Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Telematics

IMPLEMENTATION AND EVALUATION OF THE BLUE ACTIVE QUEUE MANAGEMENT ALGORITHM

Author:	István Bartók	
Supervisor:	Ferenc Baumann	Budapest University of Technology and Economics
Industrial Consultants:	Imre Juhász	Telia Prosoft AB
	István Cselényi	Telia Research AB

Diploma Thesis

May 2001

Nyilatkozat

Alulírott Bartók István, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapesten, 2001. május 18-án

_____

Bartók István

Abstract

This work documents the implementation of the BLUE Active Queue Management algorithm for the Linux Operating System and its evaluation by measurements.

The issues of Active Queue Management (AQM), and the search for the most adequate AQM algorithms are important in these days because of the changing traffic dynamics – mostly congestion – on the Internet, and the rising quality expectations held by the customers.

One of the oldest and the most known AQM scheme is RED (Random Early Detection), but there are other proposed algorithms also. The BLUE algorithm is one of these, and tries to be superior in heavily congested situations by using a different approach, which decouples the queue management from the queue length.

The conclusion is that while BLUE outperforms RED in scalability and low setup resource demand, but if not available, RED can be also hand-tuned to achieve similar or close results.

Áttekintés

Ezen munka témája a csomagkapcsolt hálózatokhoz kifejlesztett BLUE Aktív Sorkezelő algoritmus megvalósítása Linux Operációs Rendszerre és az ezt követő mérés alapú vizsgálata.

Az utóbbi években az Internet forgalmának jellege – és ezzel az okozott torlódás is – jelentősen megváltozott, és a felhasználói tábor minőségi elvárásai is emelkedtek. Ennek köszönhető, hogy az Aktív Sorkezelés (Active Queue Management) kérdései, és a kívánalmaknak leginkább megfelelő sorkezelő algoritmusok kutatása egyre fontosabbak napjainkban.

Az egyik legrégebbi és legjobban ismert Aktív Sorkezelő algoritmus a RED (Random Early Detection), de a kutatói közösség sok más további algoritmust is felvetett már. A BLUE is ezek egyike. A BLUE függetleníteni igyekszik magát a sor hosszától, így próbálja a többi algoritmusnál jobban kezelni a súlyos torlódással járó helyzeteket.

A két algoritmus vizsgálatának végkövetkeztetése az, hogy bár a BLUE skálázhatóbb, alap beállításokkal is szinte minden körülmények között jól teljesítő mechanizmus, a RED megfelelő hangolásával ez az egyes esetekre megközelíthető.

Table of Contents

List of Figures and Tables

1 Introduction

In it's first years the Internet was rather a test network among several universities and research institutes. It worked well even with very simple flow control algorithms, because the offered traffic was usually less then the transfer capacity of the network.

When usage got higher, series of congestion collapses in the late '80-s motivated the research community to elaborate and deploy congestion control mechanisms such as Slow Start and Congestion Avoidance [1] for TCP (Transmission Control Protocol). This made the participants of the communication more aware of the network bottlenecks between them.

In the '90-s the Internet grow 100 to 1000-times again in number of hosts and users [2], and by orders of magnitude higher in traffic. The shift from a research network to a business and entertainment environment also changed the type and dynamics of the traffic. In spite of this, these algorithms serve surprisingly well even for now, with only little additions.

However, this period was dominated by telephone dial-in access and best-effort-only Internet traffic, significant conditions that are likely to change in the near future. Introduction of xDSL for residential users and 100 Mb/s Ethernet access for business customers results in a much higher traffic load on the distribution and core routers as well.

1.1 Motivation

For better interactive experience, we need lower RTT (Round Trip Time) in the Internet. As generally no RTT is good enough, IP carriers are pushed to ensure 100-200 millisecond, or even shorter delays on transatlantic connections. This can be achieved with smaller queuing delays, and further optimization of the adaptation layers between IP (Internet Protocol) and the physical media. However, to preserve the good performance the queuing delay in the access networks also needs to be decreased.

Using shorter queues is generally not a straightforward solution, as in most cases it causes higher packet loss ratio, which is contrary to our efforts to improve the Internet. While TCP can cope with the loss, its throughput significantly suffers when experiencing

packet loss rate above a few percent. In addition, the throughput shows high time-scale variations – affordable for long-lived connections, but a nightmare for HTTP traffic, which uses many short-lived connections.

For newly emerging interactive applications, such us (best-effort) Voice over IP and video-conferencing, the need for low delay and low packet loss rate is a clear requirement.

Additionally, with spreading broadband access the network bottlenecks tend to move from links used by only one user towards shared links used by many of them. In dial-in access, the limiting PSTN (Public Switched Telephone Network) last mile acted as a traffic limiter for the users, effecting in relatively low over-subscription of the access and backbones. Contrary, a typical broadband ISP (Internet Service Provider) offers a guaranteed bit rate, and shares the surplus bandwidth proportionally among the customers. This higher level of over-subscription is likely to cause congestion in the busy hours, as customers are potentially unlimited in their bandwidth-hunger in a flat-rate price package.

The default drop-tail algorithm on congested links with aggregated traffic tends to cause weird behavior. It keeps the queues always full, is unfair with bursty flows, and can effect in lockouts [3].

1.2 Random Early Detection

One of the first proposed solutions for the saturated transmit queues was RED (Random Early Detection) evaluated in [4]. When the link is congested, RED randomly drops arriving packets even if they would fit into the queue, to signalize congestion to the end nodes. The probability of the packet dropping is a function of the average queue length.

While RED is adequate in situations with moderate congestion levels, it has been shown, that – depending on its parameters – the queue length either oscillates, or the algorithm reacts to the changes in traffic very slowly [5]. Many researchers also criticize the behavior of RED under steady, but high congestion.

In addition, the deployment of Differentiated Services [6] and other schemes with multiple queues and packet schedulers changes the constant line speeds seen by the queues so far. A scenario where prioritized or Round Robin queues share a single link, effects in sudden departure rate changes, especially for the low priority queues.

RED is expected to handle the above situations non-optimally, so various improvements to RED and many brand new algorithms are proposed.

1.3 Explicit Congestion Notification

Standard TCP relies exclusively on packet loss to signal congestion. This practically prevents loss-free operation even with very slight congestion. ECN (Explicit Congestion Notification), proposed in RFC2481 [7], is an experimental standard intended to deliver congestion signals to end nodes without packet dropping. The idea is to let the routers mark the traversing packets when congested (but far before queue overflow), let the receivers mirror back the congestion signals, and the senders interpret them the same way as real packet loss. Some reserved bits in the IP and TCP headers have been experimentally allocated to carry the congestion information.

Combination of ECN and algorithms similar to RED are expected to help to achieve the goals like low delay and packet loss described in Section 1.1.

1.4 BLUE

One of the newly proposed algorithms for congestion signaling – either be ECN-marking or packet dropping – is BLUE [9]. This algorithm is not based on averaged queue length, rather takes a black box approach: it uses packet loss and link utilization history to maintain the congestion signaling probability. If the queue is dropping packets due to queue overflows, the probability is increased. If the link is underutilized, the probability is decreased. To avoid oscillations, it freezes the probability after every change for a fixed time interval.

The simulations done by its author promise to achieve practically no packet loss if used with ECN even under very high congestion. Note that RED cannot achieve this if the queue length is oscillating.

1.5 The Task

The task is to evaluate BLUE, whether it is better – and in which areas if it is – than other existing algorithms, especially RED. Most router vendors have already implemented RED in their devices, this accounts for comparing especially to it. Important part of the work is to add BLUE to the Linux operating system, as this addition

will be used in measurements with real TCP implementations. The measurements should be done with ECN-capable TCP traffic, with attention to the situation when ECN and non-ECN traffic share the congested link.

1.6 About This Work

The rest of the work is organized as follows. Section 2 gives a detailed background, with description of the TCP congestion control mechanisms and a review of BLUE and other recently proposed queue management schemes. Section 3 describes the design considerations of the Linux BLUE implementation. Section 4 summarizes the most important implementation details. Section 5 describes the tests performed to verify the robustness of the implementation and compares the performance of BLUE and RED based on measurements. Finally, Section 6 concludes with a suggestion of future work.

2 Background

TCP/IP, the communication protocol suite of the Internet, is a de facto standard constantly evolving with the implementations. It is common that parts of the standard are post-documentation of reference implementations. It is contrary to the conventional telecom world, where formal standardization bodies create the standards, and implementation typically starts only after the standardization process.

Some of the primary requirements at the foundation of the Internet were continuous extensibility, and interoperability of highly different computer networking worlds. Such, a layered architecture organized around a simple datagram-oriented Internetworking Layer was a straightforward design rule to start with. After years of evolvement the core scheme stabilized in Unix implementations as seen on Figure 1.

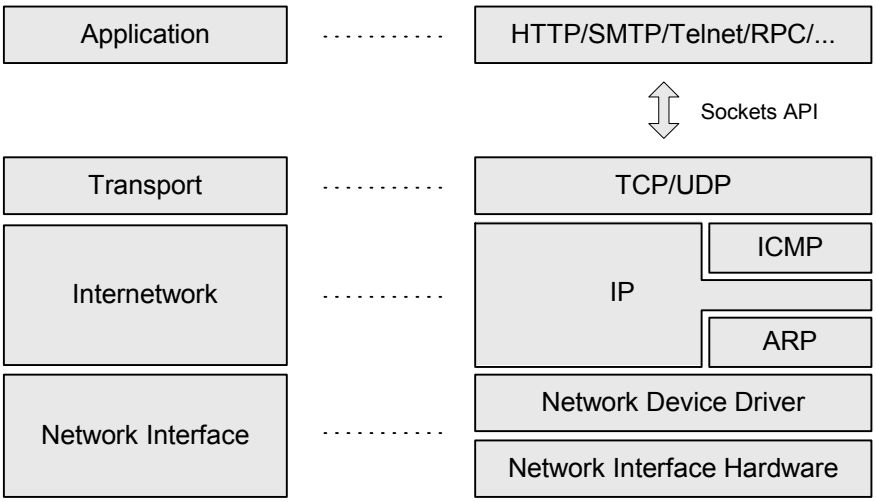

Figure 1 – TCP/IP protocol layers

Since the architecture was built bottom-up, it is most convenient to describe the layers in this order. However, this section does not aim a detailed review of all details of TCP/IP. It focuses on presenting the ones that are later referenced in the work.

2.1 Network Interface

The sites connected by the Internet had varying legacy computer equipment. Total homogenization of the computer and networking architectures used by the participants was not expected, not even in the future. Thus, the Internet have not specified its own link layer or a specific network interface, rather posed only minimal requirements on it and built upon these. Practically the only service a link layer is supposed to provide is unreliable datagram delivery.

This scheme was very successful, because the sites were free to select the underlying hardware, typically adapting lower layers of other communication equipment they used anyway. Good examples for this are X.25's LAPB (Link Access Procedure, Balanced) or LAN standards Ethernet and Token Ring.

2.2 Internet Protocol

The Internetworking Layer provides a unified view of the underlying lower layers. This allows the upper layers to communicate transparently with peers on any network connected to the Internet. Of course, the transparency means only the transparency of the interface and not its parameters like delay, etc.

IP (Internet Protocol) is the heart of this layer. IP is a connectionless protocol, and provides unreliable datagram service to the higher layers. Being so simple, it focuses on its main task, trying its best to deliver the datagram to the addressee.

