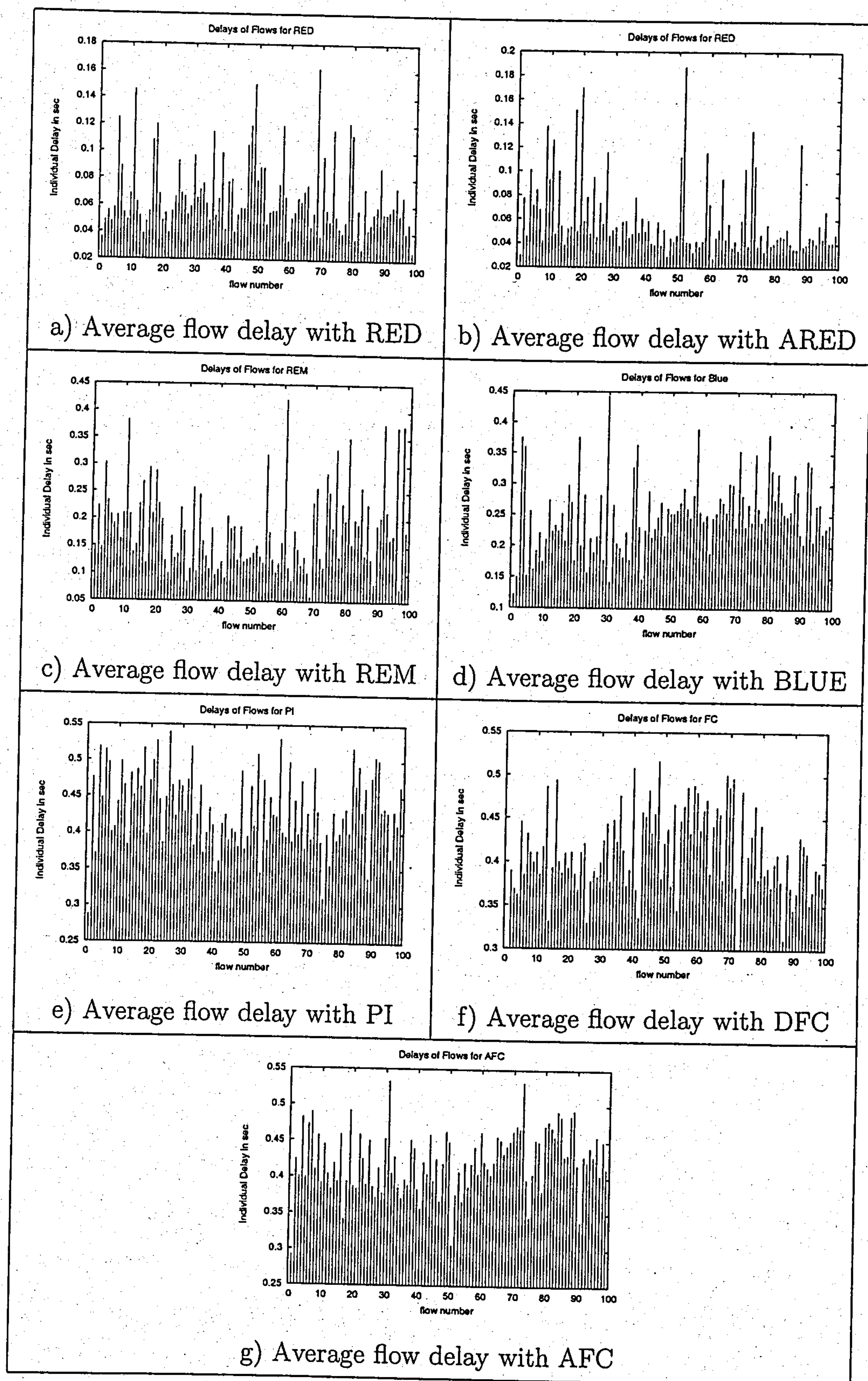
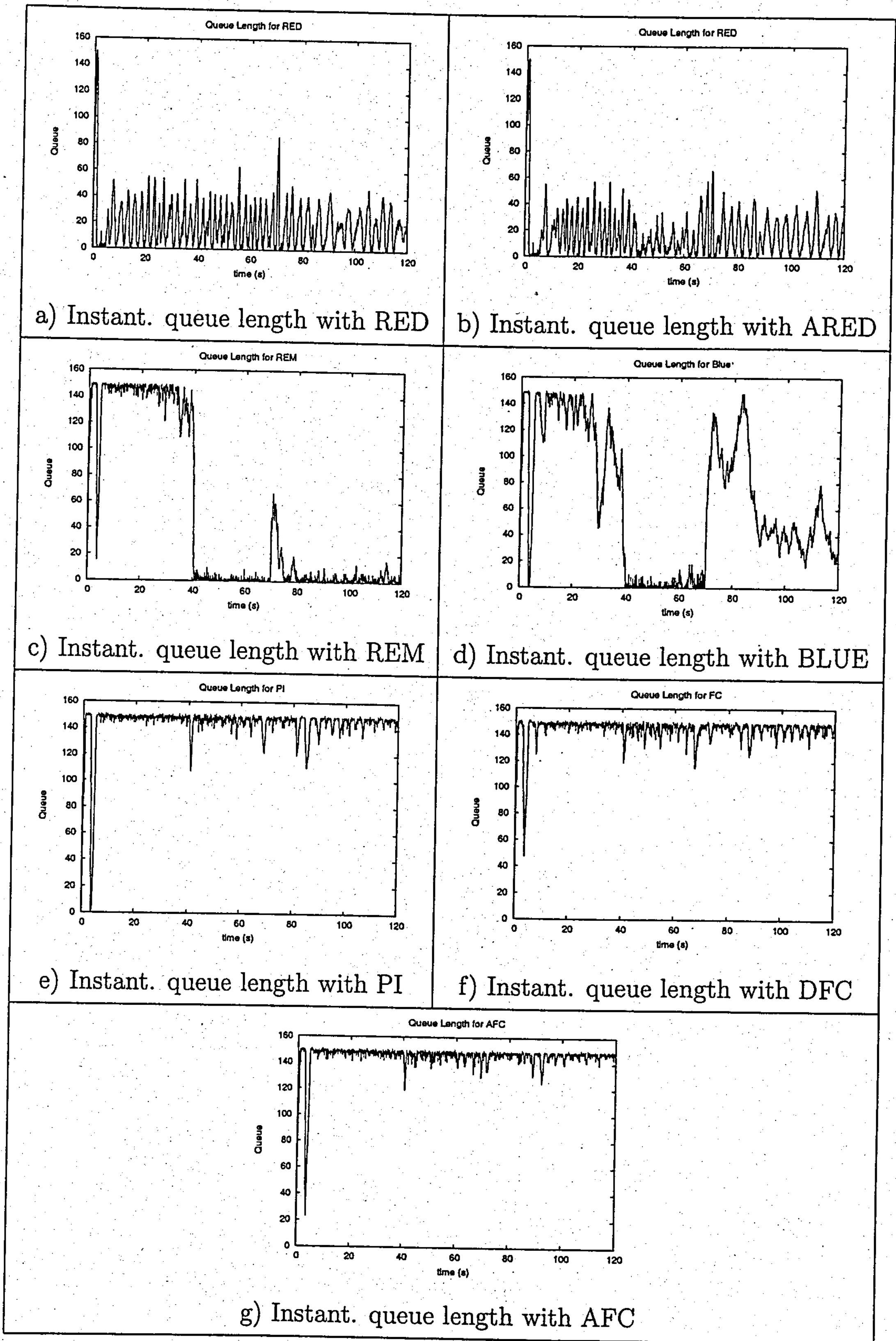


Figure 5.57. Comparison of average individual flow delays for $buffer = 150$, $nodes = 100$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic

When the CBR traffic remains and the bottleneck delay is changed to 125 ms, Figure 5.58 shows almost no difference compared to the $buffer = 200$ case in terms of queue fluctuation, however, the performance of DFC and AFC is the best. PI fluctuates a bit more compared to DFC and AFC. If we look at the bottleneck link utilization, shown in Figure 5.59, it can be observed that RED and ARED are more utilized compared to $buffer = 200$ case and PI again lacks to capture slow start dynamics. Jain's index suggests that BLUE being the most fair, with a value of 0.87, however, this is not the case if we look at the delay variation in Figure 5.61. Among the good performing algorithms DFC, AFC and PI, AFC has the highest Jain's index. RED has the lowest deviation of individual delays, but it does perform very badly in general.

There are two other cases, where the number of nodes is changed to 60 and with $buffer = 150$, one with 5 ms of bottleneck delay, the other one with 125 ms. If we look at the case with a delay of 5 ms, Figure 5.62 shows that RED, ARED and REM show similar behavior, BLUE, PI, DFC and AFC performs better compared to the case, where $buffer = 200$. AFC achieves again the best queue tracking with the best slow start behavior. It can be observed from Figure 5.63 that network utilization is higher with PI, DFC and AFC and PI is having again difficulties during slow start. Maximum utilization is achieved by DFC and AFC only. The utilization for RED, ARED, REm and BLUE is similar to the case when using those mechanism, where $buffer = 200$. BLUE, DFC and AFC have the highest fairness index in decreasing order. BLUE performs badly if focused on the delay variation shown in Figure 5.65. Also note that the least amount of packet drops is achieved by the AFC algorithm.

When the delay is increased to 125 ms for the same setup, Figure 5.66 shows that DFC performs better compared to the case, where $buffer = 200$ and AFC shows less fluctuations for this case. Network utilization shown in Figure 5.67 is similar to the "$buffer = 200$" case and AFC and DFC utilize the bottleneck link more than the others. AFC also performs the best in dropping the least amount of packets and it is more fair towards the connections, which can be also observed in Figure 5.69 with the least amount of individual delay variation. However, Jain's index is the highest with BLUE.



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

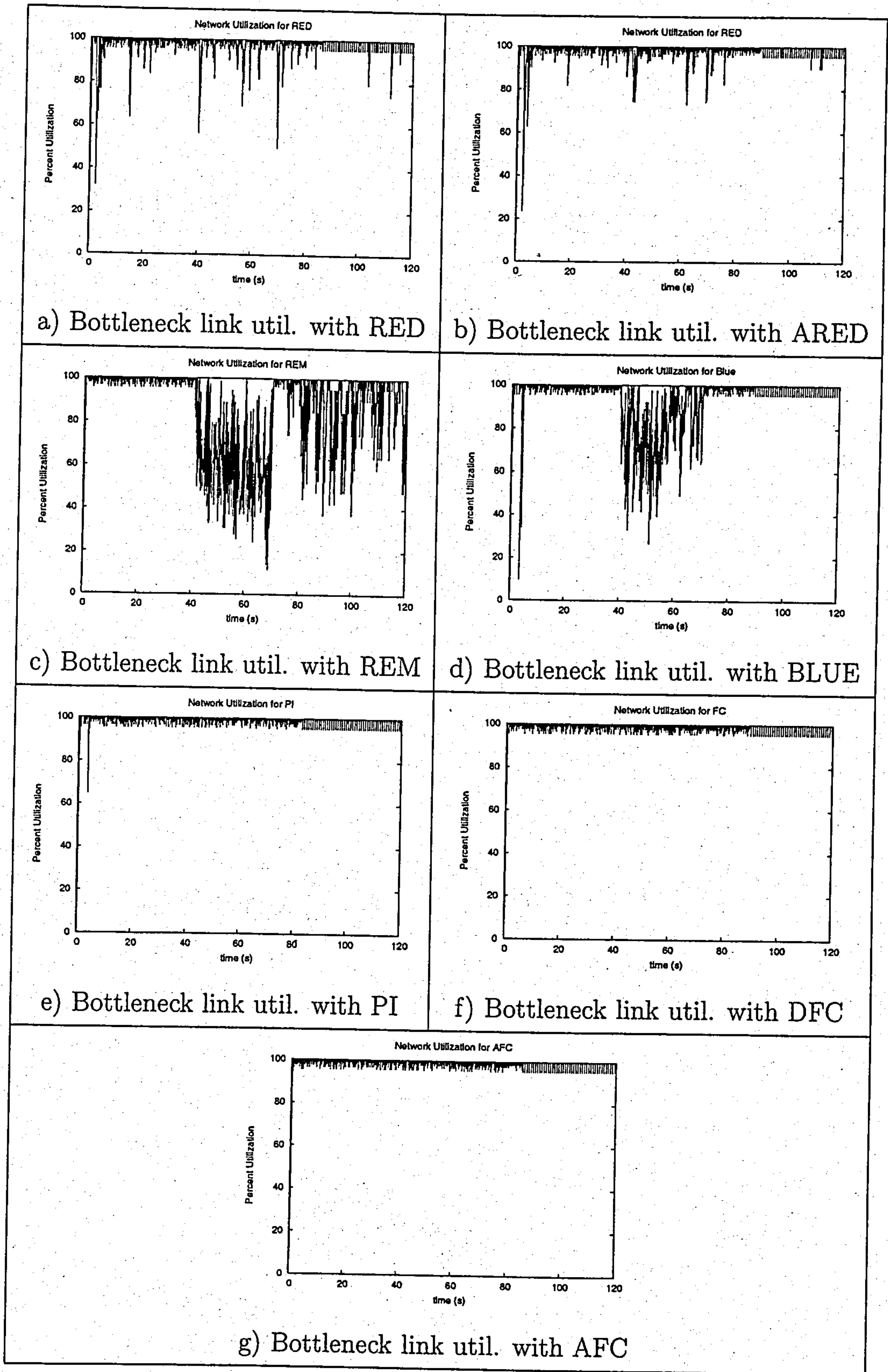
d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.58. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

