BLUE: Active Queue Management

CS756 Project Report

Sunitha Burri
May 5, 2004

INDEX

BLUE: Active Queue Management

Abstract

In order to avoid high packet loss rates many Active queue management algorithms have been proposed, which drop the packets before the queues overflow at the gateways, so that the end nodes can respond to the congestion before queues overflow. BLUE is one such active queue management algorithm, which uses packet loss and link idle events to manage congestion. In this report I'll discuss the working of BLUE and using simulation results, I'll present that BLUE performs significantly better than RED [2], another active queue management technique, which uses moving average of queue length to manage congestion.

1. Introduction

In the past, rather the decentralized and fast-changing evolution of Internet architecture has worked reasonably well. But, nowadays, the network traffic is increasing exponentially due to the integration of enormous networks with many different service providers, users and protocols. Under this situation, if the Internet is not designed carefully it will be more vulnerable to frequent congestion and collapse. Congestion occurs when the total demand exceeds the bandwidth that the available resources can provide.

The earliest congestion control mechanism that was proposed in the mid 80's is TCP end-to-end congestion control. This works as follows: Source adjusts its transmission rate according to the level of congestion it perceives through the packet loss events. It has 3 drawbacks: 1. By the time source perceives the packet loss, considerable amount of time might have elapsed from the time when the packet has been dropped at a gateway due to buffer over flows, during this time source might have dumped some more packets to the network which may get dropped based on the level of congestion. 2. Resources are wasted as the dropped packets consume resources on their way from source to destination. 3. Upon perceiving a packet loss event, TCP sources may back off at the same time leading to under utilization of the link.

In light of these observations, researchers began to consider congestion control at gateways. The mechanisms that were considered for congestion control at gateways are:
1. Queue management algorithms: These manage the length of the queues by dropping packets whenever necessary and appropriate.
2. Scheduling algorithms: These determine the next packet to be sent and allocate the bandwidth among different flows.

Traditional Queue management algorithms drop the packet only when the queues are full. Some of those techniques are:
1. Tail Drop: This technique drops the incoming packet if the queue is full. It suffers from the following two drawbacks.
   a. Lock out: In some situations tail drop allows a single connection or a few flows to monopolize queue space, preventing other connections from getting room in the queue.
   b. Full Queues: This allows queues to maintain a full or almost full status for long periods of time, as it signals congestion through a packet drop only when the queue has become full. The full or almost full status will result in high end-to-end delays. As queues are always full the packets that arrive in burst from different flows will get dropped, as a result of this all of those flows may back off at the same time leading to under utilization of the links.
2. Random Drop on full: This technique drops a randomly selected packet if the incoming packet sees the queue as full. As the randomly selected packet may be from a different flow when compared to the arriving packet, it does not suffer from lock out problem. But suffers from full queues problem.
3. Drop front on full: This technique drops the head of the queue if the incoming packet sees the queue as full. This discipline also does not suffer from Lock out problem, but suffers from full queues problem.

Full queues problem can be solved if the packets are dropped well before the queues become full, so that end nodes can respond to congestion before buffers overflow.

This technique is known as Active queue management. Goals of active queue management are: reducing number of packet losses; low end-to-end delays; and avoiding Lock out problem. As of now, many of the Active queue management algorithms have been proposed. In this project I will be focusing mainly on BLUE and RED algorithms.

RED (Random Early Detection) Algorithm was first proposed by S.Floyd and V. Jacobson [2]. This discipline maintains a moving average of the queue length to manage congestion. If this moving average of the queue length lies between a minimum threshold value and a maximum threshold value, then the packet is either marked or dropped with a probability. If the moving average of the queue length is greater than or equal to the maximum threshold then the packet is dropped. Even though, it tries to avoid global synchronization and has the ability to accommodate transient bursts, in order to be efficient RED must have sufficient buffer spaces and must be correctly parameterized. In contrast to RED, BLUE algorithm uses packet loss and link utilization to manage congestion. This algorithm is proposed by Wu-chang Feng, Kang Shin, Dilip Kandlur, Debanjan Saha,

The remainder of this report is organized as follows. Section 2 presents the BLUE algorithm. Section 3 provides Simulation model, results, and evaluates the performance of BLUE by analyzing the simulation results. Section 4 presents conclusions and Section 5 presents future work.

2. BLUE Active Queue Management Algorithm

BLUE performs queue management based on packet loss and link utilization. It maintains a marking probability p_m to either mark or drop the packets. If the queue is continually dropping the packets , p_m is incremented by a factor δ_1. If the queue is empty or link is idle, p_m is decremented by a factor δ_2. The value of δ_1 must be set significantly larger than δ_2. This is because link is underutilized when the congestion management is either too aggressive or too conservative, but packet loss occurs only when the congestion mechanism is too conservative. BLUE uses one more parameter *freeze_time*, which determines the time interval between two successive updates of *freeze_time*. It allows the

changes in the marking probability to take effect before the value is updated again. BLUE algorithm is given below.

The BLUE algorithm:

Upon Packet loss (or $Q_{len} > L$) event:

if ((*now* – *last_update*) > *freeze_time*)

$p_m := p_m + \delta 1$

last_update := *now*

Upon link idle event:

if ((*now* – *last_update*) > *freeze_time*)

$p_m := p_m - \delta 2$

last_update := *now*

Marking probability, p_m, is also updated when the queue length exceeds a certain value in order to allow room to be left for transient bursts and to control the queueing delay when the size of the buffer being used is large.

3. Performance evaluation of BLUE through simulation model

3.1 Simulation Design

Network simulator ns2 [4] is used to run the simulations in order to evaluate the performance of BLUE. A new queue object (blue) is added to ns2. The network used in simulations is given in Fig.1. This network consists of a bottleneck link between nodes A and B, which has a capacity of 35Mbs and delay of 10ms. The nodes that are on the left side of the node A (N0, N1, N2, N3) are the nodes to which the TCP agents will be attached. Each of these nodes is connected to the node A through a duplex-link of capacity 25Mbs with a delay of 10ms. The nodes that are on the right side of node B (N4, N5, N6, N7) are the nodes to which TCP sink agents will be attached. Pareto on/off sources with mean on-times of 200ms and mean off-times of 200ms, which are attached to the left side nodes send packets of size 1000 Bytes to the sinks on corresponding nodes on the right side.

Actual Queue lengths and marking probability are used in order see whether BLUE and RED are achieving the goals (avoiding global synchronization and biases against bursty sources) or not. Packet loss rates and Link utilization are used as performance metrics in

order to compare the performance of BLUE with that of RED. For RED queue *minth* and *maxth* are set at 20% and 80% of the queue size with *maxp* of 1. RED queue configurations and BLUE queue configurations that are used during the simulations are given in Table1 and Table2 respectively.