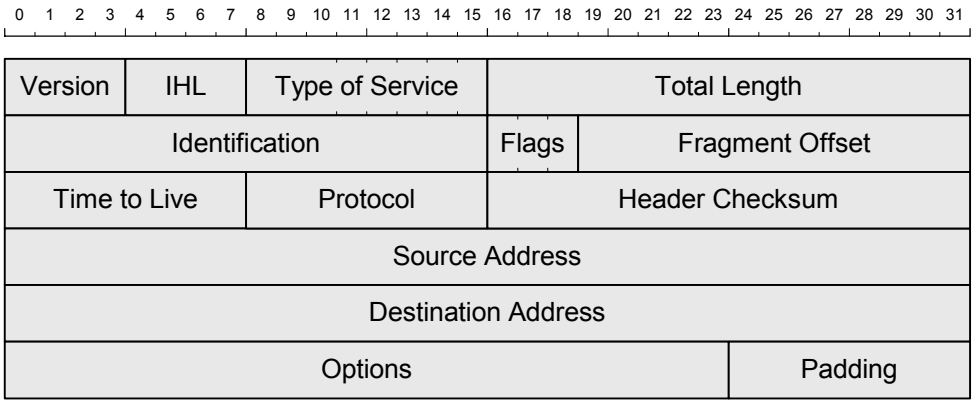

Figure 2 – IP Header

Figure 2 shows the IP Header as defined by RFC 791 [10]. Only the fields important for us are explained here, detailed description of the other fields can be found in [10].

Version

This 4-bit field indicates the IP protocol version. Different Version field usually indicates significant changes in the definition of the other fields, so they should be interpreted according to it. Such, Figure 2 applies only to IPv4 (IP version 4), which is the usual reading of IP. See for the IPv6 (IP version 6) header.

Type of Service

From RFC791 [10]: “Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. […] The major choice is a three way tradeoff between low-delay, high-reliability, and high-throughput.”

It was left to applications to set the ToS bits through the socket interface (with some restrictions). However, usage of this field was rare, and has not been fully consistent, so it was a place for experimentation by many research projects. It even was totally redefined by DiffServ in RFC2474 [11]. Figure 3 shows the original definition, while Figure 4 represents the new definition.

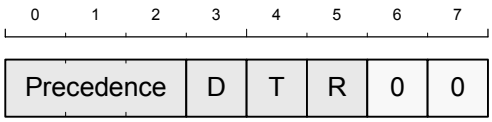

Figure 3 – Type of Service field

Bit 3	Low **D**elay
Bit 4	High **T**hroughput
Bit 5	High **R**eliability
Bits 6-7	Reserved for Future Use

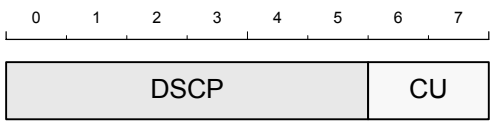

Figure 4 – DS-field

DSCP	Differentiated Services CodePoint
CU	Currently Unused

The Precedence field in the original definition used enumerated values with increasing priority. However, values other than *Routine* (the lowest priority) were used very rarely.

DSCP interprets bits 0-5 as an opaque integer value that classifies the traffic, and leaves the assignment of the classes partly open, permitting to vary between DS domains. DS domains change the DSCP field on their edges, according to the mapping arranged with the peer networks.

Note that the reserved bits remained the same with the new definition. These have been chosen later to carry part of the ECN information (See Section 2.4.12 for more details).

Header Checksum

From RFC791 [10]: “The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.”

The checksum is verified at each point the IP header is processed. When a header field changes, the checksum must be recomputed. This happens quite often, as every forwarder hop decrements the Time to Live field, or when an ECN-capable router changes the bits of the ToS field.

2.3 IPv6

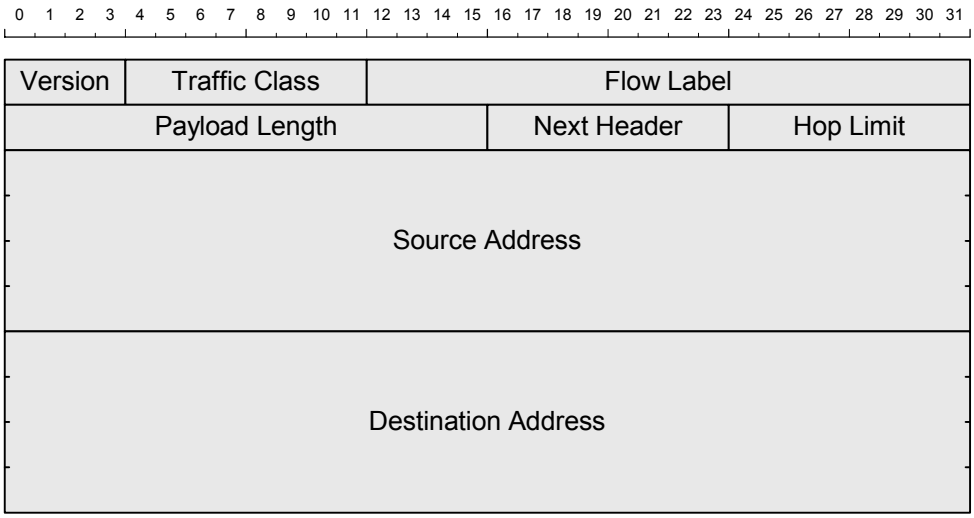

Figure 5 – IPv6 Header

IP version 6 defines its new IP header format. illustrates its current definition, based on RFC2460 [12].

Note that IPv6 does not protect its header with a checksum so it is not needed to recalculate it after the ECN-marking. Functionality of the ToS field or DS byte has been taken over by the Traffic Class field, which is used consistently with its IPv4 predecessor. Next Header has replaced the Protocol field.

2.4 Transmission Control Protocol

Dominant transport protocol of the Internet is TCP. It provides full duplex, reliable, connection-oriented service built upon the unreliable IP layer. It tries to utilize all available bandwidth, or shares it approximately fairly when more connections are competing for a bottleneck. Using TCP is straightforward for applications requiring a byte-stream.

TCP is an area so wide that heavy books could be written about it. This section tries to cover only the issues of the congestion control, which is important to understand the later parts of the document. See RFC793 [13] and the *TCP/IP Illustrated* series [16] [17] for full details on TCP.

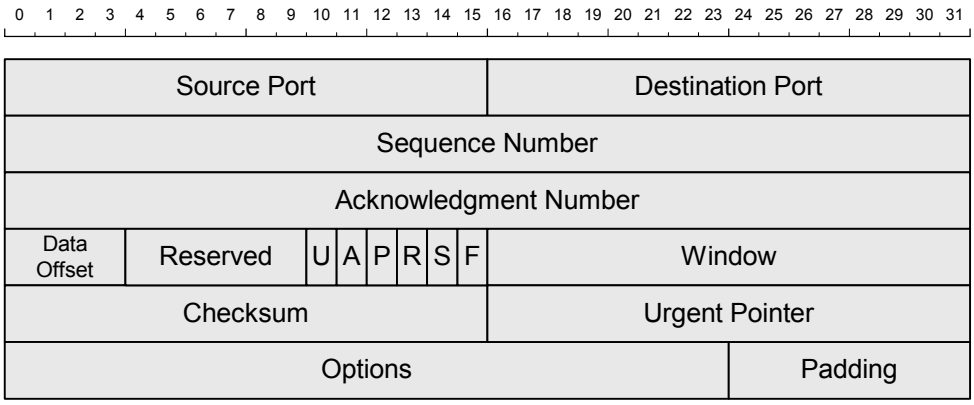

Figure 7 – TCP Header

TCP breaks the stream into pieces called *segments*, and precedes every segment with a header as shown in Figure 7. The result is then sent in an IP datagram. The receiver

responds with *acknowledgments*[1] to notify the sender about the successful reception of data. The acknowledgments use the same header as packets carrying data. Thus, a data segment can be combined with an acknowledgment in the same packet.

Source Port, Destination Port

A TCP connection is fully identified by the quadruplet formed by the source and destination IP addresses plus source and destination ports. This allows multiple simultaneous TCP connections between two hosts.

Sequence Number

Sequence number is the offset of data in the stream, measured in bytes. The field contains the sequence number of the first data byte in the segment.

Acknowledgment Number

The receiver sets this field in their acknowledgments to notify the sender about the next sequence number it is expecting to receive. This scheme is called *cumulative acknowledging*. Thus, one acknowledgment can cover multiple data segments.

Normally only one acknowledgment is sent per two data segments for established connections with continuous data flow. This saves some computing power and bandwidth.

Acknowledges can be delayed artificially by the receiver for a small amount of time. Its intention is to combine the incidental application-level answer with the acknowledgment. This time must be less then 200 milliseconds.

Control Bits

URG	Urgent Pointer field significant	not covered here
ACK	Acknowledgment field significant	
PSH	Push Function	not covered here
RST	Reset the connection	used for connection refuse/resetting

[1] The original TCP authors use the phrase *acknowledgment* while other literature and common language dictionaries often use *acknowledgement* instead. When used in TCP context and possible, this work tries to follow the original variant.

SYN	Synchronize sequence numbers	used for connection initialization
FIN	No more data from sender	used for connection close

Window

The receiver advertises the maximum sequence number it can accept (due to buffer space limitations), which is *Acknowledgment Number* + *Window*. This field is also called as *Advertised Window*. To overcome the limitation imposed by the 16-bit field, the Window Scaling extension has been introduced later (Section 2.4.8).

Checksum

Checksum contains a checksum of the header and the payload. Also covers a 96-bit pseudo header formulated from some fields of the IP header (See RFC973 [13] for the details of this pseudo header). Fortunately the pseudo header does not contain the ToS field, so recalculating this field is not necessary when changing the ToS value.

Options

Options are variable length fields beginning on byte boundaries. They have been defined to allow continuous evolution of the protocol. Most TCP extensions utilize Options to carry their specific control information.

2.4.1 TCP Connection Setup

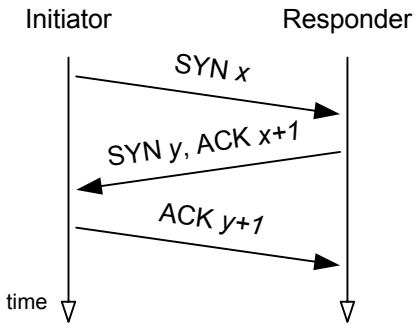

Figure 8 – TCP Connection Setup

illustrates the basic case of the three-way TCP connection setup. First, the connection initiator sends a SYN packet with its offered ISN (Initial Sequence Number). The other

end responds with a SYN-ACK, sending its ISN (note that the Sequence Numbers are independent for the two directions) and acknowledging the initiator's SYN. The initiator acknowledges the responder's SYN, and assumes that the connection is set up. The data transfer can start in any direction.

2.4.2 Flow Control

Flow control helps a fast sender to avoid flooding a slow receiver with data it cannot or do not want to receive yet[2]. The mechanism is based on the receiver's Advertised Window: the sender is not allowed to send data not fitting into the window. Figure 10 shows how the receiver's window limits the sender. The sender is also limited by it's own allocated buffer size (Sender Window).

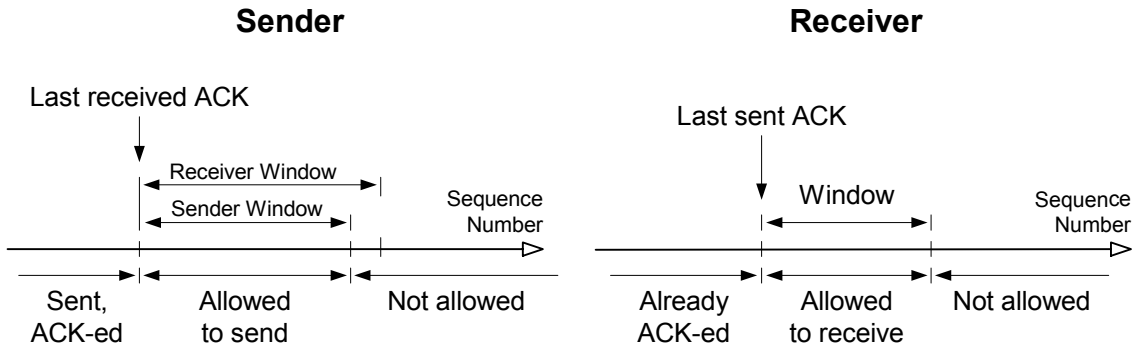

Figure 10 – TCP Windows

2.4.3 Retransmission

To achieve reliable operation, when a packet is lost, it must be retransmitted. The basic retransmission algorithm is as follows. TCP sender continuously estimates the Round Trip Time (RTT – the time between sending a segment, and receiving the ACK for it), and sets RTO (Retransmission Timeout) according to it when sending a segment. When the RTO expires, and the ACK still has not been arrived, the segment is retransmitted.