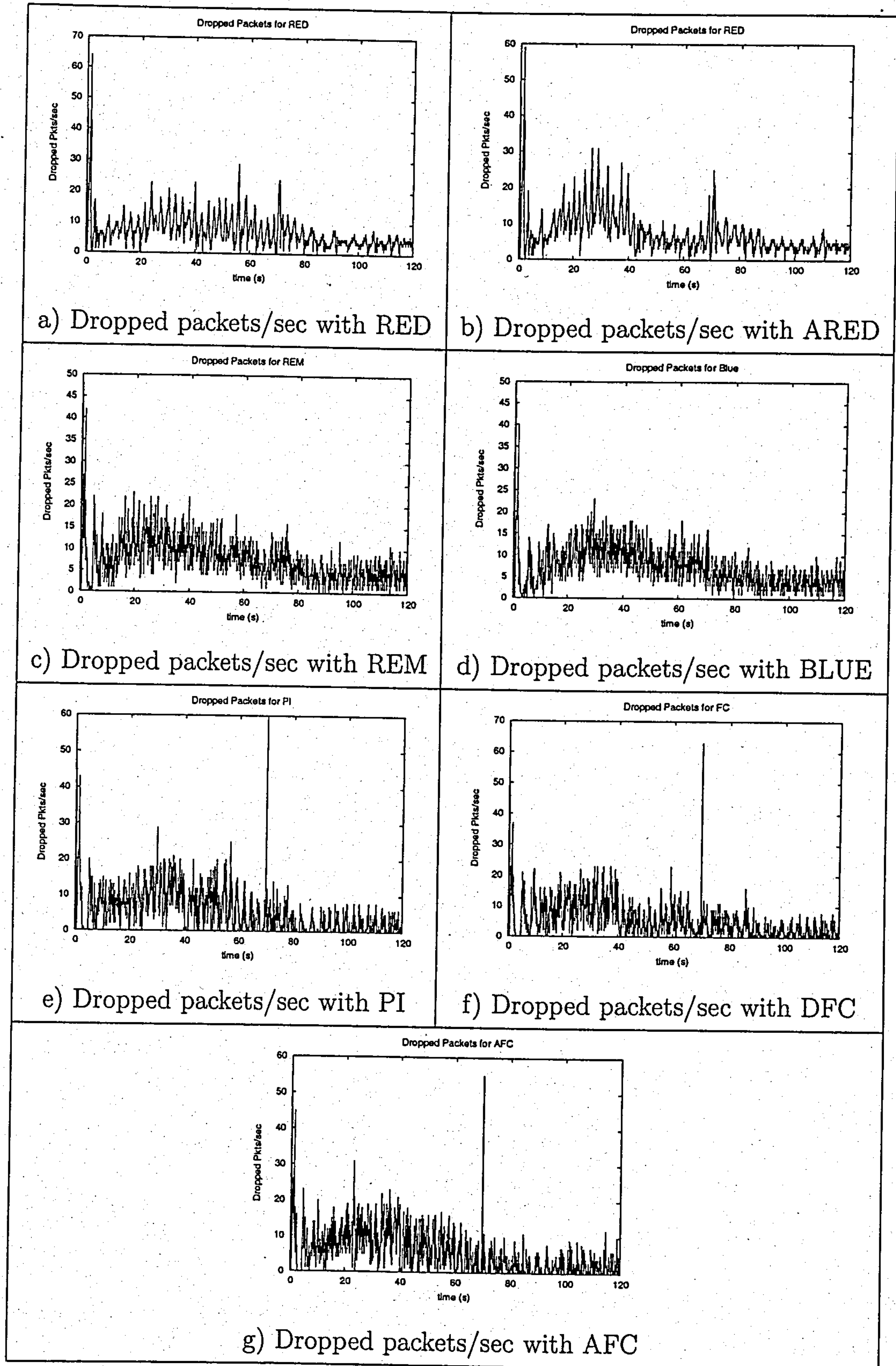
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.59. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

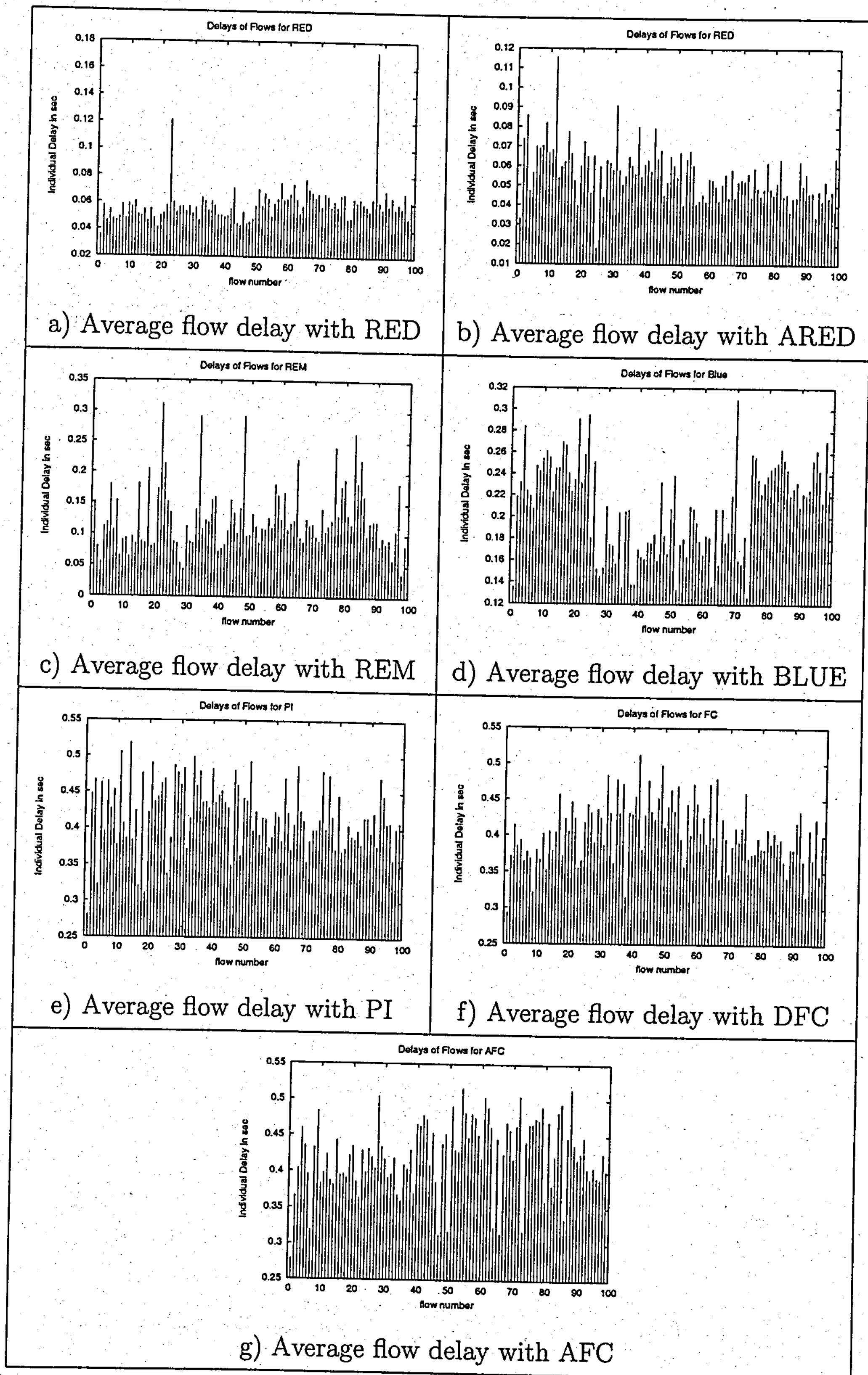
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.60. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

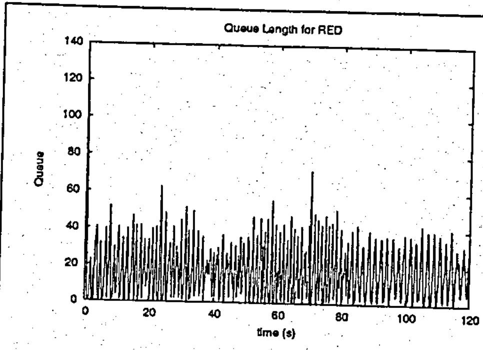
d) Average flow delay with BLUE

e) Average flow delay with PI

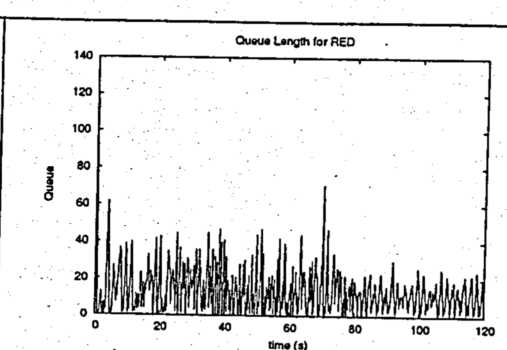
f) Average flow delay with DFC

g) Average flow delay with AFC

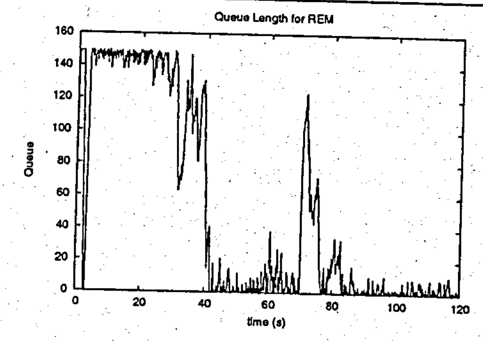
Figure 5.61. Comparison of average individual flow delays for $buffer = 150$, $nodes = 100$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