3.2 Analyzing Simulation Results

3.2.1. Actual Queue Lengths and Marking Probability plots without ECN timeouts

Simulations are run on the simple network described in the previous section for 60 seconds, Initially starting with 200 Pareto on/off sources. The number of sources are increased by 200 every 20 seconds. B4 configuration of BLUE and R2 configuration of RED are used and queue between nodes A and B is monitored in order to plot the actual queue lengths and marking probabilities.

Figure 2 shows that RED suffers from continual packet losses during the entire simulation and also at lower loads periods of packet losses or followed by periods of under utilization of the link due to the deterministic packet marking behavior. Where as from Figure 3, we can say that actual queue length plot of BLUE is stable. Only at heavy load BLUE suffers from continual packet losses. Even though BLUE drops every packet in that situation, as the TCP sources do not invoke retransmission timeouts when ECN signal is received with a congestion window of 1, TCP sources are aggressive enough to maintain full queues.

Figure 4 shows that the marking behavior of RED fluctuates a lot. Where as Figure 5, shows that marking behavior of BLUE is in accordance with the level of load.

We know that RED tries to avoid global synchronization through randomized marking as well as maintaining spacing evenly between these markings. But, Incase of large number of connections, TCP aggregate load changes rapidly leading to considerable fluctuations of marking probability over short periods of time (as shown in Figure 4). So, in that scenario RED fails to avoid global synchronization and thus leading to under utilization of link also. In contrast to RED, as BLUE marking is done randomly and

evenly over time (as shown in Figure 5), BLUE performs better in avoiding global synchronization. By limiting the queue occupancy RED allows transient bursts. From the Figure 3, we can say that there is always room for bursts as the actual queue length is less than the queue length.

From the Figure 3, Even though it may seem like queueing delay can be reduced if we can make the queue length to attain stability at a lower length, it is not possible, as the TCP sources (which are implemented without ECN timeouts), are so aggressive that actual queue lengths will be much higher.

3.2.2. Actual Queue Lengths and Marking Probability plots with ECN timeouts

Actual queue length and marking probability plots are drawn by running the simulations as described in the previous section with TCP sources that implement ECN timeouts. From the Figures 6 and 7, we can say that RED sustains continual packet losses as well as the periods of packet losses are followed by periods of under utilization of link. By observing the Figure 9, we can say that BLUE marking probability is changed in accordance with the level of congestion. By careful observation of Figure 8, we can say that link can be utilized more as well as the queuing delays can be reduced, if we can make graph to form a very low band with lower range more than 0 and higher range far less than buffer size. During the simulation I had used a fixed *freeze_time* value of 100ms. But as the number of sources increase, the RTT of all the flows increases. We can achieve the above kind of plot if we can assign effective RTT of all the connections, that are multiplexed in to the link, to *freeze_time* dynamically and dynamically change $\delta 1$ and $\delta 2$ so that the queue quickly adapts to the offered load.

3.2.3. Packet Loss Rates and Link Utilization

Simulations are run for all the 4 configurations of RED and BLUE whose parameters are given in Table1 and Table2 respectively, changing the buffer size from 100KB to 1000KB. Percent packet loss rates and percent link utilizations are plotted against buffer size.

Figure 10 and 11 are percent packet loss rates for 1500 sources and 4000 sources respectively. Figure 10 shows that all the 4 configurations of BLUE have almost 0 percent packet loss for 1500 sources. From both the figures 10 and 11 we can say that BLUE has less packet loss rates when compared to RED even under low buffer sizes. As the buffer size decreases RED suffers more packet losses.

From the Figures 12 and 13, we can observe that BLUE performs better in utilizing the link when compared to RED.

4. Conclusions

BLUE performs better when compared to RED even in cases of low buffer sizes. Even though BLUE algorithm is very simple to implement, it must also be correctly parameterized and the parameters have to be changed dynamically so as to make the queue to adapt to the dynamically changing loads. BLUE is not useful in case of non-responsive flows.

5. Future Work

Evaluating the performance of BLUE, by changing the way the marking probability is getting updated, such as modifying it multiplicatively, or both multiplicatively and additively.

References

[1] Wu-chang Feng, Kang Shin, Dilip Kandlur, Debanjan Saha, "The Blue Active Queue Management Algorithms", *IEEE/ACM Transactions on Networking*, Vol. 10, No. 4, August 2002.

[2] [RED] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, Vol. 1, pp 397-413, August 1993

[3] [RFC2309] B. Braden, 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, L. Zhang, Recommendations on Queue Management and Congestion Avoidance in the internet, April 1998.

[4] K. Fall and K. Varadhan, ns notes and documentation, The VINT project, UC Berkeley, LBL, USC/ISI and Xerox PARC (February 2000), http://www.isi.edu/nsnam/ns

[5] [Jaco88] V. Jacobson, "Congestion avoidance and control", in *Proceedings of SIGCOMM'88*, (Stanford, CA), pp. 314-329, August 1988.

Appendix A: Network Topology and Configurations

Simulation Topology

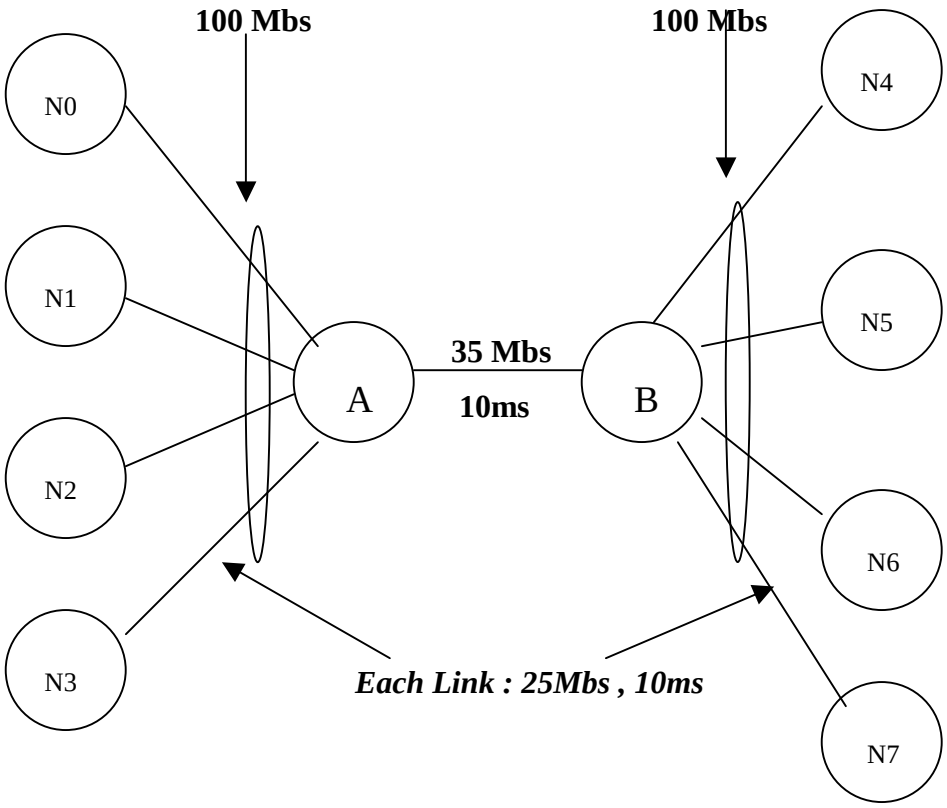


Figure 1

RED Configurations

Table 1

Configuration	wq
R1	0.0002
R2	0.002
R3	0.02
R4	0.2

BLUE Configurations

Table 2

Configuration	*Freeze_time*	δ_1	δ_1
B1	10ms	0.0025	0.00025
B2	100ms	0.0025	0.00025
B3	10ms	0.02	0.002
B4	100ms	0.02	0.002