---

[2] The changed phrasing (sender/receiver vs. the initiator/responder in the previous subsection) is intentional to emphasize that there is no direct relation between that which endpoint initiates the TCP connection and which endpoint sends data in a given time later.

As recommended in [1], the estimated RTT is calculated with an EWMA (Exponentially Weighted Moving Average) function shown in Equation 2.

$$RTT_{smoothed} \leftarrow \alpha RTT_{smoothed} + (1-\alpha) RTT_{measured} \qquad \text{Equation 2}$$

Where $0 \le \alpha \le 1$ is a constant.

Equation 3 shows the inclusion of estimated RTT variation (*V* in the equation) into the RTO calculation – this helps to reduce unnecessary retransmissions while maintaining low RTO for constant RTTs.

$$RTO = RTT_{smoothed} + 4V \qquad \text{Equation 3}$$

Retransmitted segments can be lost also, so a timeout applies to them too. RTO is doubled after every retransmission of the same segment up to an upper bound, which is in the range of 60-100 seconds in most implementations.

Defined by RFC1122 [14], the initial RTT estimate for a new connection is zero. Variation has to be set to result in a 3-second initial RTO. While this value is reasonable, it causes long delays for short connections when a connection-initiating SYN packet is lost, as it will be retransmitted only after 3 seconds.

2.4.4 Congestion Avoidance

A situation when the sender and receiver windows are bigger then the buffering capacity of the network, effects in regular packet loss. However, this is usually the case when more TCP connections share a network path, especially if we use short transmit buffers. Hence, the sender must use one more limiting factor when sending the data: the estimated buffering capacity of the network. This estimate is called the Congestion Window (CWND). Figure 11 shows the final picture for the sender windows. Note that there can be any relation between the windows. The situation, when CWND is the smallest of the three – as seen on the figure – is only a typical example.

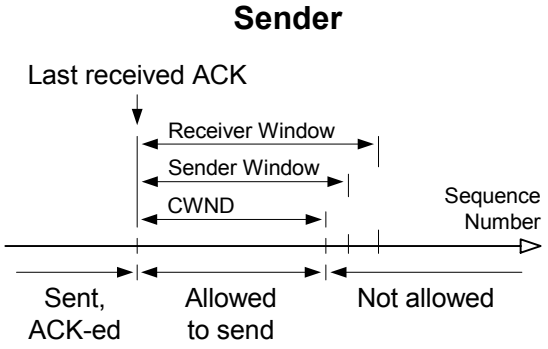

Figure 11 – TCP Sender with Congestion Window

Such, the upper limit on the resulting transfer rate (not considering the packet loss) is:

$$Rate_{\max} = \frac{\min(SenderWindow, ReceiverWindow, CongestionWindow)}{RTT}$$

Equation 4

The sender continuously tunes CWND during the transmission [15]. To maintain high utilization of the path, it is incremented by approximately one MSS (Maximum Segment Size) every RTT. When a packet loss occurs, it is a signal of overestimating the buffering capacity, so CWND is lowered (typically halved, but see Section 2.4.7 for the details of lowering). This idea of congestion control was developed as the solution for the congestion collapses in the late ‘80s [1].

2.4.5 Fast Retransmit

When a packet is lost, a sequence hole will formulate. This means that the receiver will get segments, which fit into its window but are not the next expected segment. It is assumed that the receiver will send acknowledge pointing to the start of the sequence hole for every such segment, so the sender will get duplicated ACKs (multiple ACKs with the same Acknowledge Number). The duplicated ACKs (*dupacks*) could also be a signal of packet reordering or packet duplication, but the assumption is that these are rare.

The BSD Tahoe TCP implementation introduced Fast Retransmit [15] based on this assumption. When three or more dupacks are received for a segment, the sender assumes that it has been lost and retransmits immediately (before RTO expires).

2.4.6 Slow Start

The CWND adjustment described in Section 2.4.4 opens the window too slowly for the new connections. This is notable on paths with high bandwidth-delay product. To overcome this, CWND is not incremented, but doubled every RTT at the beginning of a connection. This algorithm is called Slow Start [15].

Slow Start ends when CWND reaches the maximum window size or a threshold (*ssthresh*), or a packet loss occurs. Ssthresh is then maintained during the connection also: when a CWND halving happens, the lowered CWND value is also copied to ssthresh to keep it a lower estimate of the path capacity.

2.4.7 Fast Recovery

The BSD Tahoe version sets CWND to one MSS after the Fast Retransmit, and so followed with Slow Start to probe the network's buffering capacity again.

BSD Reno introduced Fast Recovery [15], which tries to maintain the data flow, while still adjusting the CWND to reflect a lower estimate of the buffering capacity. After detecting the packet loss and sending the fast retransmit, the sender enters the recovery phase, which is as follows.

If using only CWND halving, the sender would be quiet until getting the ACK that acknowledges all the data sent after the lost packet. After that, it would send burst of up to CWND packets. This would probably lead to packet loss again. Note that CWND would not be used properly in this situation. Remember that it represents the estimated buffering capacity of the network. However, limiting the sender to send only up to *Acknowledged + CWND* offset in the stream, the packets triggering the dupacks would not be considered in the calculation. The reception of every dupack is a signal of one data packet leaving the network. The burst at the end of the recovery state would be the result of not considering this.

Therefore, after halving (and copying it to ssthresh), CWND is incremented by 3 MSS to represent the 3 dupacks. It is incremented by one MSS for every further dupack received. There were *CWND - 3 MSS* bytes of data in flight at the time we entered the recovery phase. After getting dupacks matching *oldCWND/2* data (at about the half of the recovery period), CWND reaches its old value and the sender starts sending again.

When the ACK arrives that acknowledges new data (the receiver sends it when the retransmitted segment reaches him), the sender exits the recovery state. After that it discards the inflated CWND by copying back the halved value from ssthresh, and follows with the data transfer. It will not send a burst of packets, as it already sent exactly *oldCWND/2* data while in recovery.

This is the original algorithm implemented in BSD Reno. Linux implements the algorithm also. An implementation difference is that CWND is not inflated in Linux, but another state variables are used to keep track of the received dupacks [18].

Note that while Fast Recovery does not send a burst at the end of the recovery phase, it is quiet in the first half of the recovery period, and sends data with approximately the original rate in the second half. A possible improvement could be the Rate-Halving algorithm [19], which “adjusts the congestion window by spacing transmissions at the rate of one data segment per two segments acknowledged over the entire recovery period, thereby sustaining the self-clocking of TCP and avoiding a burst.” Rate-Halving exists in research TCP implementations, but no production TCP is known to use it.

NewReno [20] improves the original Fast Recovery algorithm by interpreting a partial acknowledgment (partial – which acknowledges beyond the original dupack point, but still in the recovery window) as a sign of another lost packet at that sequence number. This improves the throughput in the case when multiple packets are lost from one window of data. Reno would wait for an RTO for every such packet in this case and would follow with a Slow-Start. NewReno instead handles the situation with its improved recovery.

Major vendors implement Fast Recovery in their TCP implementations, most of them in a Reno or NewReno fashion.

2.4.8 Selective Acknowledgment

With cumulative acknowledgment, the sender needs to wait for one RTT to find out every lost packet. With SACK (Selective Acknowledgment) [21], the receiver can inform the sender about every successfully arrived segment explicitly, immediately disclosing the sequence holes.

The SACK-capability and the selective acknowledgment information are sent in TCP Options. The acknowledgment information is represented by edges of non-contiguous blocks of data that has been successfully arrived.

[22] compares Tahoe, Reno, NewReno and SACK TCP by simulations, and shows a tremendous advantage of SACK against Reno when multiple packets are lost from one window of data. It shows a light improvement even against NewReno.

The bias towards the throughput of the more advanced TCP implementations can make a significant difference when we want to control them with packet drop. Consider a highly congested link, shared between flows of the named different TCPs. The same packet drop rate will slow them down differently.

2.4.9 Forward Acknowledgment

FACK (Forward Acknowledgment) is a further refinement to TCP Fast Recovery. From [23]: “The goal of the FACK algorithm is to perform precise congestion control during recovery by keeping an accurate estimate of the amount of data outstanding in the network. In doing so, FACK attempts to preserve TCP's Self-clock and reduce the overall burstiness of TCP. […] The FACK algorithm uses the additional information provided by the SACK option to keep an explicit measure of the total number of bytes of data outstanding in the network.”

Note that the accuracy of this estimate is the size of the burst sent at the end of the recovery phase. Reno senders will underestimate the number of packets in flight when multiple packets are lost, as they know only about the first loss. Reno+SACK senders will behave the same, if they use the SACK information only to point out the packets to retransmit (as originally proposed).

FACK assumes that all not SACK-ed packets up to the rightmost SACK-ed packet were lost. Thus it underestimates the amount of data in flight when packet reordering takes effect. While this is legal in IP, it is a pathological behavior, characteristic to some network paths and not to the Internet as a whole [25]. Therefore, Linux falls back from FACK to NewReno (only for that connection) when the path is suspected to reorder packets.

2.4.10 TCP Vegas

To have a full picture it is important to note that there exists at least one research project – namely TCP Vegas [26] – that uses more advanced Congestion Avoidance mechanisms, not relying only on packet loss. For example it tries to avoid buffer

saturation with decreasing the CWND when the RTT is suspected to grow only because of the too much data sent into the network and the queues have been lengthened.

However, after years of research it is still not clear whether Vegas will be widely deployed, this is the reason of not considering its behavior in this work.

2.4.11 Other Extensions

RFC1323 [24] introduced three optional extensions to TCP to allow higher performance on long, high bitrate links. Of these, the following two are significant to us.

Window Scaling

The advertised free receiver window size can be a throughput-limiting factor for low-loss, high buffering capacity links, such as satellite links. Window Scaling allows bigger windows as it extends the TCP window from 16-bit, sending the most significant 16 bits of the extended value in the Window field of the header. The window can be extended by 1 to 16 bits. The size of the shift is negotiated at the connection setup.

Round Trip Time Measurement

RTTM (Round Trip Time Measurement) uses the TCP Timestamp Option to achieve more precise RTT estimate. From [24]: "…using TCP options, the sender places a timestamp in each data segment, and the receiver reflects these timestamps back in ACK segments. Then a single subtract gives the sender an accurate RTT measurement…"

The receiver sends back always the latest timestamp seen from the sender, so it gets a straightforward and accurate measured RTT with every received ACK.

Note that without RTTM, retransmitted segments can not be included into RTT estimate calculation, as we cannot decide whether the original or the resent packet triggered the ACK for it. What is worse, when a packet is lost, a whole window of data starting at the lost packet should be excluded from RTT estimation, as they are not acknowledged immediately on their arrival.

2.4.12 Explicit Congestion Notification

ECN is nothing revolutional, as similar schemes existed in other networking environments for years. Frame-Relay has FECN (Forward Explicit Congestion Notification) and BECN (Backward Explicit Congestion Notification), and ATM or DECNet also have their explicit notification.

The truth is that TCP ECN itself is also not new. It was published first in 1994 [27]. After years of further research and discussion, it was accepted in RFC2481 [7] as an experimental standard. The currently proposed version of the standard is [8], an Internet Draft expected to advance to Proposed Standard (and such an RFC) in these days.

The status of implementations is usually somewhere between the mentioned two, because of typically implementing earlier versions of the draft. This document follows the current Linux implementation, as it is the subject of the measurements. However, the implementation is expected to change when the final standard is accepted.

RFC2481 [7] redefines the IP ToS field as shown on Figure 12. The so far unreserved bits (see Figure 3 for the original definition) are used as follows.