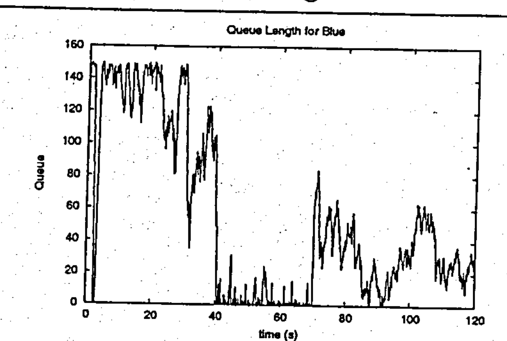
a) Instant. queue length with RED



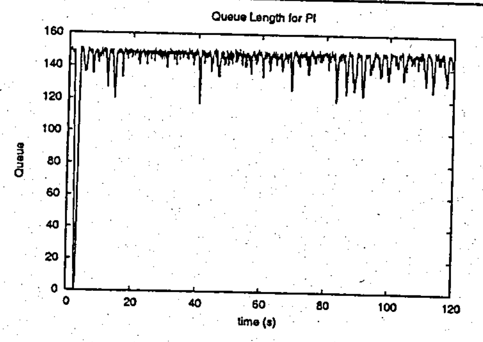
b) Instant. queue length with ARED



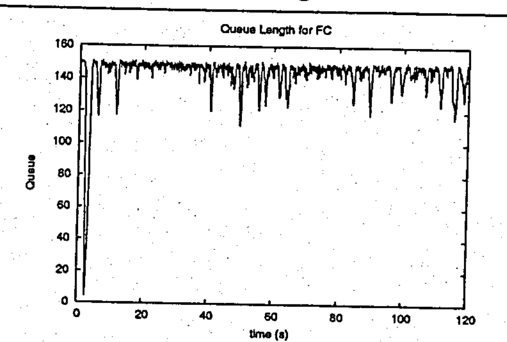
c) Instant. queue length with REM



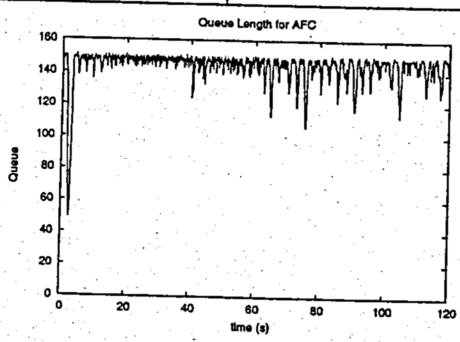
d) Instant. queue length with BLUE



e) Instant. queue length with PI

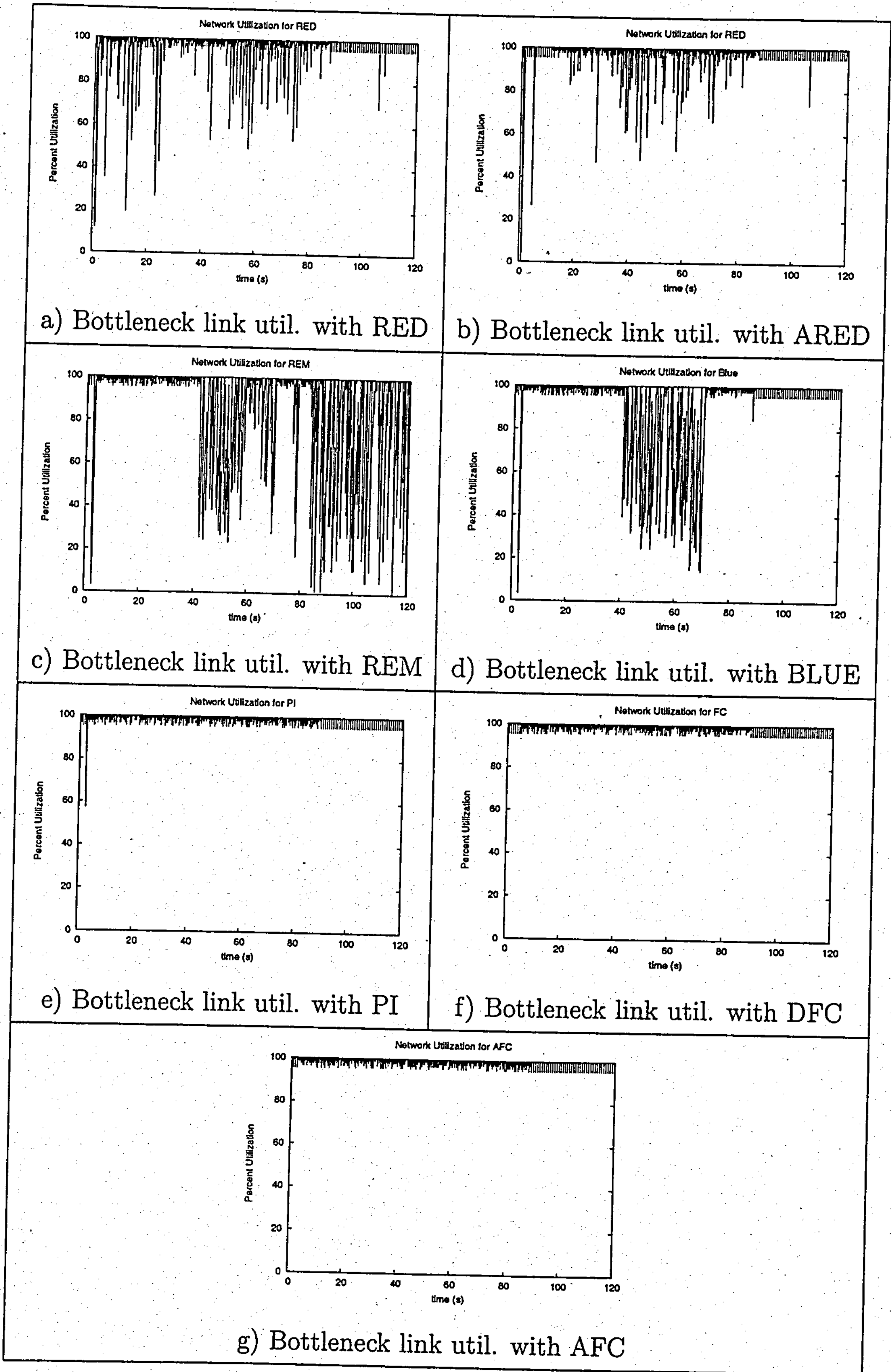


f) Instant. queue length with DFC



g) Instant. queue length with AFC

Figure 5.62. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

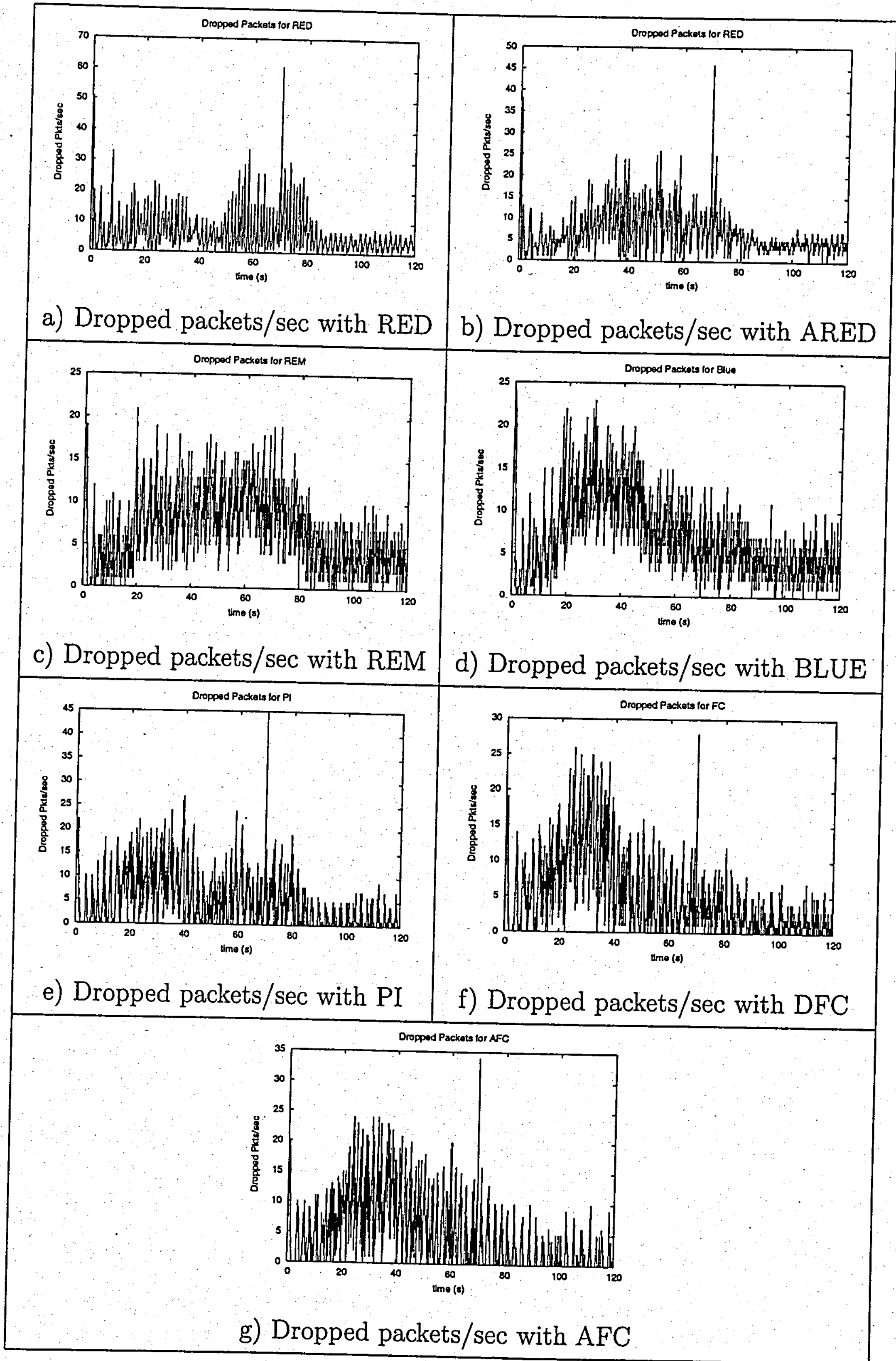
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.63. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

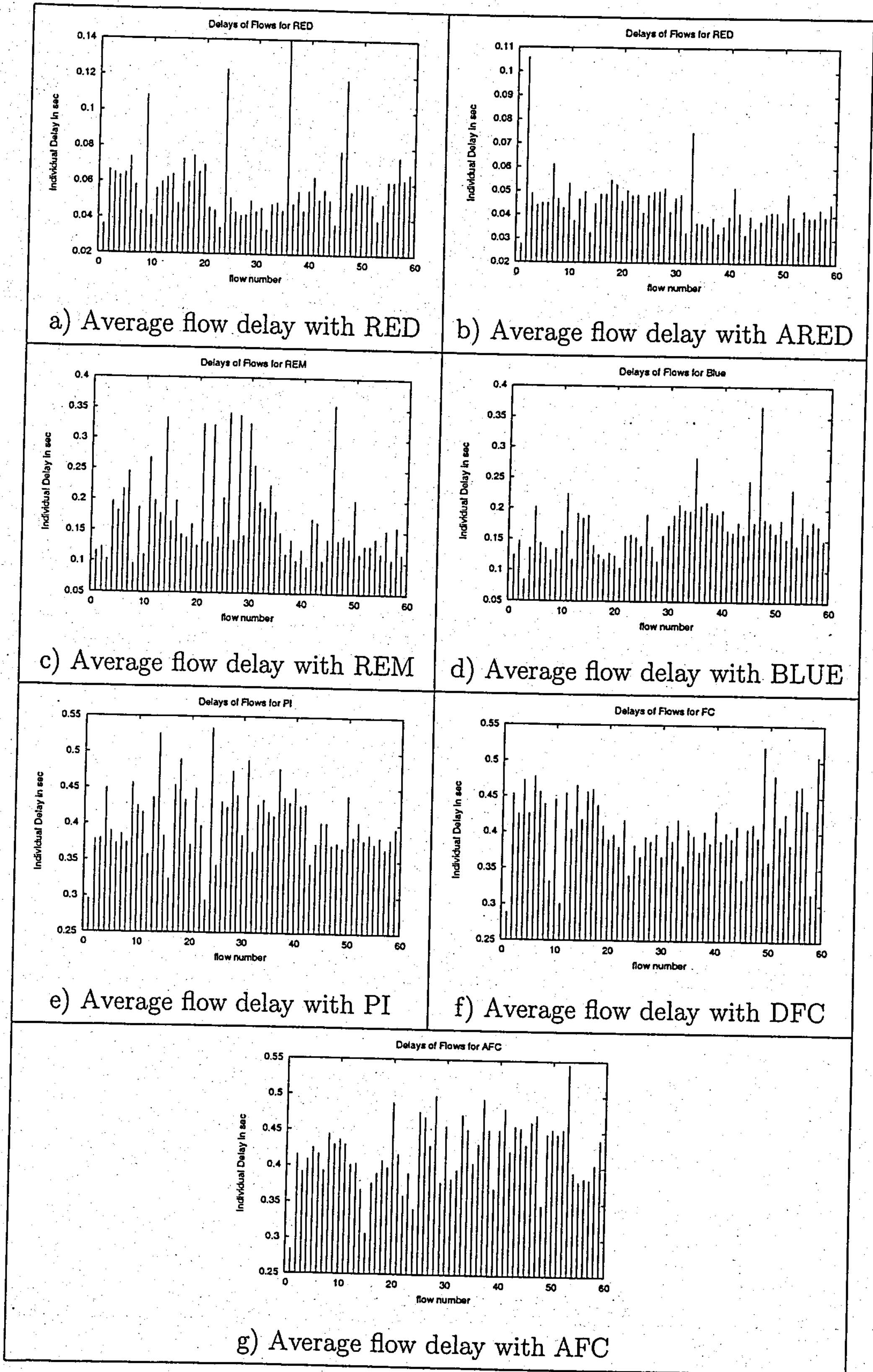
d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.64. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