Appendix B: Simulation Results

Figure 2: Actual Queue length (in KB) versus Time (in seconds) For RED , TCP sources without ECN timeouts

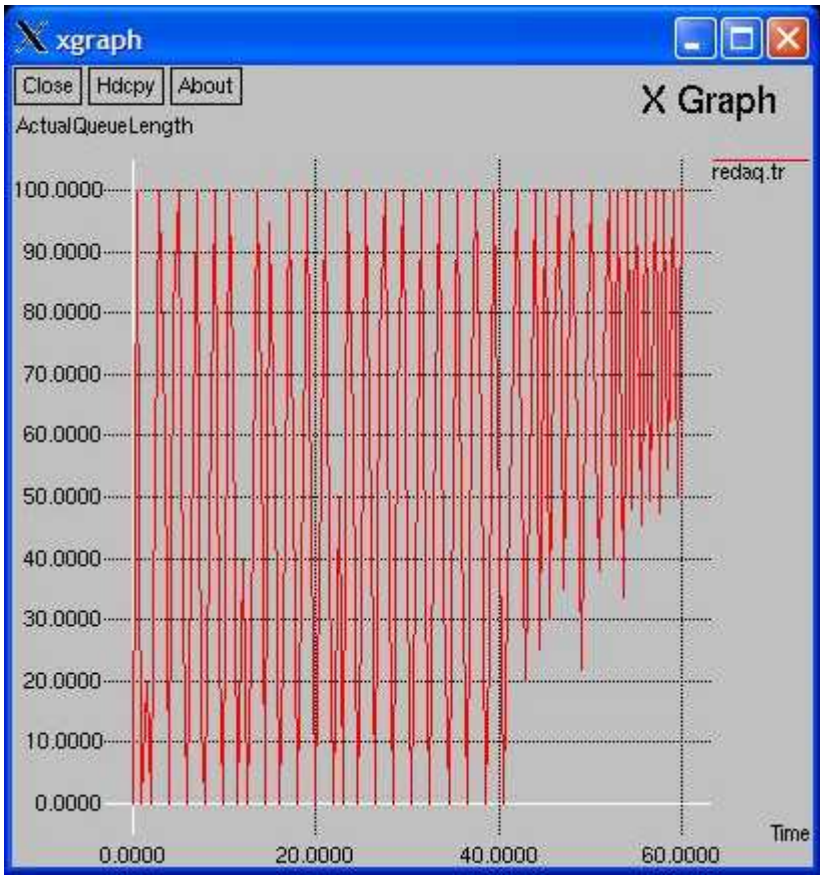


Figure 3: Actual Queue length (in KB) versus Time (in seconds) For BLUE ,TCP sources without ECN timeouts

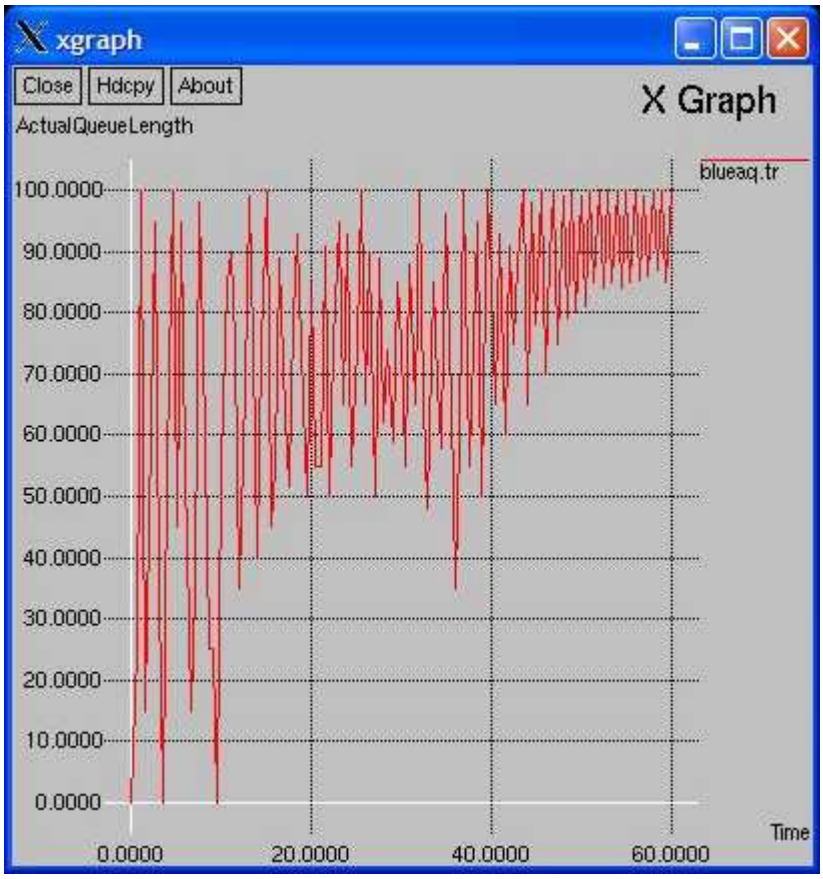


Figure 4: Marking Probability versus Time (in seconds) For RED , TCP sources without ECN timeouts