Figure 12 – ECN bits in the IP ToS field

ECT ECN Capable Transport
CE Congestion Experienced

ECT

Keeping in mind the independent TCP/IP stack vendors, only gradual deployment of such a new standard can be expected. As ECN and non-ECN flows require different

handling at the ECN-capable routers, we need to explicitly distinguish them. The ECT bit is set to one for the ECN-capable flows.[3]

Note that not only TCP can utilize ECN, hence the word *Transport* in the name of the control bit. Utilization of ECN by UDP/RTP or other transport protocols is left for future research.

CE

ECN-capable routers can set the CE bit on packets they would otherwise drop to inform the end nodes about the congestion. The receiver must mirror back the congestion signal in its transport protocol, and the sender must react to this – in terms of congestion control – in the same way it would react to a lost packet. For TCP, it means halving the CWND. In terms of packet loss, setting the CE bit *(marking)* is a cheaper way of signaling congestion to the end nodes. The benefit for the individual flow – avoiding a possible RTO – is clean also. The end nodes must react to the packet loss also, as they would otherwise.

The routers are not expected to only mark (and pass) ECT packets when their buffers are completely full. They should drop the packet as in the old way. What is more, they are encouraged to drop also ECT packets when the congestion goes beyond moderate level. This is indented to serve as an emergency brake to avoid fatal unfairness between ECT and non-ECT flows (unfairness in any relation) or DoS (Denial of Service) attacks. However, there are no exact guidelines or standards on this yet.

Note that the IP header checksum must be recalculated when the CE bit is changed.

Figure 13 shows how [7] redefines the fourth word of the TCP header (See Figure 7 for the original). Two control bits have been allocated from the reserved space to be used by ECN.

[3] This definition of the ECT bit is consistent with RFC2481 [7]. The newest Internet Draft [8] defines the so far undefined ECT=0, CE=1 combination to be equivalent with ECT=1, CE=0. Note that the Linux kernel code does not follow yet this new definition.

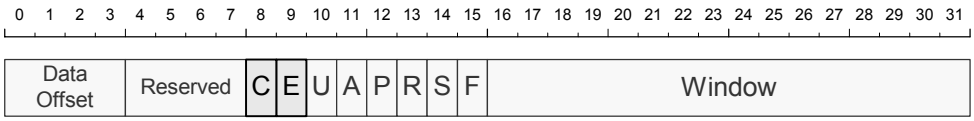

Figure 13 – ECN bits in the TCP header

CWR Congestion Window Reduced

ECE ECN Echo

ECE

When the receiver receives the packet with a CE bit set, it sets ECE in its next acknowledgment sent to the sender. In addition, it continues sending acknowledges with ECE bit, until it receives a packet with the CWR bit set from the sender.

CWR

From [8]: "When an ECN-Capable TCP sender reduces its congestion window for any reason (because of a retransmit timeout, a Fast Retransmit, or in response to an ECN Notification), the TCP sender sets the CWR flag in the TCP header of the first new data packet sent after the window reduction."

This scheme tries to avoid missing congestion signal information when the ACK, which mirrors back the CE bit in its ECE, is lost. Sending consequent ECE acknowledges will not lead to multiple reductions of CWND in the sender, as the sender does not react to ECE more then once every window of data.

If the CWND is already decreased to its minimum (1 MSS) – to slow down further – the sender should send its next packet only when the RTO expires. However, this *ECN Timeout* is not implemented in Linux [28] and FreeBSD [9] ECN implementations. In addition, while faced to only ECN marking and no packet loss, Linux decreases its CWND only down to 2 MSS. The developer's motives behind these modifications are to preserve the ACK-clock and avoid delaying the congestion notification information by the delayed ACKs. However, this effects in a more aggressive (than the standard) TCP in heavily congested situations.

There are some details of ECN that are significant, but may be not straightforward. For example there are numerous exceptions where the ECT bit cannot be set on packets of ECN-capable TCP connections:

- On the clean ACK packets (which do not carry data) – as it would break the basic principle of reacting to the ECN-marking the same way as to a lost packet. Note that the sender would not necessarily detect a lost ACK in the non-ECN case.
- On retransmitted data packets – for the DoS and other considerations in Section 6.1.5 of [8] on this.
- On connection-initiating packets – as we do not know in advance that the connection will be ECN-capable, only after the negotiation. This means that ECN cannot directly help us to avoid the 3-second initial RTO problem for short connections.

2.4.13 The Linux TCP Implementation

According to [18], the Linux 2.4 branch has a NewReno-behavioral TCP with SACK and FACK additions. It also supports features from RFC1323 [24] (Section 2.4.11). It contains full ECN support – has both ECN-capable TCP and ECN marking-capable AQMs.

2.5 Active Queue Management

Active Queue Management (AQM) is an advanced form of router queue management that tries to detect and react to the congestion prior to its fatal consequences such us full queues and bursty drops. In reaction to suspected congestion, AQM schemes drop packets *early* (or do ECN-marking) to signal the congestion to the end nodes.

The most important difference between the various AQM schemes is that *when* they suspect congestion, and *how* do they select the packets to be marked/dropped. In general, the congestion judgment can be based on current or averaged Q_{len}, on the traffic's arriving rate being higher than the departure rate, or other characteristics of the traffic or the queuing system, such us the number of recent tail-drops.

2.5.1 Random Early Detection

Random Early Detection [4] is a queue length based algorithm, as it uses the averaged Q_{len} to determine the probability with which it will mark or drop packets. The average is calculated with an EWMA function on every packet arrival. Figure 14 shows the marking probability as a function of the average Q_{len}. The probability is zero below a lower threshold, and is 1.0 above an upper threshold. Between the two it changes from zero to P_{max} linearly.

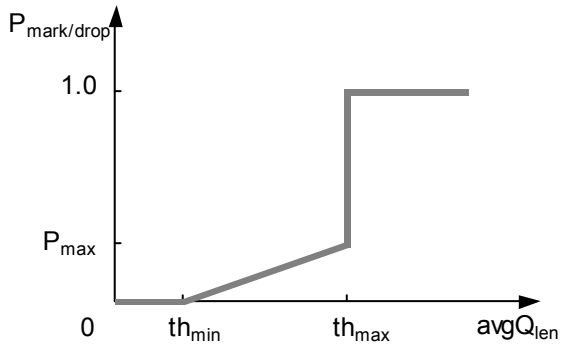

Figure 14 – Packet marking/dropping probability in RED

RED defines its parameters in a little peculiar way. Instead of directly setting the weight of the EWMA function, it is calculated from other parameters. The idea behind this design was to allow a mostly idle system (empty queue, zero average Q_{len}) to deterministically pass a given burst of packets without a single drop. Thus, the weight is set to a value so that the average Q_{len} reaches th_{min} only after passing *burst* number of packets.

Also uncomfortable parameters are the average packet length and the transmit rate (both are user-settable parameters in typical case) which are used to estimate the number of missed packet-slots for transmitter idle times. This is needed to correct the error introduced by long idle times in its packet-triggered average queue length calculation.

2.5.2 BLUE

The BLUE algorithm [9] is not based on averaged queue length, arrival/departure rate, or other explicit characteristics of the traffic. It focuses on the two types of events it tries to avoid: the tail-drops and link under-utilization. BLUE treats the system consisting of the queue and the traffic rather as a black box. Practically the only assumption is that if we increase the packet marking/dropping probability, the end nodes will respond to that with slowing their traffic and the system will shift from frequent tail drops towards link under-utilization.

The approach necessarily misses some possibilities to spot incipient congestion when compared to some other mechanisms. For example, the AVQ algorithm could signal congestion based on the arrival rate being higher then desired – even before the first dropped packet. However, not using explicit characteristic of the traffic, BLUE could be more robust in many situations where the heuristics built into other algorithms fail or are not scalable.

BLUE uses a single congestion signaling probability P_m. Packets traversing the queue are always ECN-marked with this probability regardless of the queue length. If a packet of a non-ECN flow is to be marked, it is dropped (the ECT bit of the ToS field is used to identify the ECN-capable flows).

If the queue is tail-dropping packets due to queue overflows, P_m is increased. If the link is underutilized, the probability is decreased. The simplest way to achieve this is a linear increase on packet drop, and linear decrease on a queue empty event with a freeze for a fixed time after every change to avoid oscillations. The queue empty event (when the network interface asks for a new packet to transmit, but the queue is empty) is used to signalize the link under-utilization, as when it happens, the link is supposed to be idle at least for a moment.[4]

[4] Note that this is not necessarily the case. Modern network interfaces can have their own transmit buffers for better efficiency. If these buffers are more than a few packets long, this can have a significant impact on packet schedulers or AQM schemes. However, unfortunately there is no mechanism to check whether the interface is really idle, so this approximation should be used with paying attention to the possible software tuning of these buffers to the absolute minimum required.

Note that contrary to RED, not a packet-based but a time-based memory is used. This allows conveniently setting the response time of the algorithm, or determining the time needed for the algorithm to go from zero P_m to 1.0. Note however, that the changes to P_m are still not triggered by a timer, but by the enqueued packets.

Figure 15 shows pseudo-code for the algorithm with two simple enhancements (published in the original BLUE paper [9] and supplementary source code):

- Build in a mechanism that tries to keep the queue much shorter than its maximum length, to reserve space for occasional bursts. This is a little contrary to the black-box approach, but this modification is clearly needed to achieve loss-free ECN operation as otherwise only the tail drops could trigger the P_m increase. The proposed addition is to increase when the actual queue length exceeds a predefined value, $L < Q_{max}$. Note that the averaging of this indicator is again covered by the black-box strategy and time-based memory, so this change is not expected to break the robustness of the algorithm.
- Build in an emergency brake: add an upper limit $P_{max} \geq P_m$.

Upon packet loss or $Q_{len} > L$ event:

```
if ( (now – last_update) > freeze_time ) {
       Pm = min(Pmax, Pm + inc);
       last_update = now;
}
```

Upon link idle event:

```
if ( (now – last_update) > freeze_time ) {
       Pm = max(0, Pm – dec);
       last_update = now;
}
```

Figure 15 – The BLUE algorithm

In the simulations done by its author BLUE achieves practically loss-free operation with ECN flows even under very high congestion, while having higher link utilization then RED. Note that the queue length of RED was oscillating in simulations performed in [9]. This could be a reason behind its worst performance. The referenced paper itself also

states, that using a much longer then originally suggested memory for RED it could achieve similar results.

The expected behavior is to achieve high link utilization without ECN also. While packet loss is unavoidable in this case, the ratio of tail versus early drops can make a big difference between RED and BLUE.

3 Design

This section documents the design steps of the Linux BLUE implementation. The author of BLUE provides an implementation for the FreeBSD ALTQ framework [29], but there are several motives behind choosing especially Linux for our implementation:

- The Linux Traffic Control Framework is part of the standard Linux kernel distribution. Although ALTQ is synchronized with the KAME IPv6 project[5], it is not released with the mainstream *BSD releases. This could make supporting a product utilizing BLUE potentially harder.
- Linux has better support for applications that are beyond a simple Unix-like server (better support for real-time, embedding, etc.)
- The lab where the work was carried out has history and experience rather with Linux-based developments and networking devices. This makes the integration into existing projects easier.

3.1 Requirements Specification

To achieve our goals the implementation has to fulfill the following requirements:

- Integrate well into the existing Traffic Control Framework – it is important for consistent management. Also, if BLUE is proven superior, this way it can be easily included into the mainstream Linux kernels.
- Try to be consistent with the existing Linux RED and ECN implementation – when it is possible, try to follow the practice introduced by existing implementations of related standards. Good examples for this are to allow turning on or off the ECN-marking, or interpreting the ECT code points according to RFC2481 [7] respective [8].
- Be robust enough to use in my measurements and to use by other people for testing, measurements, or including into the mainstream kernel.

---

[5] KAME is one of the alternative IPv6 implementations for Free/Net/OpenBSD

- Try to be at least as effective as the Linux RED implementation is.