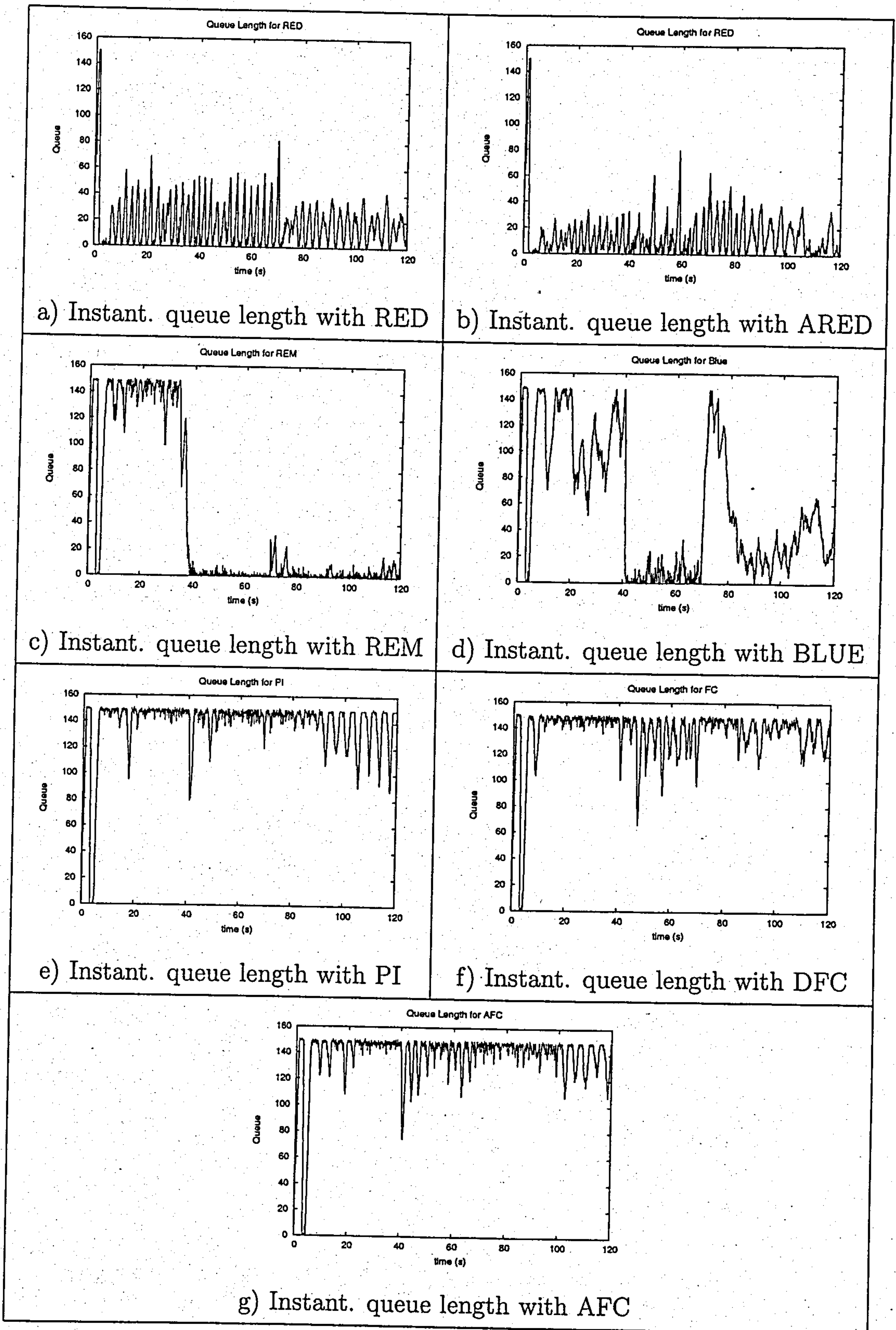
d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

g) Average flow delay with AFC

Figure 5.65. Comparison of average individual flow delays for $buffer = 150$, $nodes = 60$, a bottleneck delay of 5 ms, dynamic FTP traffic and CBR traffic



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

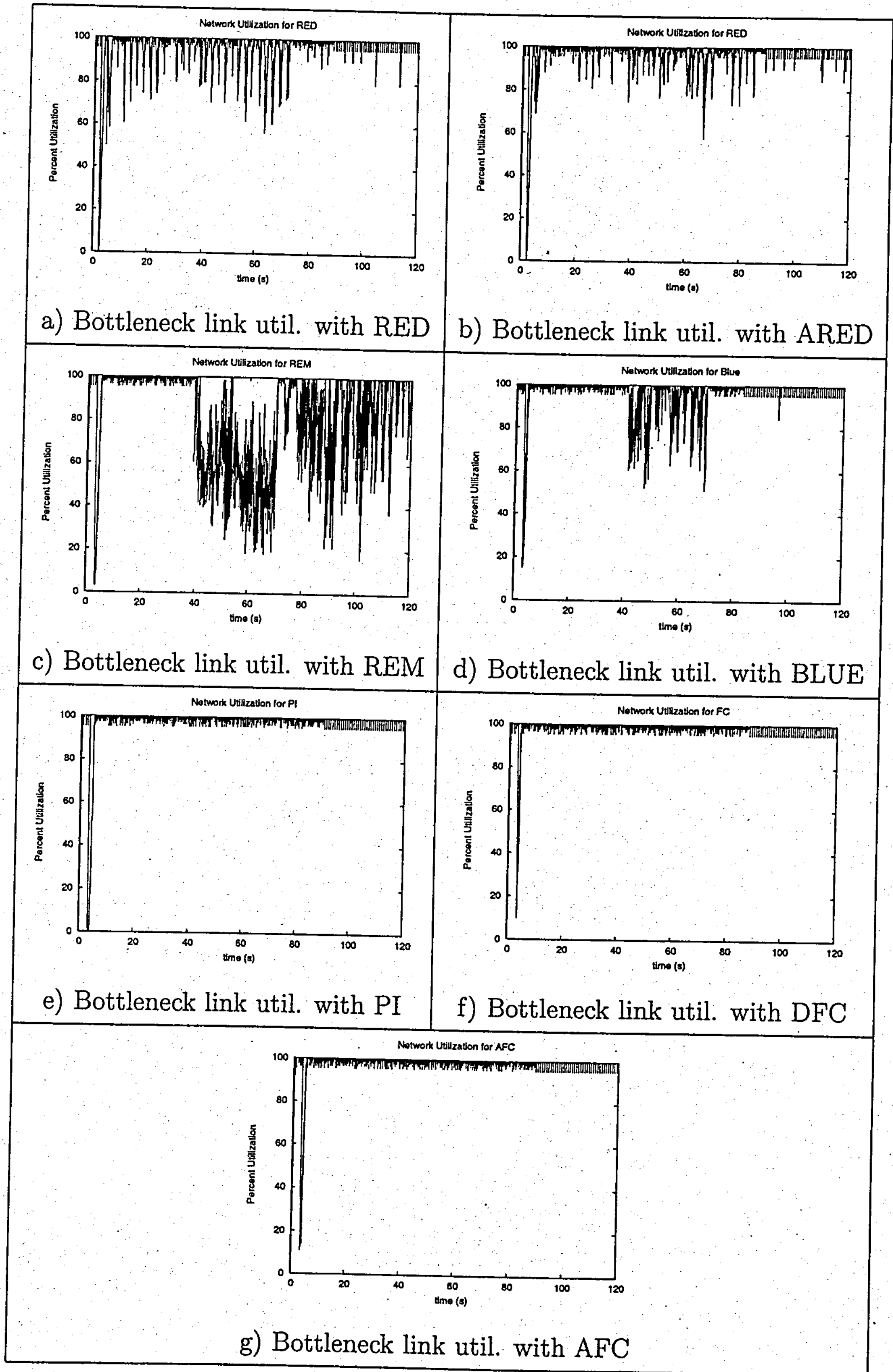
d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.66. Comparison of instantaneous queue lengths for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Bottleneck link util. with RED

b) Bottleneck link util. with ARED

c) Bottleneck link util. with REM

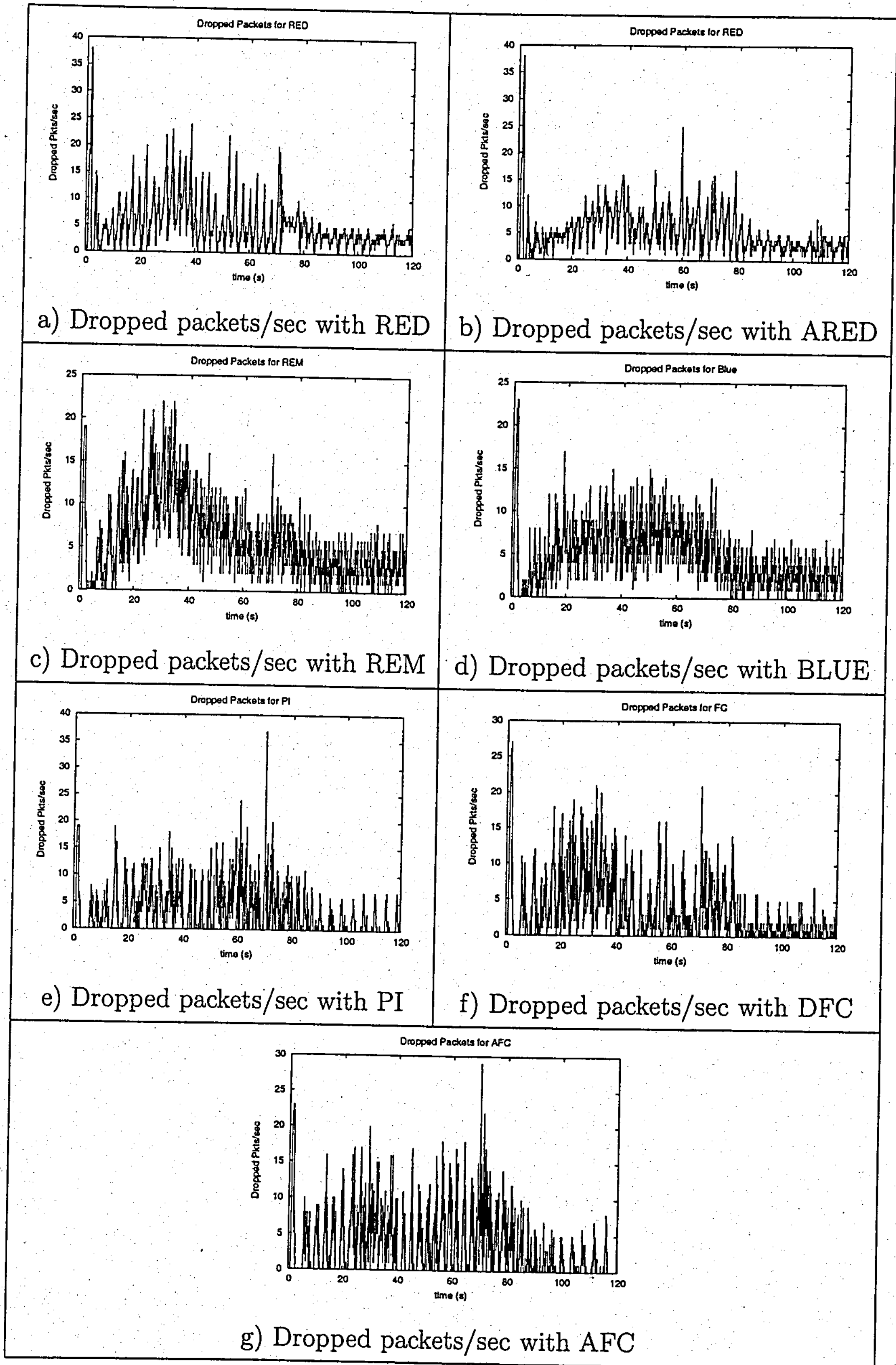
d) Bottleneck link util. with BLUE

e) Bottleneck link util. with PI

f) Bottleneck link util. with DFC

g) Bottleneck link util. with AFC

Figure 5.67. Comparison of bottleneck link utilization for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic



a) Dropped packets/sec with RED

b) Dropped packets/sec with ARED

c) Dropped packets/sec with REM

d) Dropped packets/sec with BLUE

e) Dropped packets/sec with PI

f) Dropped packets/sec with DFC

g) Dropped packets/sec with AFC

Figure 5.68. Comparison of dropped packets/sec for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic

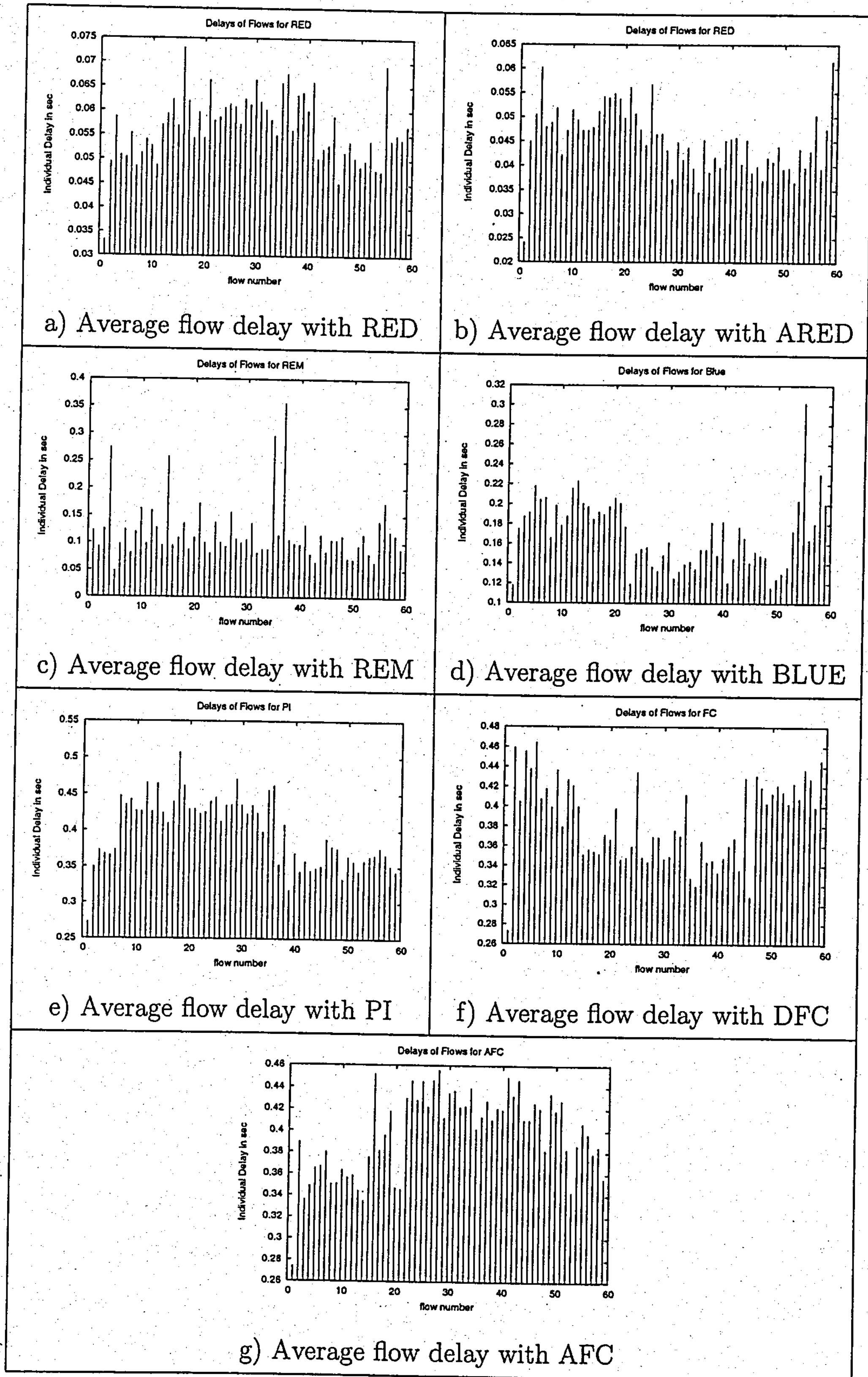


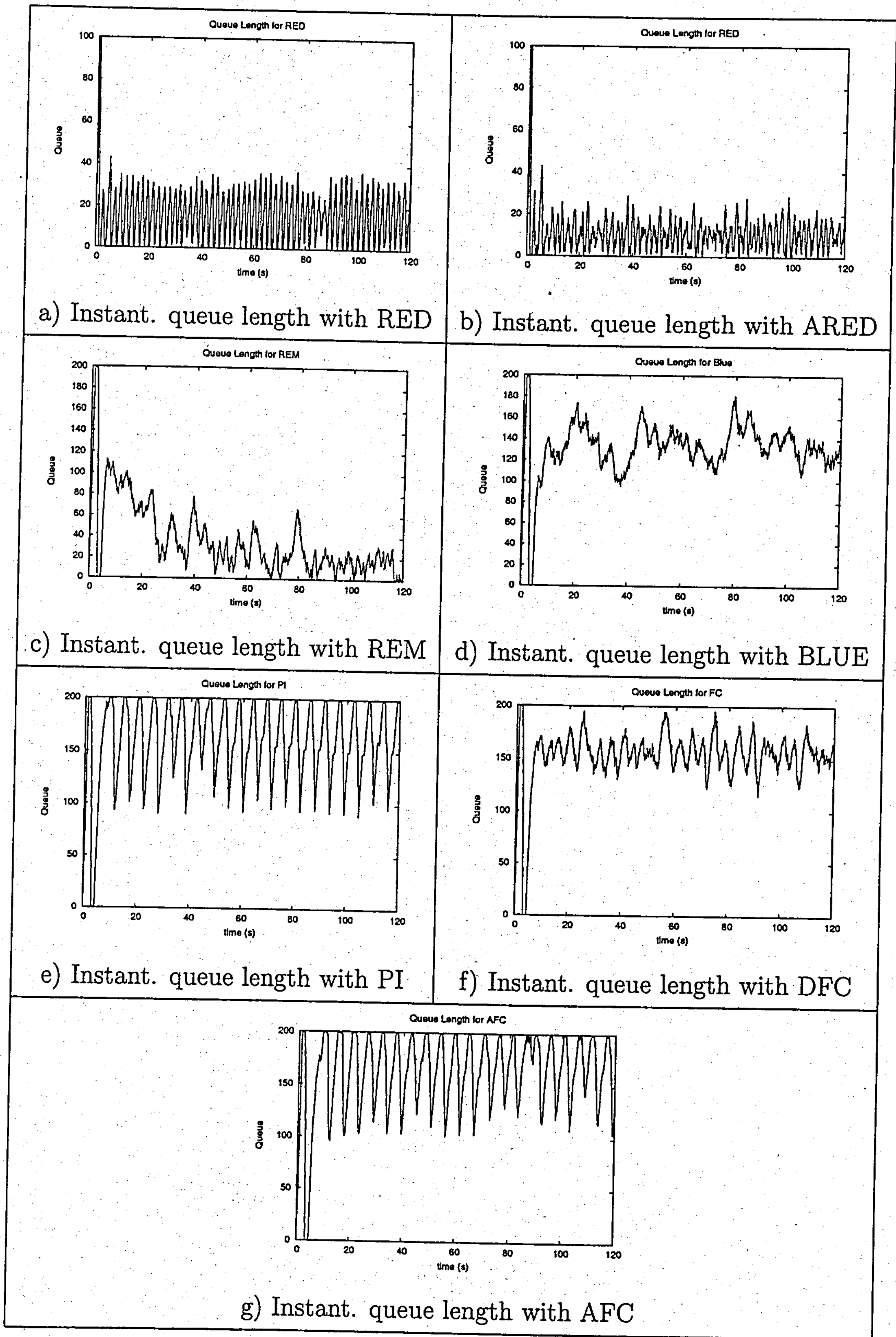
Figure 5.69. Comparison of average individual flow delays for $buffer = 150$, $nodes = 60$, a bottleneck delay of 125 ms, dynamic FTP traffic and CBR traffic

5.2.6. Scenarios with Wireless Cases

The scenarios are performed to see the behavior of the AQM mechanisms in a wireless environment. Although the setup does not reflect the scenario of today's wireless applications, in that sense the mobile nodes initiate FTP transfer, this is done to compare the performance of different AQM mechanisms. There are two wireless cases, one consisting of 30 nodes and the other one 60. Both cases only include a non-dynamic FTP traffic and a bottleneck delay of 5 ms.

For the both cases, the load on the router is not enough to fill the buffer all the time, however, RED, ARED and REM do not stabilize to a queue length. For the case with 30 nodes, BLUE, PI, DFC and AFC track a queue length, PI and AFC however, have more fluctuations, which is shown in Figure 5.70. The success of DFC over AFC arises from the aggressiveness of it compared to AFC. However, AFC can easily be made to behave similarly via decreasing the *cautious* parameter. This is also performed and shown via simulations, but not presented here to provide the same comparison standard without altering the values of the parameters. All mechanisms have similar fairness values, however, AFC drops the least amount of packets and its traffic fairness is higher with the DFC mechanism, which is shown in the delay variation of individual flows in Figure 5.73. In general, it can be said that DFC and AFC perform better than the others.