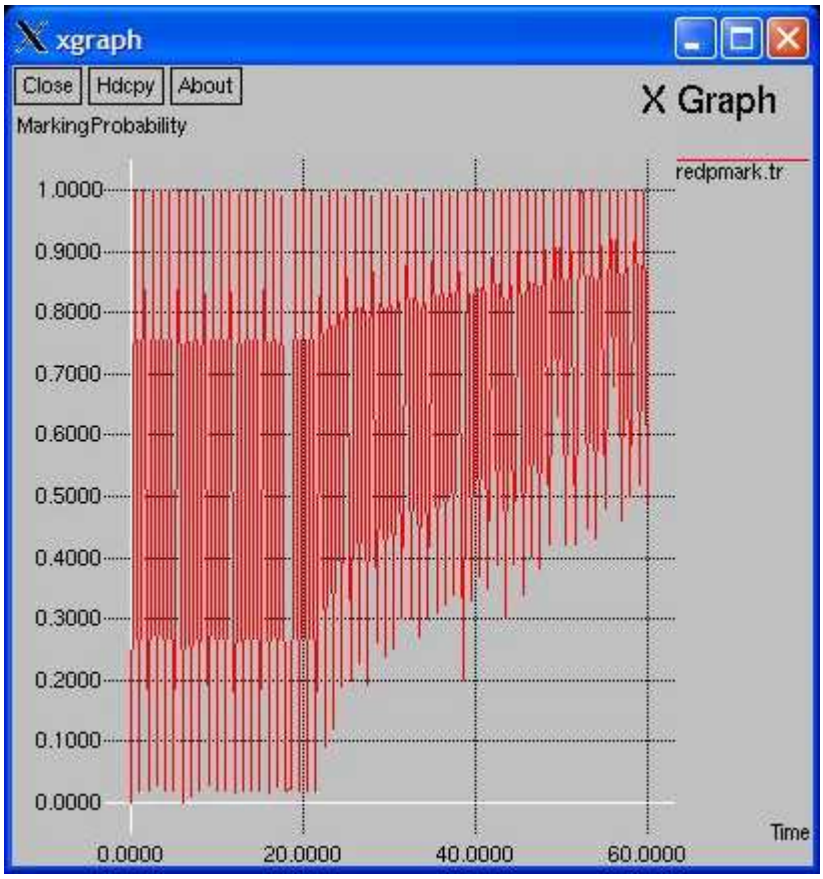


Figure 5: Marking Probability versus Time (in seconds) For BLUE , TCP sources without ECN timeouts

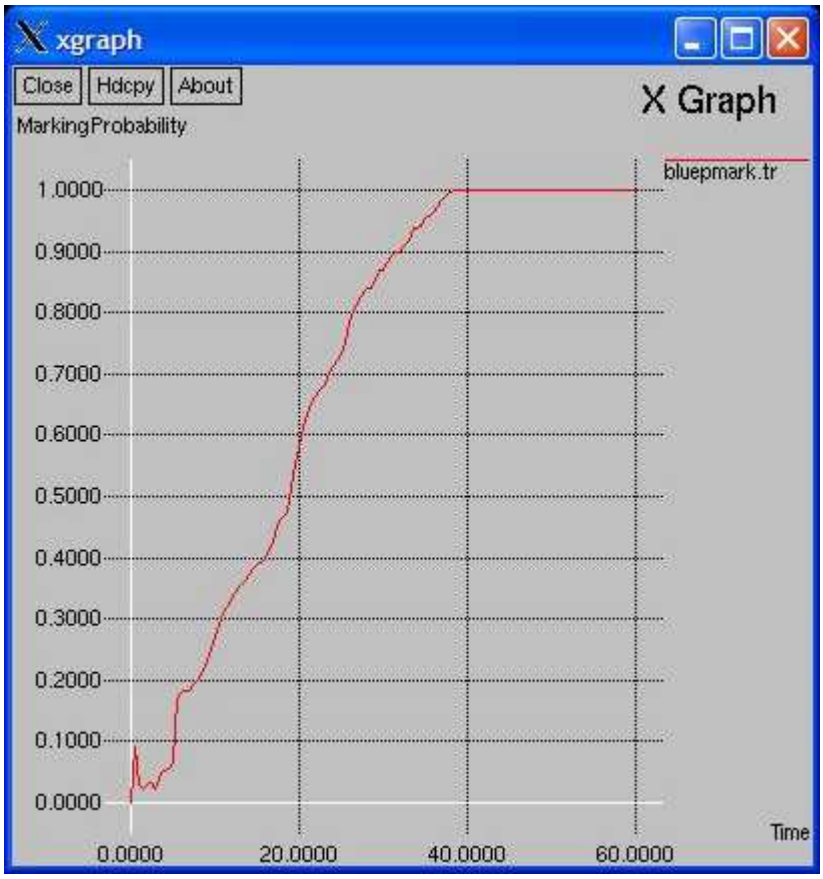


Figure 6: Actual Queue length (in KB) versus Time (in seconds) For RED , TCP sources with ECN timeouts

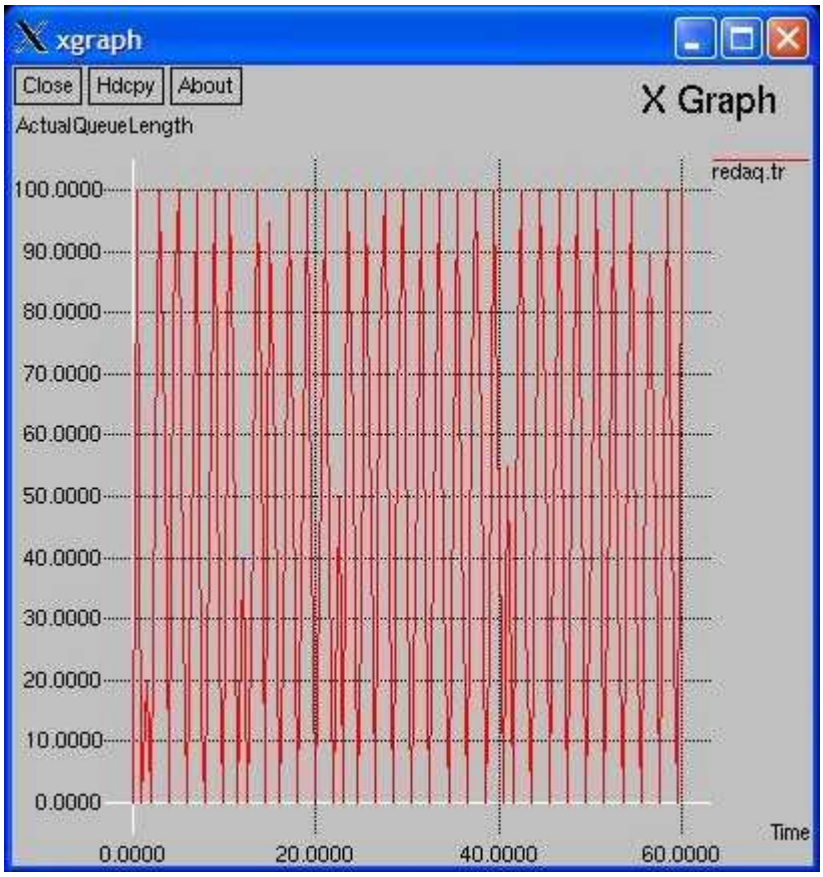


Figure 7: Actual Queue length (in KB) versus Time (in seconds) For BLUE , TCP sources with ECN timeouts