3.2 Variations of BLUE

Two variations of the algorithm described in the BLUE paper [9] can make minor changes to its behavior, so they were chosen to incorporate into the code in a way that it can be easily recompiled with any combination of them:

- Single update time – the timestamps made on P_m changes can be separated for the incrementing or decrementing, or they can use a common timestamp.
- Use, or not the $L < Q_{max}$ value described in Section 2.5.2 – the L could be also a parameter, but for the beginning I decided to use $L = Q_{max}/2$ if using this is selected (consistently with the BLUE author's simulations).

There could be other variations of the algorithm, based on whether the queue length is calculated in bytes, or packets. The length of the packet could be also calculated into the probability of dropping, as discussed in [30] for RED.

The decision was to use a byte-based queue length, but do not use the packet length in the packet dropping calculation.

- The Linux RED implementation does it this way, and so the comparison is less ambiguous.
- While the byte mode suggested by [30] tries to eliminate the bias against flows with small packets (small MTU of the path), and is definitively an important future work area, the proposed solution (packet dropping probability is a linear function of the packet length) is probably too aggressive. Consider a 10% packet dropping probability for 500-byte packets. The resulting 80% probability for 4 kB packets or more the 100% for 8 kB packets is probably not a fair weighting[6]. As working out the optimal function is beyond the scope of this work, the decision was to implement per-packet dropping.

---

[6] These MSS values seem too large for nowadays Ethernet-dominant networks, but note that many Fast Ethernet devices already support it unofficially. These and even higher packet sizes are considered for high utilization Gigabit Ethernet networking to reduce protocol overhead in the end nodes and routers.

3.3 Parameters

Based on the above, the parameters that are settable from the user-space are the following.

```
struct tc_blue_qopt
{
        __u32   limit;
        int     freeze_time;
        int     pmark_init;
        int     pmark_inc;
        int     pmark_dec;
        int     pmark_max;
        __u32   flags;
};

#define TC_BLUE_ECN    1        /* The flag 'ecn' */
```

Limit

The Q_{max} limit of the queue length, measured in bytes. This value will be interpreted in a way that no new packets are enqueued into the queue until the current Q_{len} is higher then Q_{max}. Note that this implies that Q_{len} can exceed Q_{max} at most with one MTU. On the other side this approach is fairer between small and big packets, this is why the Linux RED and byte-FIFO implementation uses the same.

The parameter is a 32-bit unsigned integer.

Freeze time

Minimum time between the P_m updates. Although the resolution of the timers in a Unix kernel is typically in the range of milliseconds, the precision of time measurements can be significantly finer, even one microsecond. This is the case for Linux also, so this parameter can be set in microseconds.

P_m initial

For measurements it can be a useful feature that the initial P_m value is not zero, but can be set explicitly. With choosing zero increment and decrement this is an easy way to achieve a fixed packet marking/dropping probability. The parameter is an integer, Section 3.8 describes how are integers used to represent the probability values.

P_m increment

$0 \leq P_m$ *increment* < 1 is the P_m increment step. P_m is increased with this value on a packet drop or $Q_{len} > L$ event.

P_m decrement

$0 \leq P_m$ *decrement* < 1 is the P_m decrement step. P_m is decreased with this value when dequeue is requested with an empty queue.

P_m maximum

This is the P_{max} upper limit on P_m.

Flags

There is only one flag for now: *ecn*. If this is set, the algorithm does marking for ECN-capable flows, and dropping for non-capable ones. If it is not set, dropping is done for both.

Parameter	Value
Freeze time	10 ms
P_m initial	0.0
P_m increment	0.0025
P_m decrement	0.00125
P_m maximum	1.0
Flags	–

Table 1 – Default parameters for BLUE

3.4 Variables of the Algorithm

Only a minimal set of internal variables is needed, as follows.

Backlog

Backlog is the current Q_{len} in bytes. It is increased on every successfully enqueued packet with the length of the packet, and decreased with it when the packet leaves the

queue. This is not a really private variable, as the BLUE uses the variable *sch→stats.backlog* provided by the TC (Traffic Control) framework. This way it is automatically shown in the statistics.

P_m

The current packet marking/dropping probability. (*pmark* in the code)

Last update / Last increment + Last decrement

The timestamp of the last change to P_m. Depending on the chosen variation of the algorithm, it can be a single or separated value. (*last_update* vs. *last_inc* and *last_dec* in the code)

3.5 Statistics

```
struct tc_blue_xstats
{
        int     pmark;
        __u32   marked;
        __u32   early_drops;
        __u32   limit_drops;
        __u32   other_drops;
};
```

P_m

The current P_m is shown in the statistics. It is useful to see it, as this is the most important internal state variable of the algorithm.

Marked packets

The number of packets that have been ECN-marked by the probability decision.

Early drops

The number of packets that have been dropped by the algorithm by the probability decision.

Limit drops

The number of packets dropped because of queue overflow *($Q_{len} > Q_{max}$)*.

Other drops

The number of packets dropped because of an explicit *drop()* function call from the TC framework. Packet schedulers such as CBQ (Class Based Queuing) can penalize queues with this mechanism.

3.6 System Overview

Figure 16 shows where the Linux kernel TC (Traffic Control) framework fits into the outgoing path from the Network layer towards the Network Interface. This section focuses on settling the BLUE implementation in it, a detailed description of the framework can be found in [32].

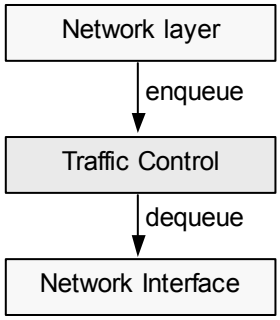

Figure 16 – Traffic Control in the Linux kernel

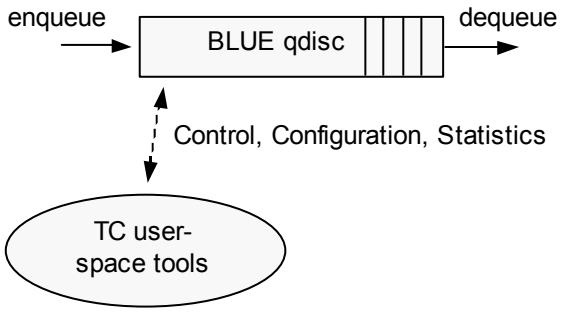

Figure 17 – Using the BLUE queue standalone

The TC framework allows changing the default drop-tail FIFO queuing algorithm modularly. Its modules are:

Qdisc

Packet schedulers and queuing algorithms are implemented in *qdisc* (queuing discipline) modules. They are fed by the framework with packets through their *enqueue()* function, and they are asked through their *dequeue()* function to output packets.

Class

A qdisc can have classes, which are typically used in packet schedulers to represent the different queues. The classes can contain a qdisc again, so this structure allows unusually flexible queuing configurations.

Filter

Filters are used to direct the packets into the classes. Filters are like in a packet-filter firewall, they can make the classification decision based on packet header fields.

BLUE is implemented as a qdisc module, and is typically used as a root qdisc (runs standalone, replaces the drop-tail FIFO) for a network interface as shown in Figure 17.

Figure 18 shows BLUE when used as a queuing algorithm for one of the queues of a classful qdisc, for example the CBQ packet scheduler.

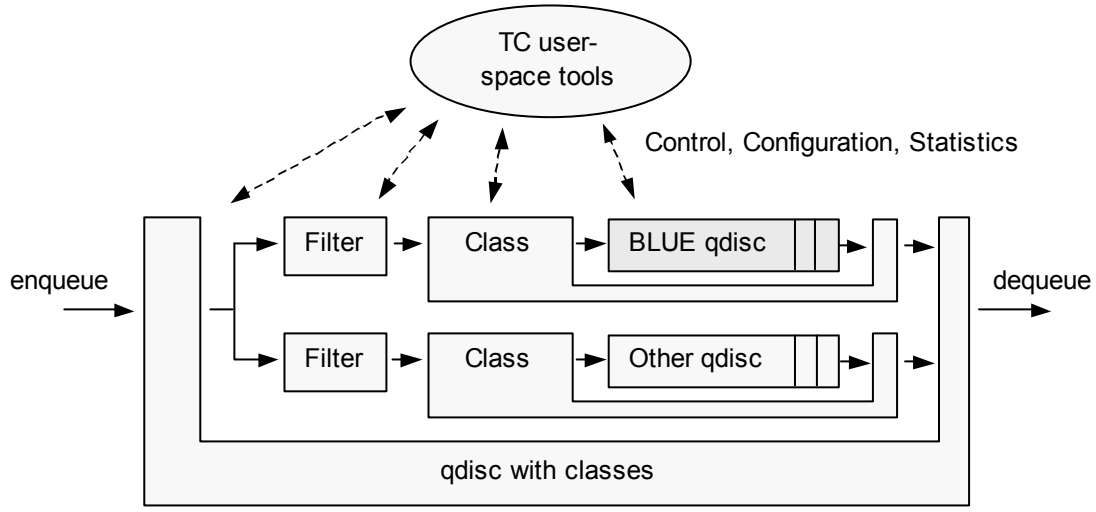

Figure 18 – Using the BLUE queue as a sub-qdisc of another class-based qdisc

For managing the framework – setting up or deleting structures and gathering statistics information – the Netlink interface [33] is used between the user-space configuration program and the kernel-space framework.

3.7 The Interface

To implement the qdisc functionality and to interface to the kernel, we have to fill in and register a C struct containing a few parameters and addresses of functions. This method, resembling the virtual functions of the object-oriented methodology is a common way of inserting new modules (file systems, device drivers) into the open interfaces of the kernel. The struct to fill in is *Qdisc_ops* in this case:

```
struct Qdisc_ops
{
        struct Qdisc_ops        *next;
        struct Qdisc_class_ops  *cl_ops;
        char                    id[IFNAMSIZ];
        int                     priv_size;

        int                     (*enqueue)(struct sk_buff *, struct Qdisc *);
        struct sk_buff *        (*dequeue)(struct Qdisc *);
        int                     (*requeue)(struct sk_buff *, struct Qdisc *);
        int                     (*drop)(struct Qdisc *);

        int                     (*init)(struct Qdisc *, struct rtattr *arg);
        void                    (*reset)(struct Qdisc *);
        void                    (*destroy)(struct Qdisc *);
        int                     (*change)(struct Qdisc *, struct rtattr *arg);

        int                     (*dump)(struct Qdisc *, struct sk_buff *);
};
```

The BLUE module will fill in the struct as shown below:

```
struct Qdisc_ops blue_qdisc_ops =
{
        NULL,
        NULL,
        "blue",
        sizeof(struct blue_sched_data),

        blue_enqueue,
        blue_dequeue,
        …
        blue_dump,
}
```

Some clarification of the fields of this struct (see [32] for more details)

next Used internally by the TC framework – the available qdiscs are kept in a linked list.

cl_ops Pointer to a similar structure describing the class operations functionality. As BLUE is a classless queue, this is NULL.

id A character string, the name of the queuing discipline.

priv_size The size of the private data struct. The framework will allocate an area of this size before init (and free the area after destroy) and pass its address to the qdisc.

A short summary of the functions to be implemented (their addresses will be passed in the struct for registering) is as follows.

blue_enqueue Enqueues a packet. Parameters: the packet and the destination qdisc. Returns the result of the queuing: success, success with congestion, or drop.

blue_dequeue Asks the next packet from the qdisc for transmitting on the network interface. Parameter: the desired source qdisc. Returns the dequeued packet or NULL if no packet is to send.

blue_requeue Puts a packet back to the queue (in the front), undoing the results of a dequeue call. It is needed because of some broken network interfaces that can request a packet to transmit but change their mind on transmit problems. Parameters: the packet and the destination qdisc. Returns the result of the operation (however, it really should be successful).

blue_drop Drops a packet from the queue (to penalize queues eg. by CBQ). Parameter: the desired qdisc. Reports in its return value whether a packet has been dropped – note that the queue can be empty.

blue_init Initializes and configures the qdisc. Parameters: the target qdisc, and the struct describing the desired configuration. Informs the caller about success or failure in its return value.

blue_reset Resets the qdisc: clears the queue and sets back its state variables to the initial values. Parameter: the target qdisc. It always should succeed, so it does not have a return value.

blue_destroy The opposite of blue_init, it prepares the removing of the qdisc given in parameter. It should always succeed also.

blue_change Requests to change the configuration of the qdisc, but without full init. Parameters: the target qdisc and the desired configuration. Reports the success or failure in its return value.

blue_dump Dumps diagnostic data. Its main use is to return configuration information (needed to set up a qdisc like the questioned), and statistics. Parameters: the qdisc in question and a Netlink packet into which the dumped data will be written. Returns success or failure.