When the number of nodes is changed to 60, the generated traffic is enough for most of the times to use all the resources of the router. Queue length tracking is similar to the "30 nodes" case, however, BLUE experiences some difficulties to keep the queue length at a higher value, which means that it performs unnecessary drops. RED, ARED and REM do not show much difference, PI, AFC and DFC stabilize around a reference value, DFC performing the best and AFC is the second one, which is shown in Figure 5.74. Network utilization shown in Figure 5.75 is similar to the "30 nodes" case, PI and AFC drops the least amount of packets, and the fairness index is close to each other for each mechanism, except the index of RED and ARED, which is lower. Delay variation shown in Figure 5.77 favors PI slightly more than AFC and DFC.



a) Instant. queue length with RED

b) Instant. queue length with ARED

c) Instant. queue length with REM

d) Instant. queue length with BLUE

e) Instant. queue length with PI

f) Instant. queue length with DFC

g) Instant. queue length with AFC

Figure 5.70. Comparison of instantaneous queue lengths for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms

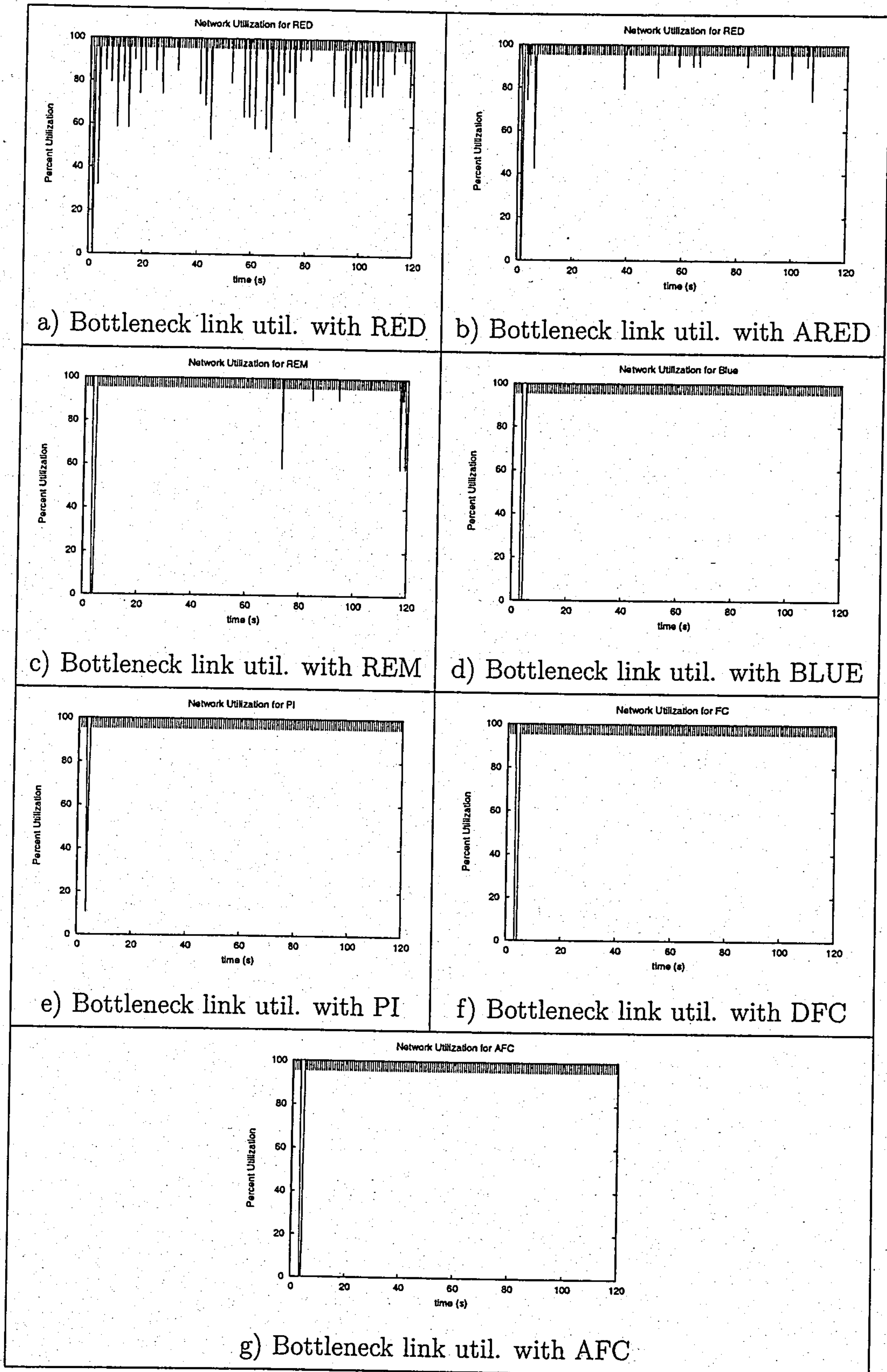


Figure 5.71. Comparison of bottleneck link utilization for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms

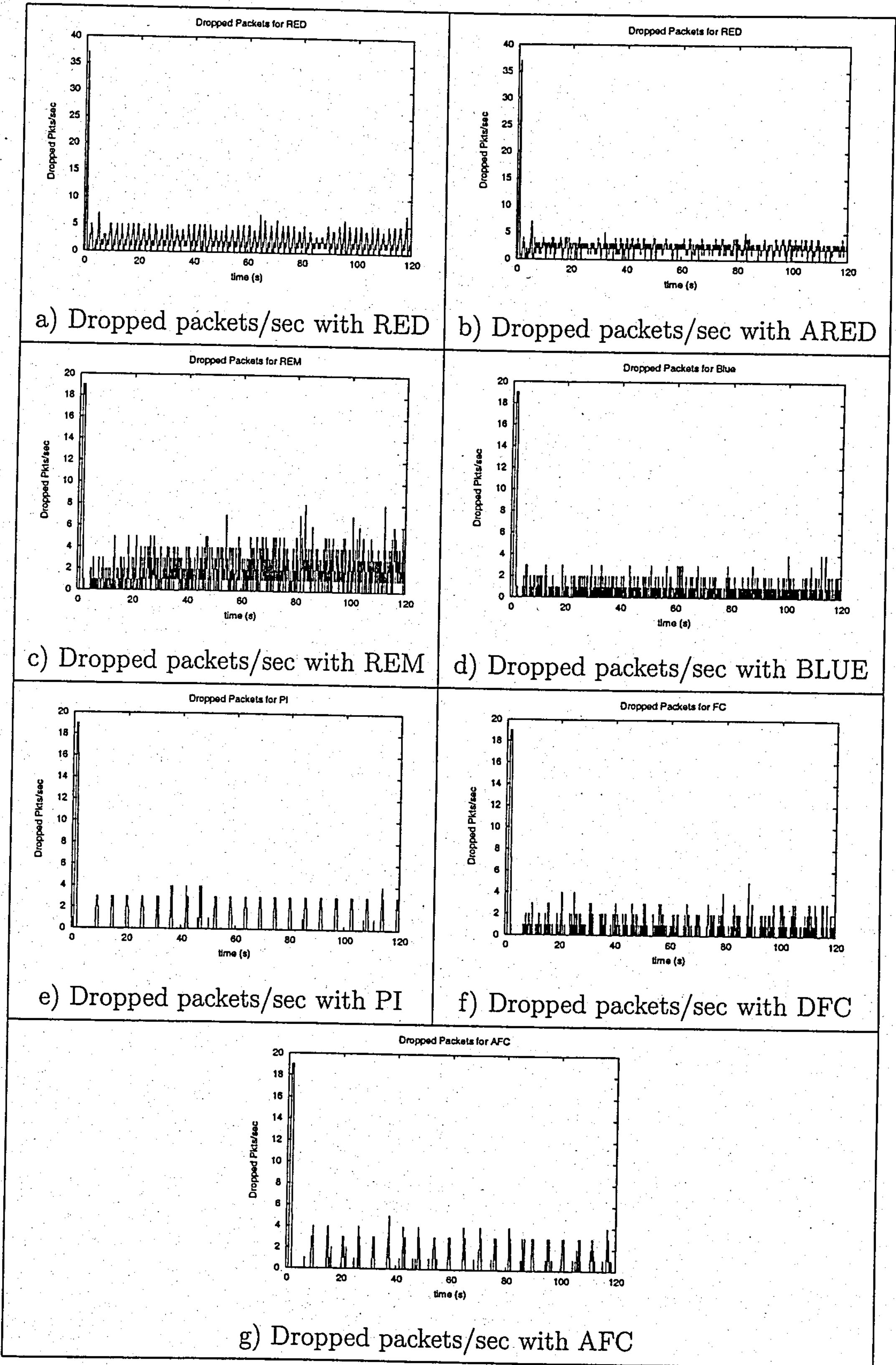


Figure 5.72. Comparison of dropped packets/sec for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms

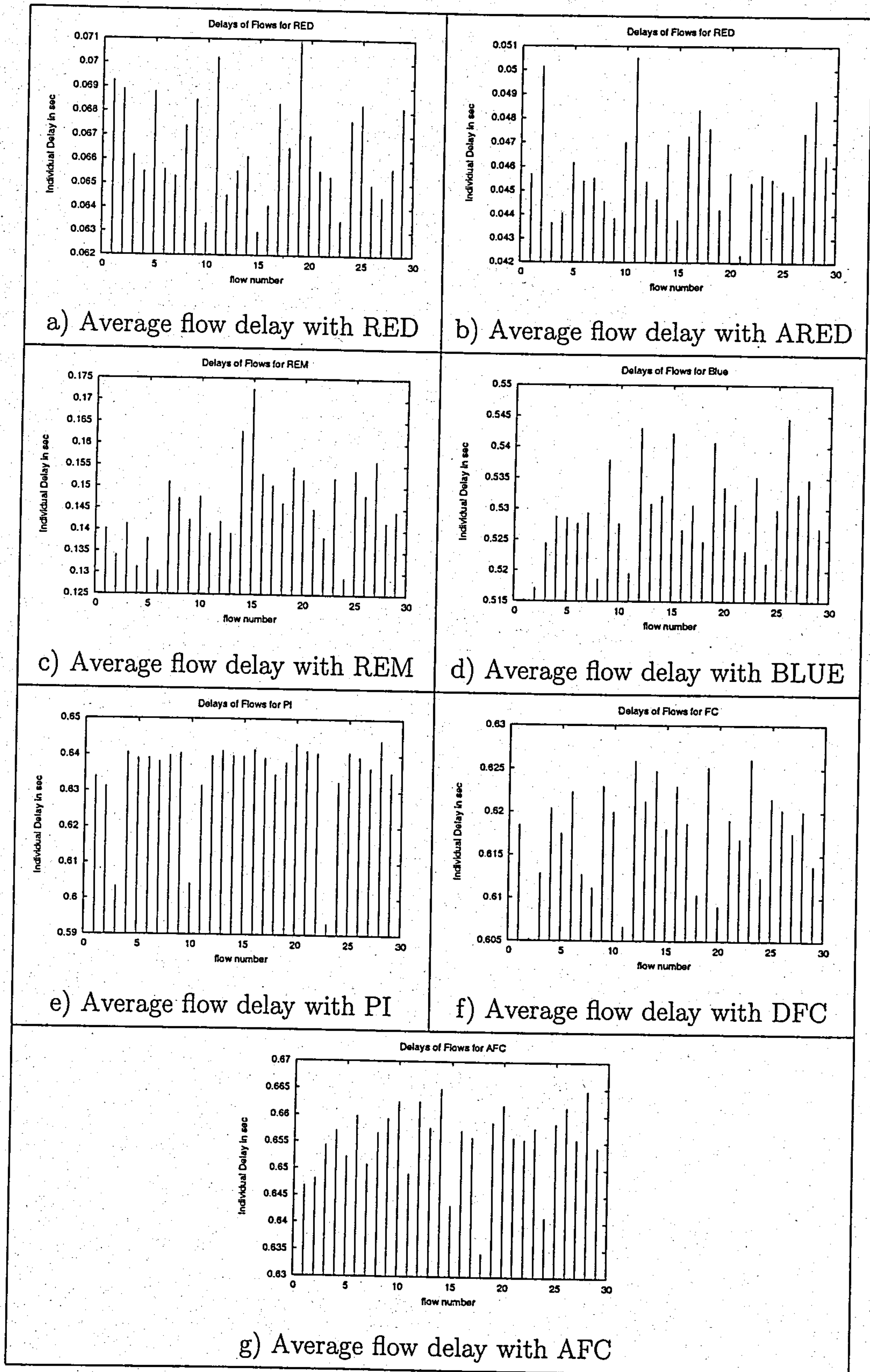


Figure 5.73. Comparison of average individual flow delays for wireless FTP traffic with $buffer = 200$, $nodes = 30$ and a bottleneck delay of 5 ms