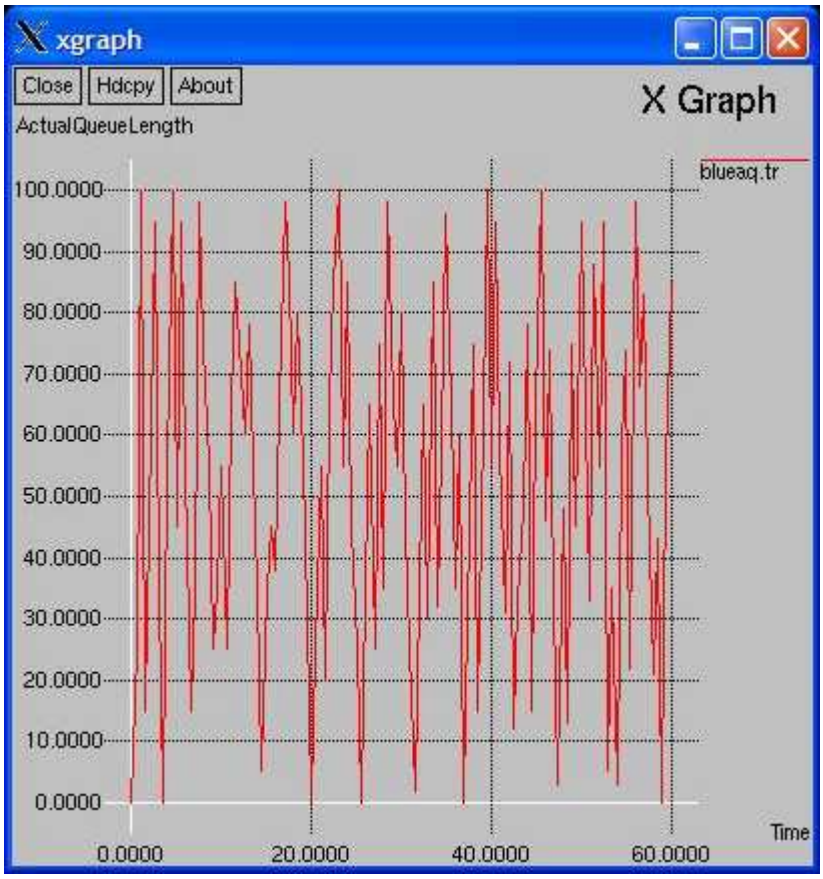


Figure 8: Marking Probability versus Time (in seconds) For RED
, TCP sources with ECN timeouts

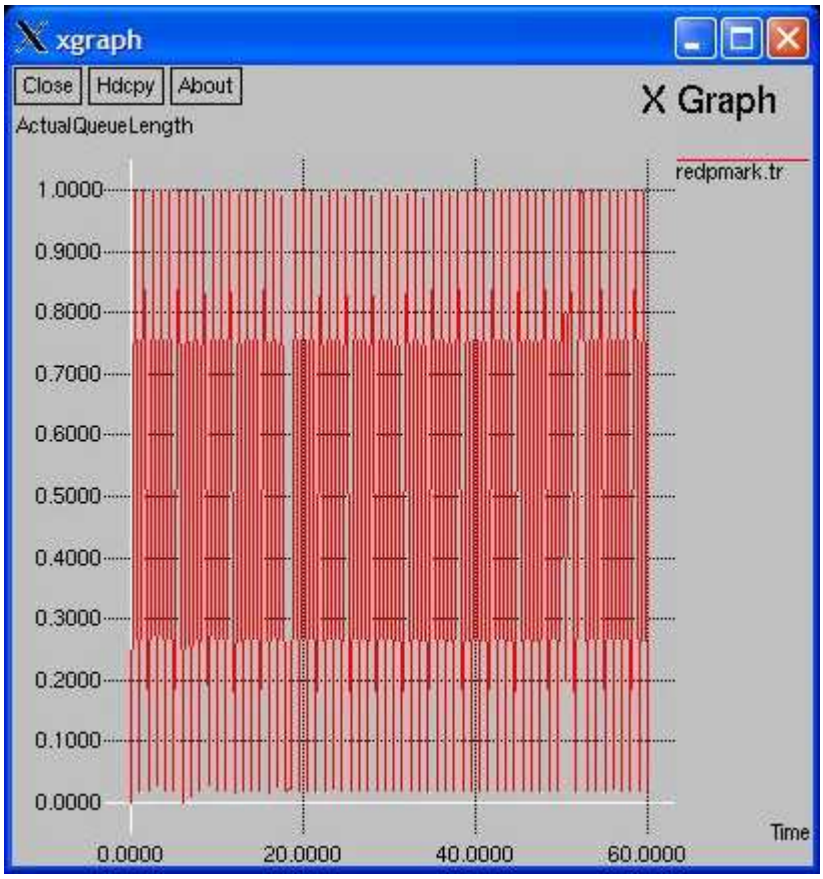


Figure 9: Marking Probability versus Time (in seconds) For BLUE , TCP sources with ECN timeouts