3.8 Fixed-Point Arithmetic

Note that the *float* or *double* types of the language C cannot be used in the Linux kernel code. To represent the probability values (P_m, initial P_m, P_{max}, increment, and decrement) and to allow fast computations on them, a fixed-point arithmetic is needed.

As we only use the *0 ≤ value ≤ 1* range for these, and want a simple checking for over/underflows, it is straightforward to use the two's complement fractional fixed-point format from the DSP (Digital Signal Processor) world:

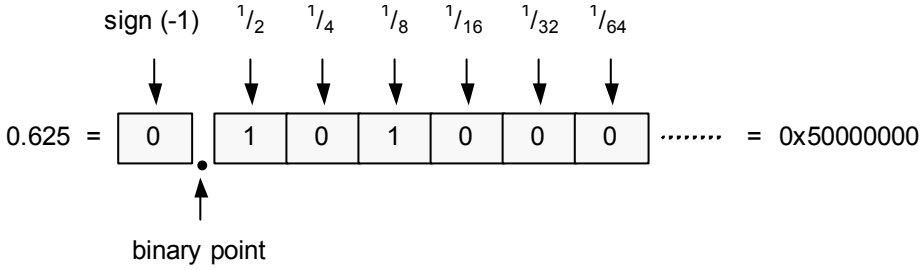

Figure 19 – The Fixed-Point format used to represent P_m

Note that this scheme can describe only values in the *[-1, 1-2^{31}]* range, so only a close approximation of 1.0 can be represented. However, this precision is enough for our purposes.

4 Implementation

4.1 Software Environment

As the implementation consists of modifying other programs that were written in C, the programming language (and the GNU gcc compiler, because of at least the Linux kernel depends on it) is so determined.

As a base platform, the Debian GNU/Linux 2.2 (potato) release was chosen. One of the main reasons for this was the wide user community support for the software, so its maintenance (especially security updates) requires only very low resources from the system administrator. We have many Linux computers connected openly to the Internet in the lab, so the absolute minimum maintenance level is to have them security-patched always up-to-date.

The kernel and some of its close surroundings have been upgraded to kernel version 2.4.3. As this new stable branch has been available for more than half a year, it makes sense to develop new additions such as BLUE primarily for that. It is expected that most of the experimenter people (potential users and feed-backers of the BLUE implementation) prefer this branch to 2.2 versions. The 2.4 series also has ECN support built in by default, which is important for this work.

To allow easy including into the mainstream sources, the newest available snapshot of the TC command line configuration tool was used, namely from package iproute2-001007 (released on 7th Oct 2000).

The tcpdump packet sniffer has been modified to dump the ECN control bits in the IP and TCP headers. As this modification is a very short code, the easier integration into the used Linux distribution had a higher priority then using the latest snapshot from the developers of the tcpdump package. Hence, the Debian source package of tcpdump-3.4a6 was selected as the basis. See Appendix A for the detailed modifications.

4.2 Time Measurement Granularity

The default time measurement in the Linux kernel is based on the *jiffies* variable. This global variable always contains the number of ticks (of the kernel 100 HZ timer) elapsed from the booting of the system. If this precision is adequate, it can be used for time measurements. The internal timers can work also with jiffies granularity. For example, there is an efficient mechanism in the kernel to ask to call a callback function (still in the kernel of course) at a given jiffies time. TCP timers are implemented like this, so having a better resolution would effect in better granularity of the TCP retransmission or delayed acknowledge timers. This can be important if measurements are to be done with only a few traffic generator computers.

The default 100 HZ rate can be speeded up in a limited fashion. Using 1024 HZ is generally not considered hairy, as most of the kernel is written with this possibility in mind – Linux on the Digital Alpha platform uses 1024 HZ because of the hardware suggestion. However, on Intel 32 platforms one problem can arise sooner this way: the jiffies variable is an unsigned long, and various timing problems can appear in certain parts of the kernel at its wrap-around. Using 100 HZ this occurs after 497 days, with 1024 HZ this time is shortened to approximately 48.5 days. The following modification to the Linux kernel is needed to change the tick rate to 1024 HZ:

```
diff -urN -X dontdiff v2.4.3/linux/include/asm-i386/param.h linux-2.4.3-
blue/include/asm-i386/param.h
--- v2.4.3/linux/include/asm-i386/param.h       Fri Oct 27 20:04:43 2000
+++ linux-2.4.3-blue/include/asm-i386/param.h   Wed Apr 18 19:25:18 2001
@@ -2,7 +2,7 @@
 #define _ASMi386_PARAM_H

 #ifndef HZ
-#define HZ 100
+#define HZ 1024
 #endif

 #define EXEC_PAGESIZE  4096
```

As packet schedulers typically need better then 10-millisecond (even maybe better then 1-millisecond) resolution, the TC framework contains various solutions for microsecond resolution time measurement. Depending on the used hardware, this can be surprisingly effective. On the Intel Pentium (and higher) and Digital Alpha processor platforms the provided machine code instruction can be utilized to achieve a very accurate timestamp quickly. These processors have high-resolution time measurement availability built in, in form of a readable register counting at the processor clock. The following modification to

the Linux kernel turns on the usage of this feature (otherwise a jiffies-based mechanism is used):

```
diff -urN -X dontdiff v2.4.3/linux/include/net/pkt_sched.h linux-2.4.3-
blue/include/net/pkt_sched.h
--- v2.4.3/linux/include/net/pkt_sched.h        Tue Mar 27 01:48:17 2001
+++ linux-2.4.3-blue/include/net/pkt_sched.h    Thu May 24 19:30:23 2001
@@ -5,7 +5,7 @@
 #define PSCHED_JIFFIES                  2
 #define PSCHED_CPU              3

-#define PSCHED_CLOCK_SOURCE     PSCHED_JIFFIES
+#define PSCHED_CLOCK_SOURCE     PSCHED_CPU

 #include <linux/config.h>
 #include <linux/pkt_sched.h>
```

Both the traffic generator and the BLUE queuing router machines used these modifications during the development and following measurements. There were no fatal problems with them, but it seems they interfere to NTP (Network Time Protocol) timekeeping, because the NTP Daemon running on the machines complained more often into the logs about losing synchronism than without them.

4.3 The Running System

The implemented system can be set up the following way (Consult [34] if not familiar with the *tc* command):

- Load the kernel module that implements the BLUE queue.
- Replace the default packet FIFO queue of the selected network interface with BLUE using the desired parameters.

```
blue:~# modprobe sch_blue
blue:~# tc qdisc replace dev eth2 root blue help
Usage: ... blue limit BYTES [ freeze TIME ] [ init PROBABILITY ]
           [ inc PROBABILITY ] [ dec PROBABILITY ] [ max PROBABILITY ]
           [ ecn ]
blue:~# tc qdisc replace dev eth2 root blue limit 50kB freeze 10ms inc 0.0025
dec 0.00125 max 1.0 ecn
blue:~#
```

After running some traffic through the queue, the statistics can be shown with this command:

```
blue:~# tc -s qdisc
qdisc blue 8002: dev eth2 limit 50Kb freeze 10.0ms inc 0.002500 dec 0.001250
max 1.000000 ecn
 Sent 41699148 bytes 28128 pkts (dropped 180, overlimits 7465)
 backlog 15140b 10p
  pmark 0.277500, marked 7288, drops: early 177 limit 3 other 0
blue:~#
```

The command dumps back the parameters used to setup the queue in the first line of its output. The second line (indented by a space) contains standard traffic statistics summary of the queue: successfully enqueued bytes and packets, and the number of total drops and *overlimit* situations (when the queue throttled the traffic in a form of an early drop or ECN mark). If there is a non-zero queue, its length is also reported both in bytes and packets. This is also standard for all queues. The last line (indented bye one more space) contains the private statistics of the queue, in this case as described in Section 3.5.

The source code of the implementation and all connected material can be downloaded from the project's home page [35].

5 Testing and Measurements

The subject of the test is the BLUE kernel and user-space code. The goal is to verify the usability and robustness of the code and after that perform traffic measurements to evaluate the algorithm.

5.1 Module testing

As the environment code – the TC framework – is not well documented, it is important to make sure that the assumptions made in the design and implementation phases are right. In addition, many things in the Linux kernel are subject to change in the future without notice in the nowadays documentation or code. It is clear that a test suite that is constantly maintained with the code can significantly improve the quality of a product.

It is expected that a software module of a (desired) high-availability telecommunication device is tested more systematically at least for the very trivial errors than just run a few times with common-use conditions and start searching for the bugs only when the failures appear in the test operations phase. Even stress testing with traffic generators – while important – is able to disclose only some of the problems and its black-box approach can generate hardly reproducible errors.

Module testing aims the elimination of trivial coding errors and wrong or outdated assumptions with a gray-box approach. It separates the code into modules, and tries to verify the functionality of the modules knowing their internals, and following the more code paths. It is built on the principle that even the simplest program code has nonzero probability to contain errors. Thousands of newly written program code lines are likely to contain wrong pointer initializations, never tried (and crashing) error handling paths, unhandled memory allocation failures and other similar bugs. Module testing tries to detect these with double-checking in controlled situations.

Using the classical module testing approach for such a small project would be a little over-engineering, but it makes sense to utilize its basic ideas. Providing a test suite with the software allows the people with other software or hardware configuration not available in the lab (eg. 64-bit, or big endian processors) to perform easily the basic self-checks of the software.

From the viewpoint of testing, the ideal software should consist of well-separated modules, which could be tested separately. This is often not the case in the real-world situations, as most software has not been designed with module testing in mind, or uses a complex environment that is not easy to substitute or emulate.

In our case both the kernel and user-space sources are small modules integrated into rather monolithic software – this is the typical case for using a complex environment. Thus only limited module testing can be done, and the module test code is to be included into the tested module, and run at initialization time in its real-life environment.

Partial module testing of BLUE has been performed, with focus on the Fixed-Point Arithmetic code which is most suspected to be a source of latent bugs, especially when run on non-i386 CPU architectures.

5.2 Performance Testing

The validity and forwarding performance of the implemented queuing module has been verified with IXIA 1600 traffic generator.

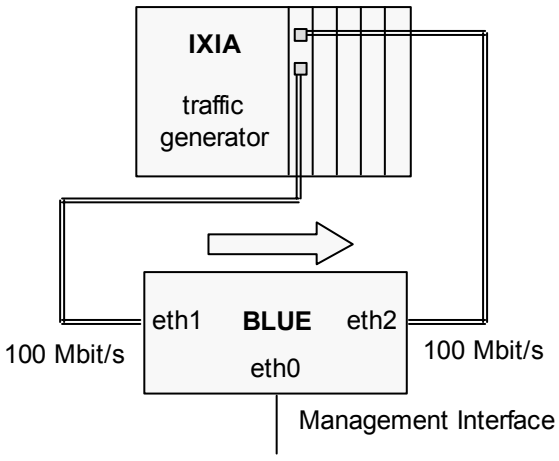

Figure 20 – Performance testing setup

Figure 20 shows the measurement setup. The Linux computer named BLUE has three Fast Ethernet interfaces, and is connected to the traffic generator with crossover UTP (Unshielded Twisted Pair) cables to achieve full-duplex 100 Mbit/second line speed.[7]

The measurements were done with the eth2 interface set up to use the byte-FIFO, RED, and BLUE queuing mechanisms in three turns. The following commands were used to set up the queues:

```
byte-FIFO:  tc qdisc replace dev eth2 root bfifo limit 50kB
RED:        tc qdisc replace dev eth2 root red limit 50kB min 10kB max 40kB
             avpkt 600 burst 100 probability 0.1 bandwidth 100Mbit ecn
BLUE:       tc qdisc replace dev eth2 root blue limit 50kB ecn
```

For all three cases the maximum queue length was 50 kB.