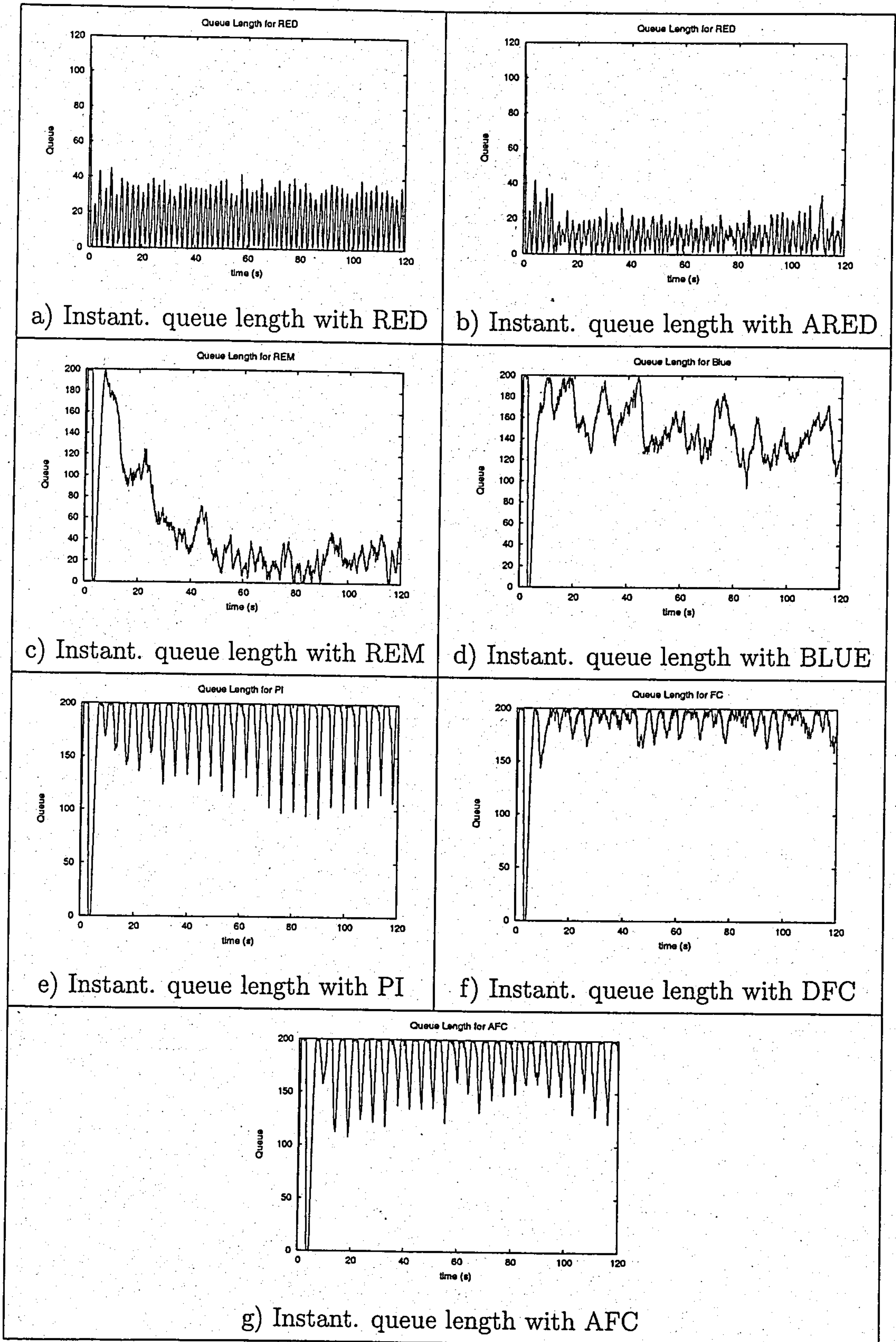


Figure 5.74. Comparison of instantaneous queue lengths for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms

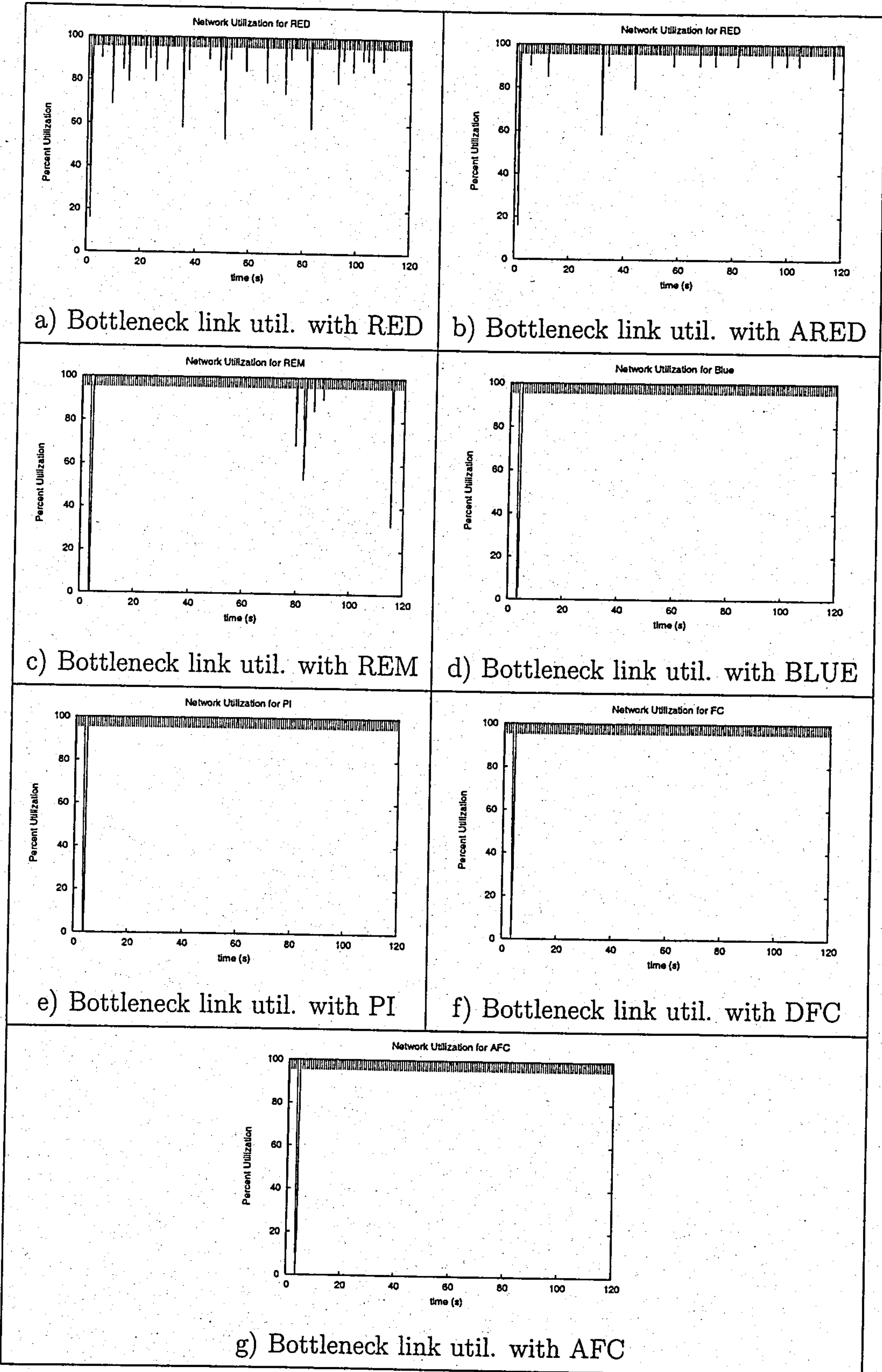


Figure 5.75. Comparison of bottleneck link utilization for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms

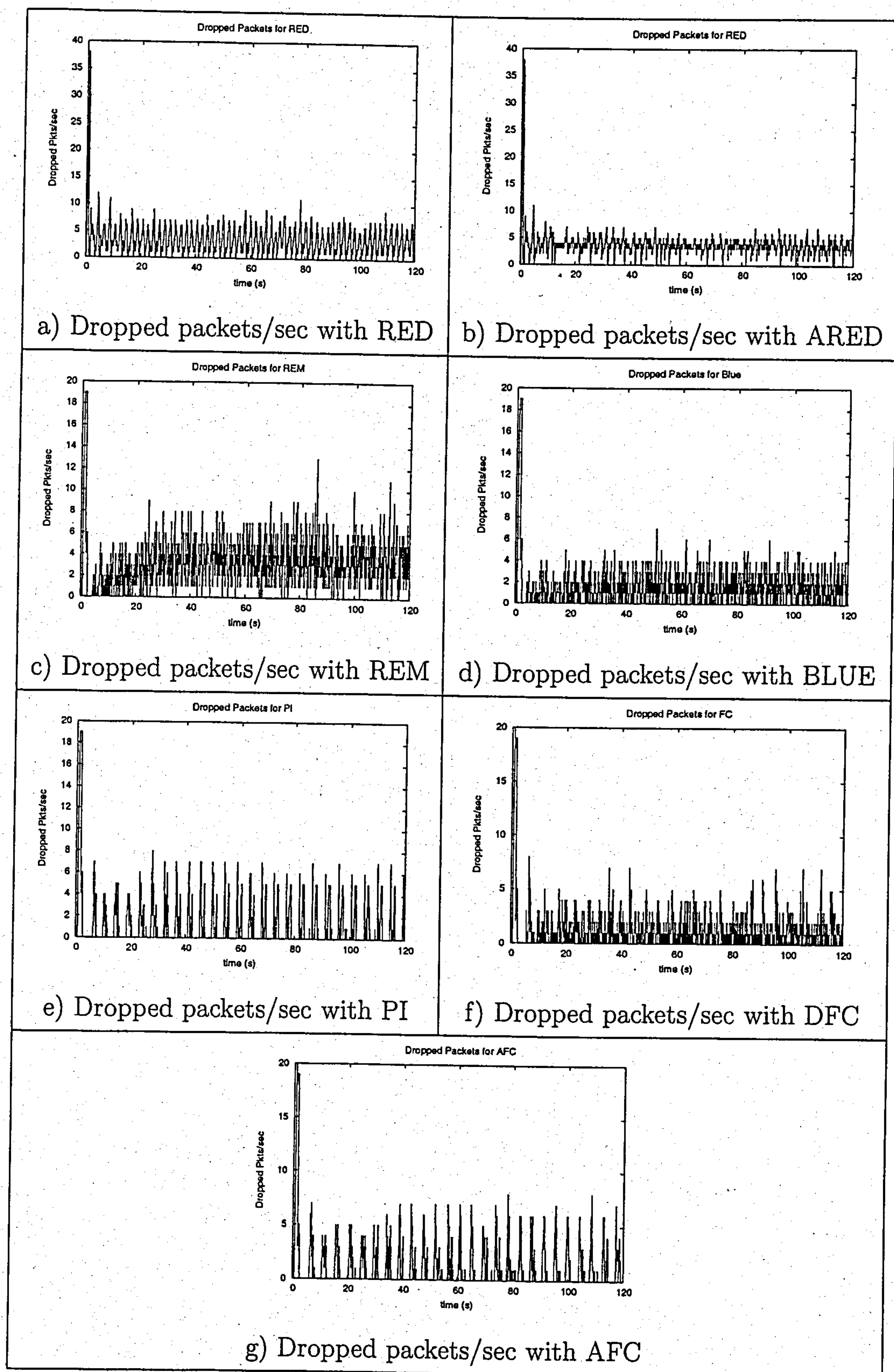
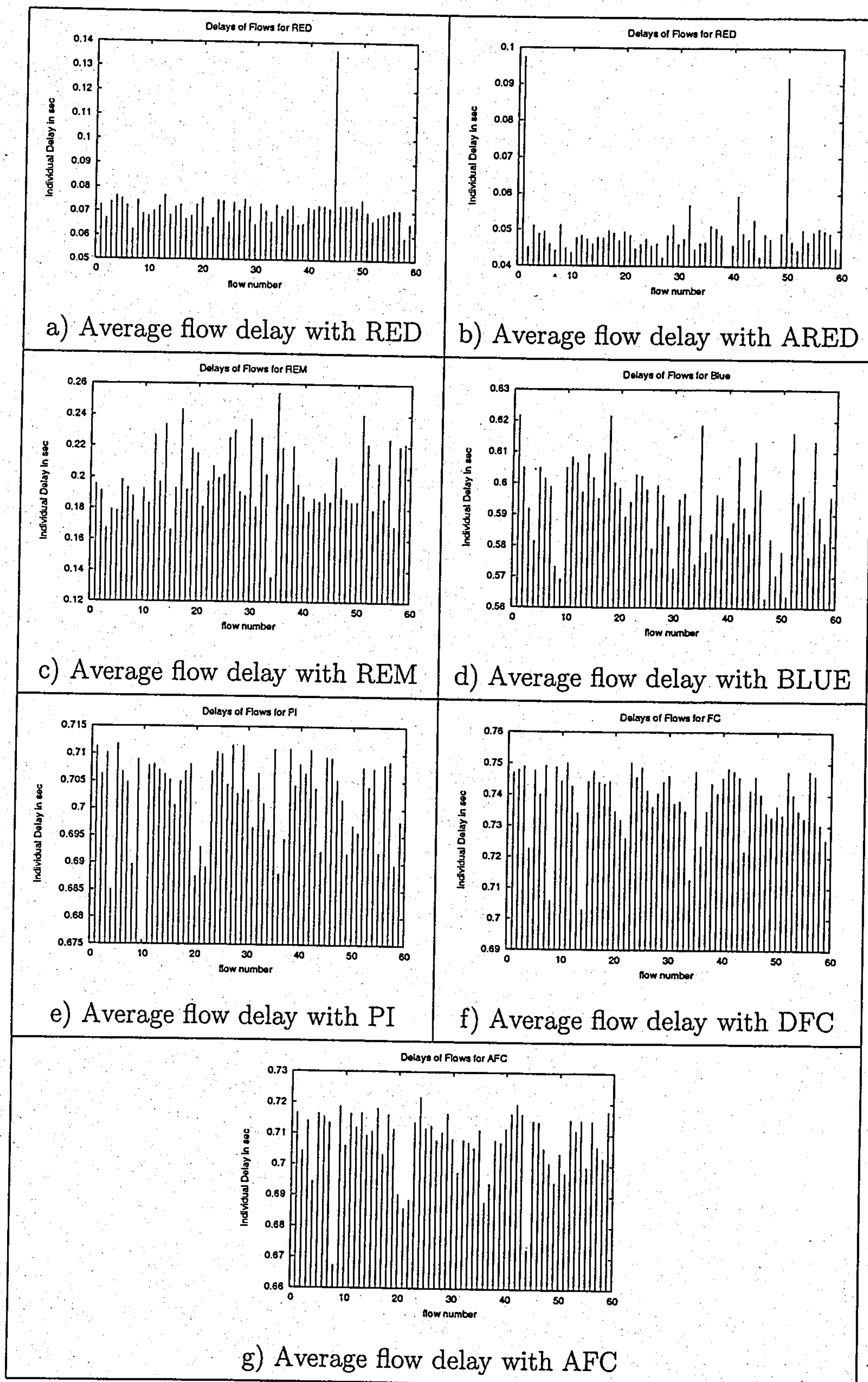