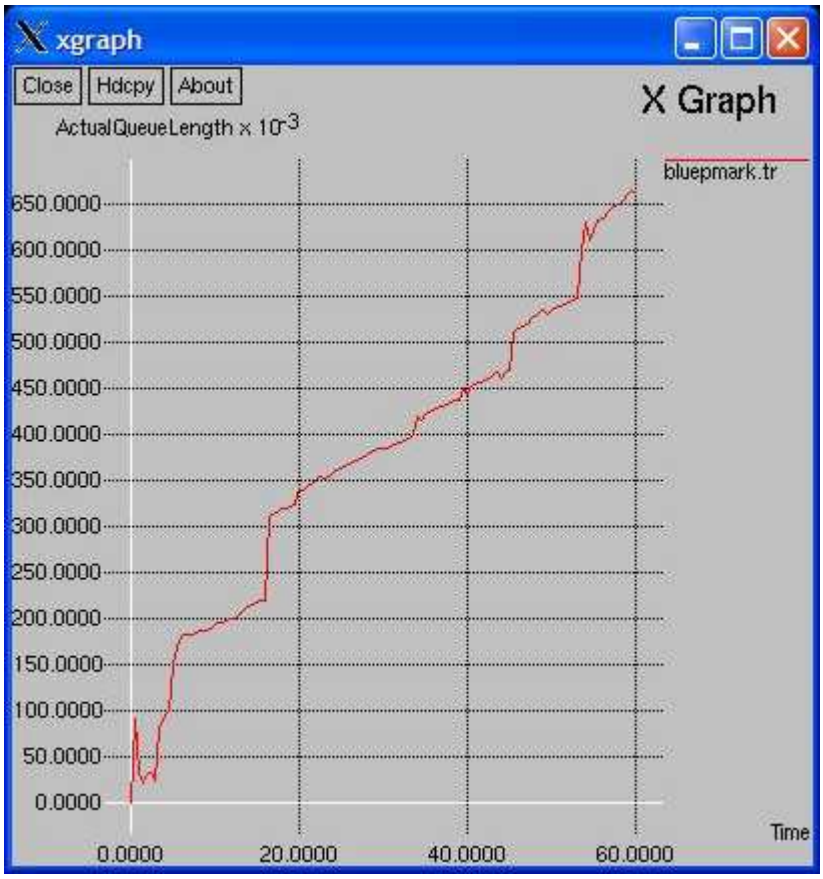


Figure 10: Percent Packet loss versus Buffer Size (in KB) for 1500 sources

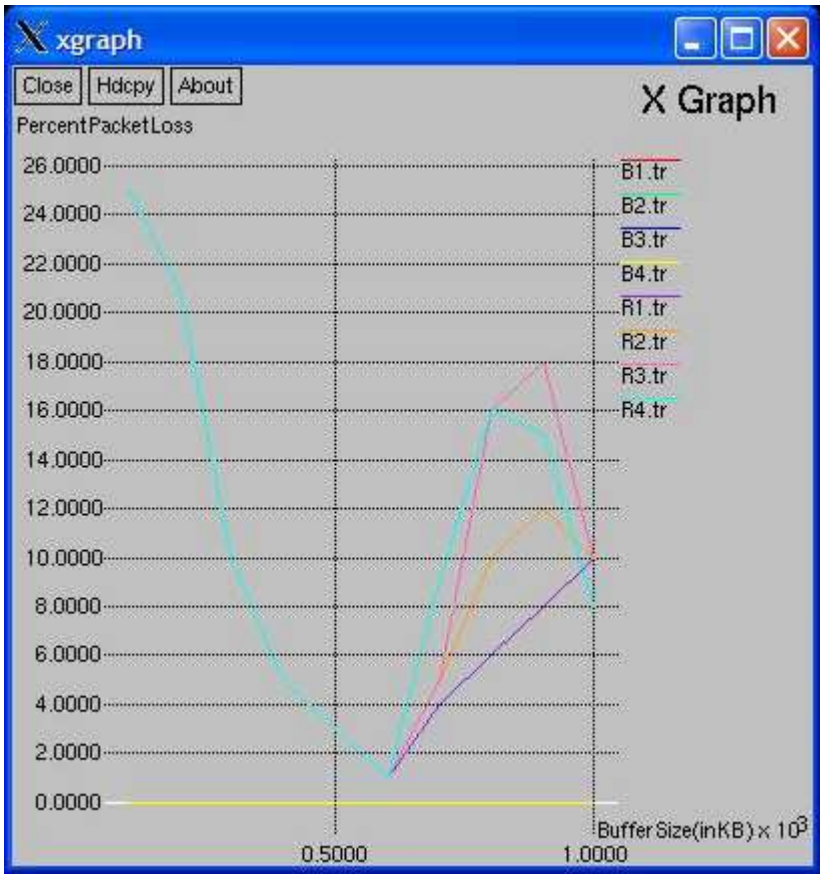


Figure 11: Percent Packet loss versus Buffer Size (in KB) for 4000 sources

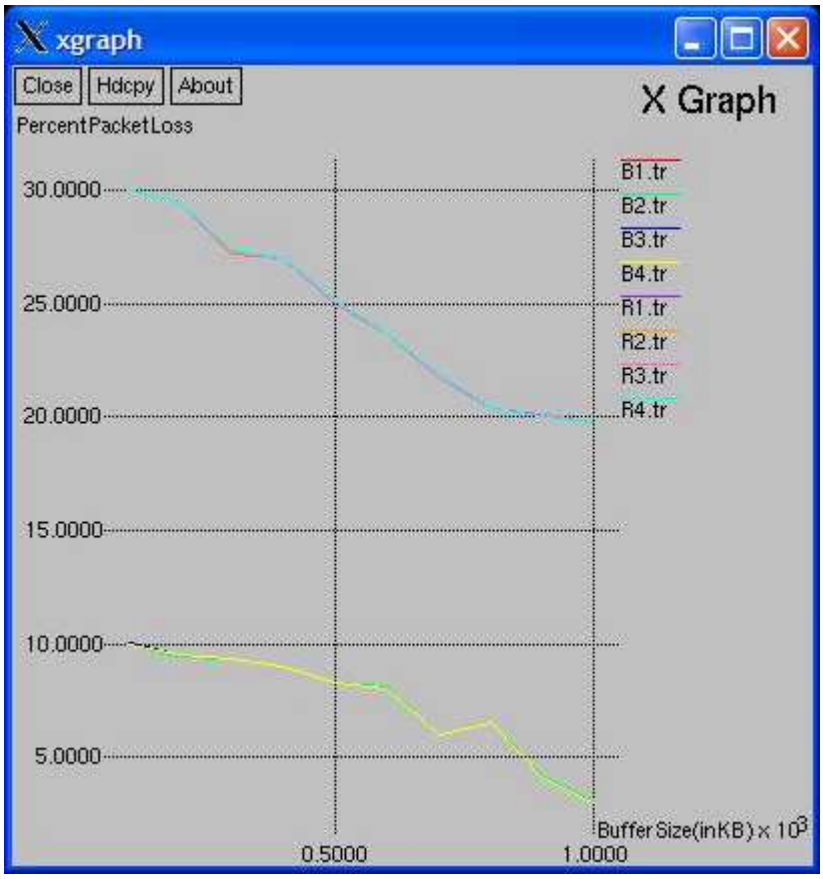


Figure 12: Percent Link Utilization versus Buffer Size (in KB) for 1500 sources

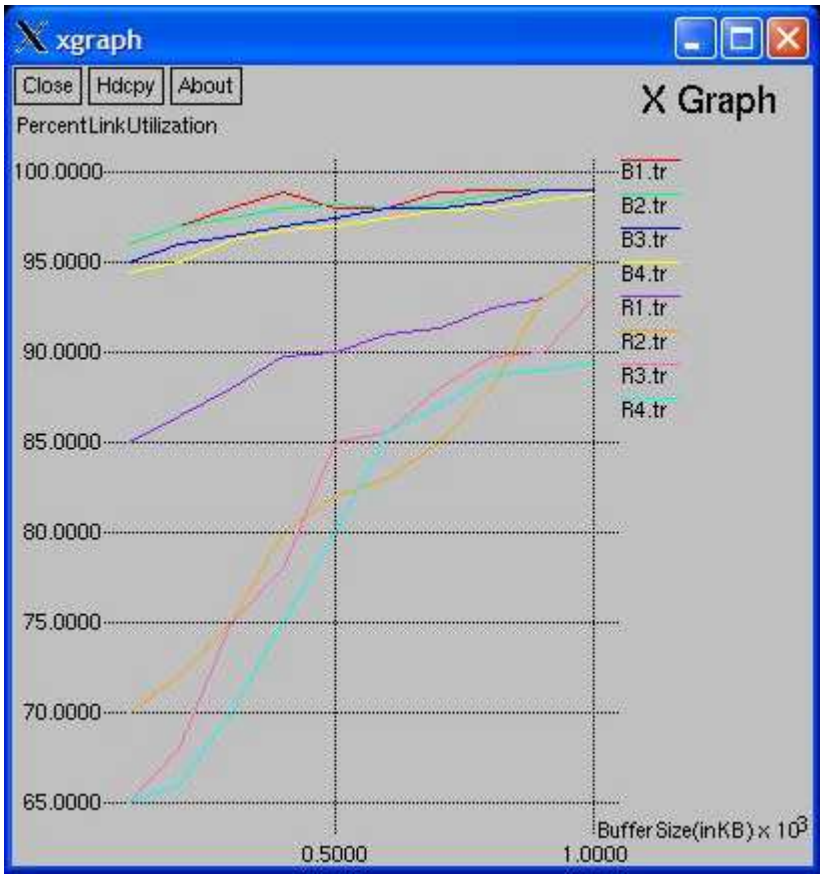
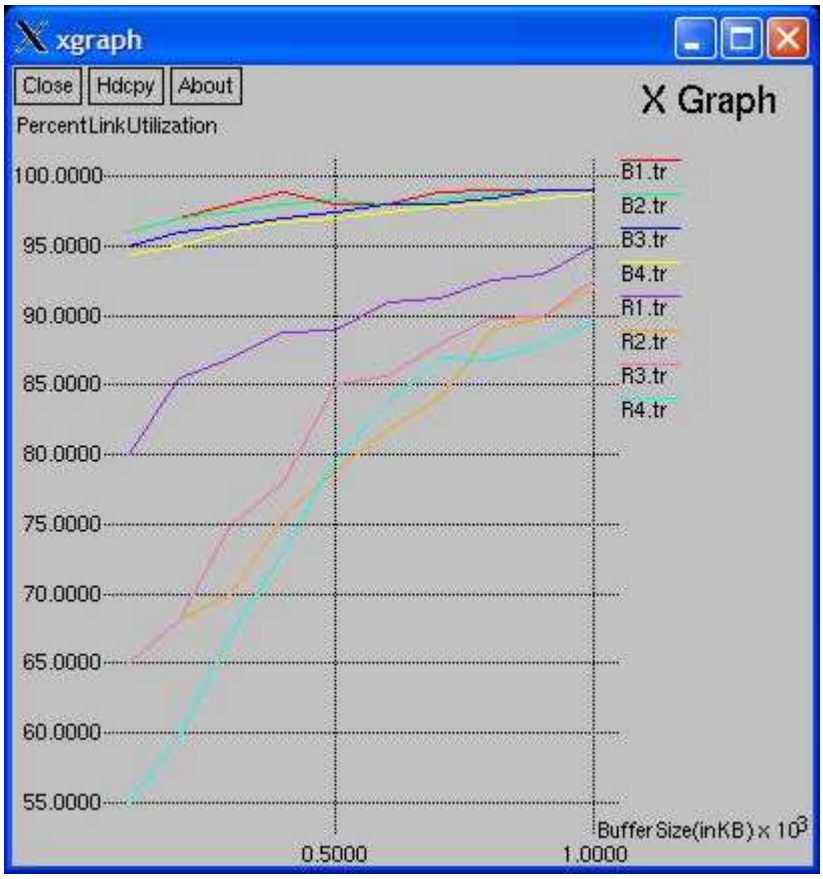


Figure 13: Percent Link Utilization versus Buffer Size (in KB) for 4000 sources



Appendix C: C++ code

“blue.h “

```
#ifndef NS_BLUE_H
#define NS_BLUE_H

#include <string.h>
#include "queue.h"
#include "config.h"
#include "trace.h"


class Blue : public Queue {
  public:
      Blue();
      ~Blue();

  protected:
      int command(int argc, const char*const* argv);
      void enque(Packet*);
      Packet* deque();
      PacketQueue *q_;  /*  FIFO queue */


      void reset();
      void inc_marking_prob();  /*  incrementing the marking
probability */
      void dec_marking_prob();  /*  decrementing the marking
probability */

      double marking_prob_;     /* marking probability */

      double inc_factor_;       /* factor by which marking probability
will be incremented */
      double dec_factor_;       /* factor by which marking probability
will be decremented */
      double last_update_time_; /* time at which marking probability is
updated last time */
      double bandwidth_;        /* bandwidth of the link */
      int setECNbit_;           /* ECN enabled */
      int idle_;            /* whether the link is idle */
      double idletime_;         /* time at which the link is idle last
time */
      double freezetime_;       /* time period between successive
updates of marking probability */

      double ptc_;             /* The ptc is the max number of (avg
sized)pkts per second which can be placed on the link */
      int mean_pktsize_;
      int drop_front_;        /* bool for dropping packet from front