The BLUE implementation was used mostly with its default parameters (See Table 1).

The forwarding performance was tested with one-way traffic of evenly paced UDP packets (with the ECT bit set), measuring the CPU utilization on the Linux machine with the *vmstat* command. Note that the UDP traffic is used here to verify the forwarding performance of the mechanism, not its marking or dropping correctness. The ECT bit is set to avoid early packet drops. Figure 21 shows the results with 1518-byte Ethernet frames.

[7] See Appendix B for the exact hardware configuration of Linux machines used in the measurements

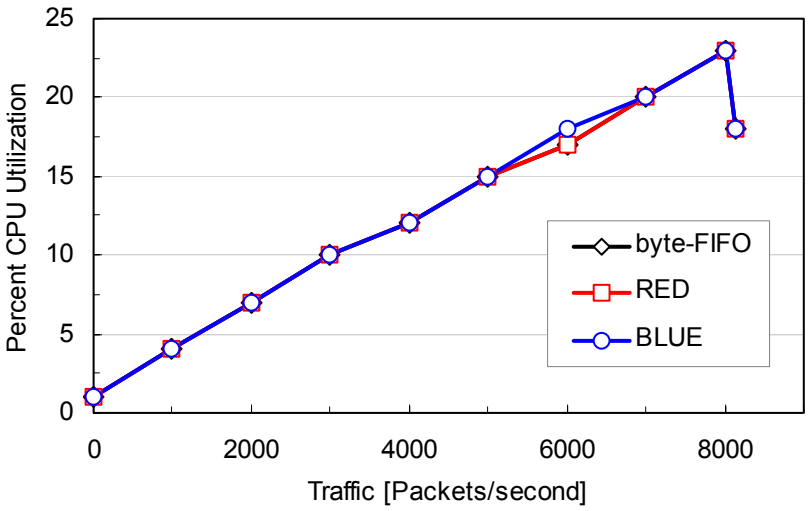

Figure 21 – Forwarding Performance with 1518-byte frames

The measured three queuing mechanisms do not show a significant difference and as expected, the CPU demand is approximately a linear function of the offered traffic. The Linux machine handled the traffic without packet loss.

Note the CPU demand decrease at 8127 Packets/second – this is the 100% utilization point of the 100 Mbit/second Fast Ethernet. The reason behind the decrease is that when more consecutive frames are sent or received on a network interface – depending on the hardware and device driver implementation – more frames can be handled with one hardware interrupt service. This helps the performance a lot, as the bottleneck in PC-based routers is rather the interrupt-handling, and not the forwarding or queuing itself. The number of hardware interrupts per second – as monitored with *vmstat* – showed about the same decrease, it dropped from the approximately 2 interrupts per a forwarded packet seen before.

Figure 22 shows the results with 64-byte Ethernet frames. Note that in this case the router cannot forward near the full line speed traffic, as it would mean approximately 148809 frames/second (considering the Ethernet gap and preamble).

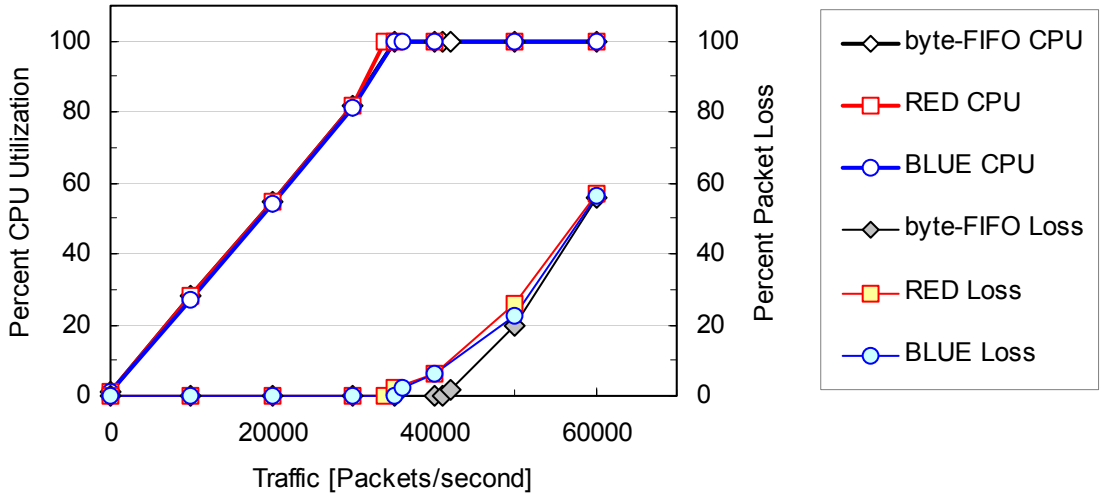

Figure 22 – Forwarding Performance with 64-byte frames

The CPU usage was expected to be higher in this case, as most of the work done at the forwarding – interrupt handling, IP header checks, routing decision, queuing – is proportional to the number of packets. The three queuing mechanisms performed similarly, with the byte-FIFO being a little faster than the others, and BLUE being very slightly faster than RED. The points, where the queues first reach 100% CPU utilization, and where the first packet losses occur have been measured out with 1000 Packets/second accuracy, and are shown in Table 2.

Queue	Packets/second	
	First reaching 100% CPU utilization, but no packet loss	First packet losses occur
byte-FIFO	35000	42000
RED	34000	35000
BLUE	35000	36000

Table 2 – Maximum forwarding speed of the compared queues

There was a third set of measurements performed with 512-byte Ethernet frames, but its results are so similar to the 1518-byte case that they are not presented here in details. In short, all queues handled the 100% line rate traffic (23496 frames/second) with 50%

CPU usage and without packet loss. The CPU usage has decreased the same way at the full link utilization, and for same Packets/second values it was approximately the same as in the 1518-byte case (for example 17% for 6000 Packets/second).

It is interesting to note that longer than zero length queues could be observed only when running at full line speed in the 512-byte or 1518-byte measurements. Otherwise, the transmit path was not a bottleneck, there were no output queue drops even in the 64-byte measurements with packet loss.

The conclusion is that the implemented BLUE module is at least as effective in terms of packet queuing and administration as the RED implementation is, and is capable to handle close the traffic that a simple FIFO queue is. However, there are other serious bottlenecks in the software router structure – probably the interrupt handling of the network interfaces – that prevent us to more precisely measure the situation in a default setup.

5.3 Traffic Measurements

For the traffic measurements, Linux computers were used to generate TCP traffic. The measurement network is shown on Figure 23. The links between the machines and towards the Ethernet switch were forced to full duplex 10 and 100 Mbit/second operation to formulate two bottlenecks at the interfaces of the router machine BLUE, where the measured AQM scheme can be set up.

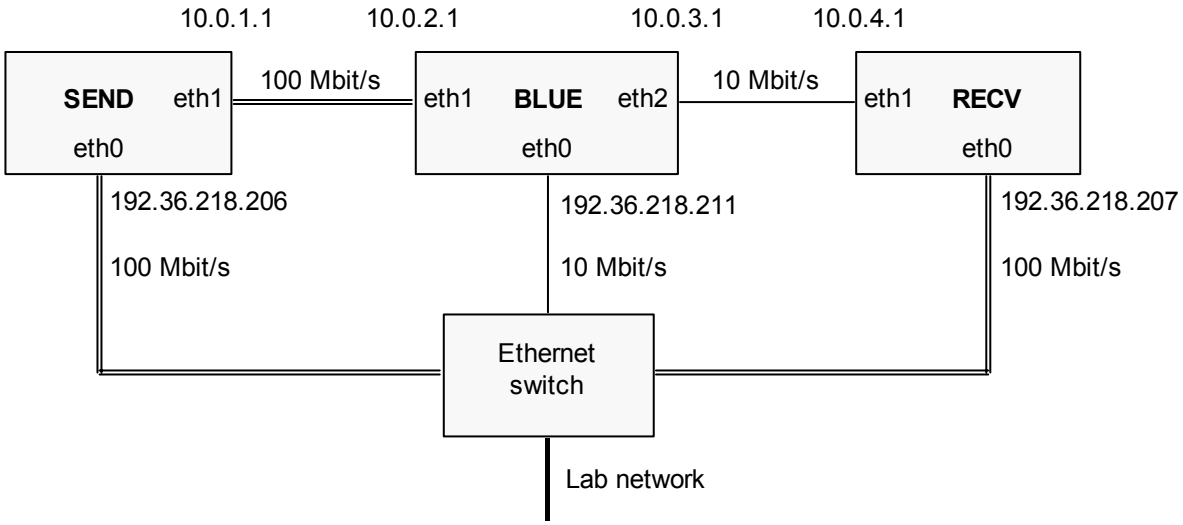

Figure 23 – The traffic measurements environment

All three machines can be used to generate traffic, and as the interfaces of the machines use different IP addresses, it is possible to direct the traffic according to the two measurement scenarios shown on Figure 24 and Figure 25.

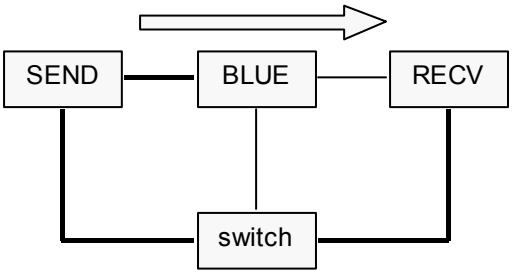

Figure 24 – Scenario 1 – AQM applied to a bottleneck link

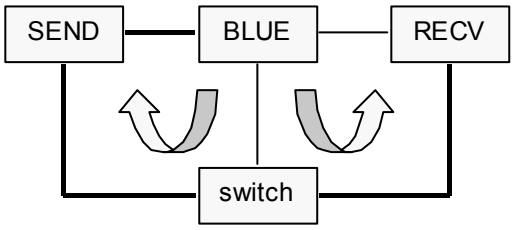

Figure 25 – Scenario 2 – ECN and non-ECN capable traffic sharing a link

Scenario 1 is the basic measurement setup, where the traffic is generated between SEND and RECV and sent through the eth2 bottleneck of BLUE.

Scenario 2 is used to perform measurements where non-ECN and ECN capable traffic share a single link. In this case the traffic is sent from BLUE to SEND and RECV through the eth0 bottleneck of BLUE. The easiest way to control the ECN-capability in Linux is the */proc/sys/net/ipv4/tcp_ecn* sysctl, which applies to all newly created connections and has a system-wide context. Hence, one of the two receivers runs with ECN disabled in the measurements in this scenario (and thus the new connections do not negotiate to use ECN).

It is not suspected that the bottleneck router and the traffic sender machine being the same would significantly bias the measurements in this scenario. The work of the machine BLUE can be even easier, as the traffic must only be sent, and not received (as compared to the forwarding case). Note also that only approximately 10 Mbit/second traffic was generated.

Netperf [36] sessions were used to generate traffic. Packet loss statistics were gathered on machine BLUE with the *tc* tool, directly from the AQM module used.

5.3.1 High bias against non-ECN flows

When experimenting with the measurements it got obvious that the naive approach – using the P_m ECN-marking probability also for packet dropping from non-ECN flows – works well only with light congestion. Using more then a few competing flows P_m gets so high, that the non-ECN flows experiencing the packet drops are subject to starvation.