Figure 5.76. Comparison of dropped packets/sec for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms



a) Average flow delay with RED

b) Average flow delay with ARED

c) Average flow delay with REM

d) Average flow delay with BLUE

e) Average flow delay with PI

f) Average flow delay with DFC

g) Average flow delay with AFC

Figure 5.77. Comparison of average individual flow delays for wireless FTP traffic with $buffer = 200$, $nodes = 60$ and a bottleneck delay of 5 ms

6. CONCLUSIONS AND FUTURE WORK

In this thesis, we have proposed two AQM mechanisms, one is the direct fuzzy controller and the other one being the adaptive one. Both are based on the fuzzy control theory. To test the effectiveness of the proposed mechanisms, different cases of scenarios are performed, ranging from wireless ones to CBR and WEB traffic. To have a good point of view on the AQM mechanisms, the number of source nodes, bottleneck link delay and the buffer size at the congested router are also changed. Some of those cases are performed with ECN bit set to have an insight on it also. For comparisons of the proposed mechanisms, REM, RED, ARED, BLUE and PI are used. The performance metrics examined are tracking of the reference value set for the queue length, packet drops, bottleneck link utilization, delay variation and fairness. To provide a better understanding, the graphical representation of those metrics, except fairness, are provided. In fact, delay variation and fairness are coupled since an increase in the variation means that the mechanism does not behave fair, if we look at it from a traffic point of view. From the load point of view, fairness is calculated using Jain's fairness index.

From the results, it can be seen that DFC performs better than almost all algorithms in almost every case. AFC performs even better for the cases, in which the scenarios differ greatly than the ones used for parameter optimization. Both algorithms show good results under heavy or very heavy load and outperform the others mostly. The AFC algorithm is more powerful in general, since it has the ability to adapt to different conditions.

Focusing on the results of the simulations, we believe that AFC or even DFC can be used at heavily congesting routers with great success. The implementation is not difficult, since the procedure of fuzzy control is based on a table look-up procedure after initialization, which can easily be done offline. The storage for the results after initialization is not a big problem, since it will only require very few memory. The computations required after initialization are only comparisons and basic arithmetic

operations.

There is nothing such as "optimum" for the design of the fuzzy controller, since it highly depends on the heuristics and hence fuzzy rules, which can be defined entirely different by some other person. For the future work, the design parameters can be examined to obtain even better results. There is also the chance to change the table of the fuzzy rules or even add another design parameter as another input membership function. For example, the ECN bit can be added as another design parameter for future work. However adding a single bit as another design parameter is not as easy as it looks. The dynamics of the FMRLC must be carefully examined before performing such enhancement. Hence, we do not claim that those designs are optimum, however they are achieving to survive and perform well under a different variety of network conditions.

To serve well in a differentiated services environment, the designs can also include different linguistic rules and other variables. The designs of the fuzzy controllers proposed can be improved to provide the requirements of the differentiated services in the future.

REFERENCES

1. Ryu, S., C. Rump, and C. Qiao, "Advances In Internet Congestion Control", *IEEE Communications Surveys and Tutorials*, Vol. 5, No. 1, pp. 28–39, Third Quarter 2003.

2. Postel, J., "Transmission Control Protocol", *IETF*, RFC 793, September 1981.

3. Jacobson, V., "Congestion Avoidance and Control", *Proc. ACM SIGCOMM '88*, pp. 314–29, August 1988.

4. Allman, M., V. Paxson, and W. Stevens, "TCP Congestion Control", *IETF*, RFC 2581, April 1999.

5. Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery Algorithms", *IETF*, RFC 2001, January 1997.

6. Braden, B., D. Clark, J. Crowcroft, B.Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", *IETF*, RFC 2309, April 1998.

7. Hollot, C. V., V. Misra, D. Towsley, and W. B. Gong, "A Control Theoretic Analysis of RED", *Proceedings of IEEE/INFOCOM*, April 2001.

8. Christiansen, M., K. Jeffay, D. Ott, and F. D. Smith, "Tuning RED for Web Traffic", *IEEE/ACM Transactions on Networking*, Vol. 9, No. 3, pp. 249–264, June 2001.

9. Low, S. H., F. Paganini, J. Wang, S. Adlakha, and J. C. Doyle, "Dynamics of TCP/RED and a Scalable Control", *Proceedings of IEEE/INFOCOM*, June 2002.

10. Floyd, S., R. Gummadi, and S. Shenker, "Adaptive RED: An Algo-

rithm for Increasing the Robustness of REDs Active Queue Management", http://www.icir.org/floyd/papers/adaptiveRed.pdf, August 2001.

11. Floyd, S., "Recommendation on Using the Gentle Variant of RED", http://www.icir.org/floyd/red/gentle.html, March 2000.

12. Floyd, S., "Optimum Functions for Computing the Drop Probability", Email available at http://www-nrg.ee.lbl.gov/floyd/REDfunc.txt, October 1997.

13. Rosolen, V., O. Bonaventure, and G. Leduc, "A RED Discard Strategy for ATM Networks and its Performance Evaluation with TCP/IP Traffic", *Computer Communication Review*, Vol. 29, No. 3, July 1999.

14. Floyd, S., "A Report on Recent Developments in TCP Congestion Control", *IEEE Communication Magazine*, Vol. 39, No. 4, pp. 84–90, April 2001.

15. Floyd, S., "TCP and Explicit Congestion Notification", *ACM Computer Communication Review*, Vol. 24, No. 5, pp. 10–23, October 1994.

16. Paganini, F., Z. Wang, S. H. Low, and J. C. Doyle, "A new TCP/AQM for Stable Operation in Fast Networks", *Proceedings of IEEE/INFOCOM*, 2003.

17. Kalampoukas, L. and A. Varma, "Explicit Window Adaptation: A Method to Enhance TCP Performance", *Proceedings of IEEE/INFOCOM*, pp. 242–251, 1998.

18. Savoric, M., *Fuzzy Explicit Window Adaptation: A Method to Further Enhance TCP Performance*, TKN (Telecommunication Networks Group), Technical Report, TKN-03-010, Berlin, May 2003.

19. Feng, W., D. Kandlur, D. Saha, and K. Shin, "A Self-Configuring RED Gateway", *Proceedings of IEEE/INFOCOM*, pp. 1320–1328, March 1999.

20. Feng, W. C., D. Kandlur, D. Saha, and K. Shin, *BLUE A new Class of Active Queue Managament Algorithms*, University of Michigan, Department of Electrical

Engineering and Computer Science, Technical Report, CSE-TR-387-99, Computer Science and Engineering Division, Room 3402, EECS Building, Ann Arbor, Michigan 48109-2122, USA, 1999.

21. Athuraliya, S., V. H. Li, S. H. Low, and Q. Yin, "REM: Active Queue Management", *IEEE Network*, May/June 2001.

22. Misra, V., D. Towsley, and W. B. Gong, "Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED", *Proceedings of ACM/SIGCOMM 2000*, September 2000.

23. Widmer, J., R. Denda, and M. Mauve, "A Survey on TCP-Friendly Congestion Control", *IEEE Network*, May/June 2001.

24. Hollot, C. V., V. Misra, D. Towsley, and W. B. Gong, "On Designing Improved Controllers for AQM Routers Supporting TCP Flows", *Proceedings of IEEE/INFOCOM*, pp. 1726–1734, April 2001.

25. Sridharan, M., S. Chellappan, A. Durresi, H. Ozbay, and R. Jain, "Tuning RED Parameters in Satellite Networks Using Control Theory", Submitted to ICC 2003, 2003.

26. Balakrishnan, H., V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements", *Proceedings of IEEE/INFOCOM*, March 1998.

27. Padhye, J., V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Reno Performance: A Simple Model and its Empirical Validation", *IEEE/ACM Transactions on Networking*, Vol. 8, No. 2, pp. 133–145, April 2000.

28. Hollot, C. V., V. Misra, D. Towsley, and W. B. Gong, "Analysis and Design of Controllers for AQM Routers Supporting TCP Flows", *IEEE Transactions on Automatic Control*, Vol. 47, No. 6, pp. 945–959, June 2002.

29. Passino, K. M. and S. Yurkovich, *Fuzzy Control*, Addison Wesley Longman, Inc., 1997.

30. Chrysostomou, C., A. Pitsillides, G. Hadjipollas, Y. A. Sekercioglu, and M. Polycarpou, "Fuzzy Logic Congestion Control in TCP/IP Best Effort Networks", *In Proceedings of the Australian Telecommunications, Networks and Applications Conference (ATNAC'03)*, Melbourne, Australia, December 2003.

31. "NS: Network Simulator", http://www.isi.edu/nsnam/ns/index.html, 2004.

32. Chiu, D. M. and R. Jain, "Analysis of the Increase and Decrease Algorithms for Conggestion Avoidance in Computer Networks", *Computer Networks and ISDN Systems*, Vol. 17, 1989.