when queue overflows */
      int qib_;         /* queue measured in bytes? */
```

```
      double blue_l_;         /* For allowing transient bursts and
                              controlling queing delay */

      Tcl_Channel tchan_;     /* place to write trace records */
      TracedInt curq_;  /* current qlen seen by arrivals */
      TracedDouble marking_prob_trace_;  /*  for trace purposes */
      void trace(TracedVar*); /* routine to write trace records */
};

#endif
```

“blue.cc “

```
#include <math.h>
#include <stdlib.h>
#include <sys/types.h>
#include "random.h"
#include "flags.h"
#include "delay.h"
#include "blue.h"

/* defining linkage between C++ and Tcl */
static class BlueClass : public TclClass {

 public:
      BlueClass() : TclClass("Queue/Blue") {}
      TclObject* create(int argc, const char*const* argv)
      {     return (new Blue); }
} class_blue;


Blue::Blue() {
      q_ = new PacketQueue();
      pq_ = q_;
      /* binding the variables */
      bind("marking_prob_", &marking_prob_);
      bind("inc_factor_", &inc_factor_);
      bind("dec_factor_", &dec_factor_);
      bind_bool("setECNbit_", &setECNbit_);
      bind_time("freezetime_", &freezetime_);
      bind("pktsize_", &mean_pktsize_);
      bind_bool("queue_in_bytes_", &qib_);
      bind_bool("drop_front_", &drop_front_);
      bind("blue_l_", &blue_l_);

      bind("curq_", &curq_);
      bind("marking_prob_trace_", &marking_prob_trace_);

}

Blue::~Blue() {
            delete q_;
}

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

```
	Tcl& tcl = Tcl::instance();
	if (argc == 3) {
		// attach a file for variable tracing
		if (strcmp(argv[1], "attach") == 0) {
			int mode;
			const char* id = argv[2];
			tchan_ = Tcl_GetChannel(tcl.interp(), (char*)id,
&mode);
			if (tchan_ == 0) {
				tcl.resultf("BLUE: trace: can't attach %s for
writing", id);
				return (TCL_ERROR);
			}
			return (TCL_OK);
		}

		// link stats
		if (strcmp(argv[1], "link") == 0) {
			LinkDelay* link_  =
(LinkDelay*)TclObject::lookup(argv[2]);
			if (link_ == 0) {
				tcl.resultf("Blue : No link delay Object %s\n",
					    argv[2]);
				return(TCL_ERROR);
			}
			bandwidth_ = link_->bandwidth();
			return(TCL_OK);
		}
		if (!strcmp(argv[1], "packetqueue-attach")) {
			delete q_;
			if (!(q_ = (PacketQueue*)
TclObject::lookup(argv[2])))
				return (TCL_ERROR);
			else {
				pq_ = q_;
				return (TCL_OK);
			}
		}
	}

	return Queue::command(argc, argv);
}

void Blue::reset()
{
	Queue::reset();
	/*
	 * Compute the "packet time constant" if we know the
	 * link bandwidth.  The ptc is the max number of (avg sized)
	 * pkts per second which can be placed on the link.
	 * The link bw is given in bits/sec, so scale mean psize
	 * accordingly.
	 */
	ptc_ = (bandwidth_ / (8 * mean_pktsize_));
	idle_ = 1;
	idletime_ = Scheduler::instance().clock();
```

```
	marking_prob_ = 0;
	marking_prob_trace_ = marking_prob_;
	freezetime_ = 0;
}

void Blue::inc_marking_prob()
{
	double now = Scheduler::instance().clock();
	int qlen = qib_ ? q_->byteLength() : q_->length();
	curq_ = qlen;

	if ((now - freezetime_ > last_update_time_) || (curq_ > blue_l_))
{
		last_update_time_ = now;
		marking_prob_ += inc_factor_;
		if (marking_prob_ > 1.0)
			marking_prob_ = 1.00;
		marking_prob_trace_ = marking_prob_;
	}
}

void Blue::dec_marking_prob()
{
	double now = Scheduler::instance().clock();
	if (now - freezetime_ > last_update_time_) {
		marking_prob_ -= dec_factor_;
		if (marking_prob_ < 0)
			marking_prob_ = 0.0;
		marking_prob_trace_ = marking_prob_;
	}
}

void Blue::enque(Packet* p)
{
	double now = Scheduler::instance().clock();

	bool dropped = false;
	bool ECNbit_set = false;
	hdr_flags* hf = hdr_flags::access(p);
	double u = Random::uniform();
	/* Find whether to drop or not */
	if (u <= marking_prob_) {
			if (setECNbit_ && hf->ect()) { /* if ECN enabled */
				hf->ce() = 1; /*set ECN bit */
				ECNbit_set = true;
			}
			else {
				drop(p); /* drop packet */
				dropped = true;
			}
			inc_marking_prob();
	}
	if (!dropped)
	{
		int qlim = qib_ ? (qlim_ * mean_pktsize_) : qlim_;
		q_->enque(p);
		if (q_->length() >= qlim) {
```

```
                if (!ECNbit_set)  inc_marking_prob();
                q_->remove(p);
                drop(p);
          }
     }
     int qlen = qib_ ? q_->byteLength() : q_->length();
     curq_ = qlen;     // for tracing purposes
}


Packet* Blue::deque()
{
     Packet* p = q_->deque();

     if (p != 0) {
          idle_ = 0;
     }
     else { /* if link is idle */
          dec_marking_prob();
          idle_ = 1;
          idletime_ = Scheduler::instance().clock();
     }
     int qlen = qib_ ? q_->byteLength() : q_->length();
     curq_ = qlen;     // for tracing purposes

     return (p);
}

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

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

     if (tchan_) {
           int n;
           double t = Scheduler::instance().clock();
           // XXX: be compatible with nsv1 RED trace entries
           if (strstr(v->name(), "curq") != NULL) {
                 sprintf(wrk, "A %g %d", t, int(*((TracedInt*) v)));
           } else {
                 sprintf(wrk, "P %g %g", t, double(*((TracedDouble*)
v)));
           }
           n = strlen(wrk);
           wrk[n] = '\n';
           wrk[n+1] = 0;
           (void)Tcl_Write(tchan_, wrk, n+1);
     }
     return;
}
```

Appendix D: Simulation Scripts

```
set ns [new Simulator]

set f [open out.tr w]
$ns trace-all $f
$ns namtrace-all [open out.nam w]

#set the no of TCP flows here
set nodenum 4

set start_time 0.0
set finish_time 60.0

# create the nodes
for {set i 0} {$i < $nodenum} {incr i} {

    set s($i) [$ns node]
    set r($i) [$ns node]
}
set n1 [$ns node]
set n2 [$ns node]

# create the links between the senders and n1, receivers and n2
for {set i 0} {$i < $nodenum} {incr i} {

    $ns duplex-link $s($i) $n1 1Mb 1ms DropTail
    $ns duplex-link $r($i) $n2 1Mb 1ms DropTail

}

#Bottle neck link between between n1 and n2
$ns simplex-link $n1 $n2 10Mbps 100ms Blue
$ns simplex-link $n2 $n1 10Mbps 100ms DropTail

#Configure BLUE queue parameters here
set blueq [[$ns link $n1 $n2] queue]
$blueq set inc_factor_ 0.002
$blueq set inc_factor_ 0.02
$blueq set freezetime_ 100ms
$blueq set setECNbit_ true

set tchan_ [open blueq.tr w]
$blueq trace curq_
$blueq trace marking_prob_trace_
$blueq attach $tchan_

#set the queue-limit between n1 and n2
$ns queue-limit $n1 $n2 100
# create TCP agents
for {set i 0} {$i < $nodenum} {incr i} {

    set tcp($i) [new Agent/TCP/Sack1]
    $tcp($i) set fid_ [expr ($i + 1)]
    $tcp($i) set ecn_ 1
```

```
    set sink($i) [new Agent/TCPSink/Sack1/DelAck]
    $sink($i) set ecn_ 1
    $ns attach-agent $s($i) $tcp($i)
    $ns attach-agent $r($i) $sink($i)
    $ns connect $tcp($i) $sink($i)
}

set srcnum 0
for {set j 0} {$j < $finish_time} {set j [expr ($j + 20.0)]} {
    for {set num 0} {$num < $nodenum} {incr num} {
      for {set k $srcnum} {$k < [expr ($srcnum + 50)]} {incr k} {
      set p($k) [new Application/Traffic/Pareto]
      $p($k) set packetSize_ 1000
      $p($k) set burst_time_ 200ms
      $p($k) set idle_time_ 200ms
      $p($k) set shape_ 1.5
      $p($k) set rate_ 10000K
      $p($k) attach-agent $tcp($num)

      $ns at $j "$p($k) start"
      }
    set srcnum [expr ($srcnum + 50)]
    }
}
$ns at $finish_time "finish"
proc finish {} {
    global ns sink nodenum srcnum tchan_
     set awkCode {
      {
          if ($1 == "A" && NF>2) {
            print $2, $3 >> "aq.tr";
            set end $2
          }
          else if ($1 == "P" && NF>2)
                  print $2, $3 >> "prb.tr";
      }
    }

    $ns flush-trace
    if { [info exists tchan_] } {
      close $tchan_
    }

    exec awk $awkCode blueq.tr

    puts "running nam..."
    exec nam out.nam &

    exit 0
}


$ns run
```