To reproduce the problem in a controlled situation, the throughput of a TCP flow as a function of the experienced rate of packet loss or ECN marking was measured. The measurements were performed in Scenario 1 (Figure 24). The fixed rate packet marking/drop was generated using the initial P_m setting feature of the BLUE queue using commands like this:

```
blue:~# tc qdisc replace dev eth2 root blue limit 1MB init 0.02 inc 0 dec 0
ecn
```

Because of the *increase* = *decrease* = *0* parameters, the marking probability is fixed during the whole measurement (2% in this example). The 1 MB maximum Q_{len} is set to surely avoid tail drops. Non-ECN traffic was achieved with disabling ECN in RECV.

At every P_m value 10 tries of 30-second Netperf throughput measurements were performed. Figure 26 shows the results of the successful tries (with $P_m \geq 0.6$ the measurements often failed due to timeouts) and their average for the ECN and non-ECN case. Note that the maximum result in the measurements is only 9.41 Mbit/second, as Netperf reports the TCP payload throughput.

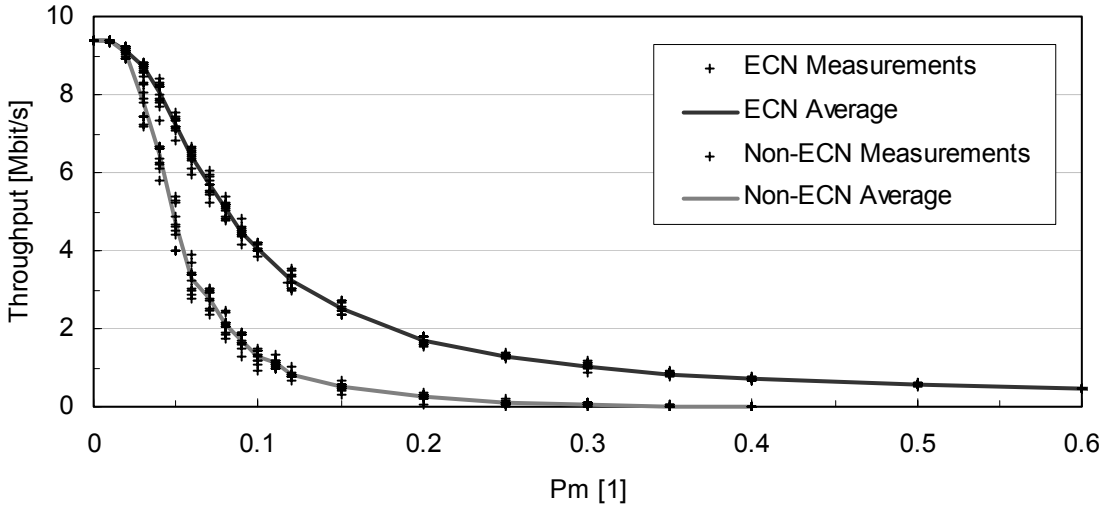

Figure 26 – Throughput of ECN and non-ECN flows as a function of P_m

To more emphasize the bias, Table 3 shows the ratio of the throughput of the ECN and non-ECN flows for selected P_m values. Note that with $P_m \geq 4\%$ the bias is definitely noticeable, and above a few percent it rapidly gets fatal for the non-ECN flows.

P_m	0%	1%	2%	3%	4%	5%	10%	20%	30%	40%
Ratio	1	1	1.006	1.112	1.249	1.543	3.2	6.47	18.37	143.4

Table 3 – Ratio of the throughput of ECN and non-ECN flows experiencing the same probability of packet marking or packet loss

In today's Internet it is not uncommon to happen that 10% packet drop is needed at a highly congested bottleneck link to cope with the congestion in the busy hours. Considering a future world, where most of the TCPs would be ECN-capable, approximately 30% packet marking probability would be needed to hold back the TCP senders at about the same sending rate than the 10% packet drop did.

An old-timer, not ECN-capable TCP would have hard times in that situation if faced to a simple $P_{drop} \leftarrow P_m$ mapping, effecting in an approximately 18–fold bias towards ECN at that point.

One of the reasons behind the high bias is the Linux ECN implementation not implementing the ECN timeouts and waiting for real packet loss to decrease CWND below 2 MSS. This could be solved with a function that maps the internal P_m to a (P_m, P_{drop}) pair for ECN and single P_{drop} for non-ECN considering this characteristic.

To suggest a fair mapping is left for future work, as it would need very thorough measurements and simulations in multi-gateway scenarios and with at least a few longer delay links also to avoid the too short delays distorting the picture. Note that thanks to small CWND during the measurements (it is small because of the packet loss), the Fast Retransmit scheme often fails to work, and as the RTO has a lower limit of 200 milliseconds on Linux[8], it expires very lately.

5.3.2 Comparing RED and BLUE

To compare RED and BLUE, the setup according to measurement scenario 1 (Figure 24) was used. ECN-capable TCP traffic was generated using Netperf sessions.

To achieve a fixed amount of parallel sessions, a wrapper shell script was used, which always restarts the Netperf command with the desired parameters – in this measurement these were to run a 30 second TCP throughput test. Starting a fixed amount of such shell scripts, we do have the same amount of – at least trying – Netperf sessions. After starting the scripts with 1-second pacing, the measurement data gathering started at the 100th second after starting the first script, and lasted for 100 seconds.

The queues were setup with the following parameters:

```
BLUE:  blue:~# tc qdisc replace dev eth2 root blue limit 50kB ecn
RED:   blue:~# tc qdisc replace dev eth2 root red limit 50kB min 8kB max 25kB
               avpkt 1000 burst 50 probability 0.1 ecn
RED2:  blue:~# tc qdisc replace dev eth2 root red limit 50kB min 8kB max 25kB
               avpkt 1000 burst 500 probability 0.6 ecn
```

BLUE is run mostly with its default parameters. The first RED queue is run with parameters according to the ordinary suggestions. As the variation of BLUE that reacts also to the Q_{len} exceeding $Q_{max}/2$ is used, RED's th_{max} is set similarly.

[8] This lower limit is needed to seamlessly interoperate with TCP/IP stacks that fully utilize the allowed 200 milliseconds for the delayed ACKs.

The second RED queue demonstrates a RED queue tuned to handle high congestion. Note the much longer EWMA memory (*burst*), and the higher P_{max} probability.

Figure 27 shows the results: the packet loss (early + tail drops) as a function of the number of Netperf sessions.

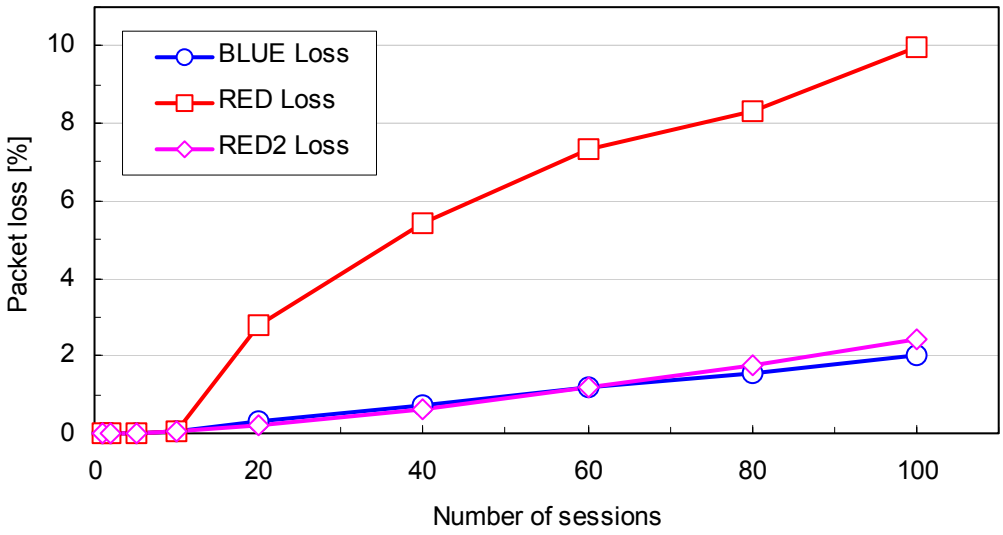

Figure 27 – Measured packet loss

To estimate the severity of the measured congestion, P_m reported by BLUE can be a possible guideline. It stabilized around 0.66 with 100 sessions. According to measurements in Section 5.3.1, it has approximately the same congestion control force, as 15% packet loss.

It is interesting to note, that BLUE practically totally avoided the tail drops, as they first occurred only when running 100 sessions, and even then they measured only 0.036% of all the forwarded packets. All the other drops were *early* drops, as such packets with the ECT bit not set.

6 Conclusions and Future Work

This work demonstrated that the basic idea of an AQM scheme mostly decoupled from the queue length such us BLUE is viable, and should be developed further.

One of the biggest advantages of BLUE is that it works well in a wide range of situations up to very high levels of congestion without manual fine-tuning for any special case.

Of course, being such a new research project, BLUE still could be improved in many areas. It could have a mechanism to avoid the P_m remaining high for a long time on a link that has gone suddenly idle. Its P_m adjustment algorithm could be also improved from the simplest linear increase/decrease. These all are exciting areas for future work.

One of the nice surprises during the experiments was that RED – even if it can be empirically proved that a queue-length-only based AQM is not adequate for many situations – can be tuned to handle severe congestion. Unfortunately, (plain) RED is generally not that scalable.

An unexpected (at least in terms of its seriousness) problem has been disclosed with ECN, namely the implementations not strictly following the standard could break the fairness between ECN-capable and non-ECN-capable sessions, an issue maybe needing special handling at the routers.

The Linux environment demonstrated its power for rapid development and high performance, although the hardware limitations prevent building high-end routers from commodity PCs. Note however, that with specialized network interfaces such as Gigabit Ethernet adapters focusing on saving on interrupt handling it can be possible to build very flexible access routers.

As direct continuation of the work, utilizing the ready module there could be other measurements done, with non-Linux TCP implementations, and also with non-TCP congestion control – streaming media and WAP just to name a few.

Further studying the disclosed bias against non-ECN flows could be also an interesting area – and also important when ECN gets widely deployed.

Acknowledgements

First of all I would like to thank to my industrial consultants Imre Juhász and István Cselényi and my supervisor Ferenc Baumann for their guiding, help, and providing the conditions to fulfill this work. I would like to special thank also to Telia Prosoft AB and Telia Research AB for sponsoring to carry out the work in Farsta, Sweden.

I would like to thank to Szabolcs Daróczy and Csaba Füzesi for the comments on my work and to all my colleagues in the Farsta High Performance Networks Laboratory – namely Péter Botka, István Csáki, Ede Zoltán Horváth, Péter Váry and Norbert Végh – for the work atmosphere and their support.

I am also thankful to Zsolt Turányi for the comments on my views of ECN, Csaba Tóth for the comments and discussion on the Linux Traffic Control framework, and Krisztián Kovács for information regarding the current TCP developments in the *BSD world.

I would like to acknowledge the work done by the operators of the Internet sites I used most often during this work. They are Attila Vonyó and Károly Dévényi for the Web-based Hungarian-English dictionary, the operators of the search engine at Google.com, the citation indexer at Citeseer.nj.nec.com and the Linux kernel source browser at Lxr.linux.no.

References

[1] V. Jacobson. *Congestion Avoidance and Control*
http://citeseer.nj.nec.com/jacobson88congestion.html

[2] *Hobbes' Internet Timeline*
http://www.zakon.org/robert/internet/timeline/

[3] B. Braden, S. Floyd, V. Jacobson, K. Ramakrishnan, and others. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. RFC2309, April 1998.

[4] S. Floyd, V. Jacobson. *Random Early Detection gateways for Congestion Avoidance.* IEEE/ACM Transactions on Networking, V.1 N.4, August 1993, p. 397-413.
http://www.aciri.org/floyd/papers/red/red.html

[5] C. V. Hollot, V. Misra, D. Towlsey, W. Gong. *A Control Theoretic Analysis of RED.* UMass CMPSCI Technical Report 00-41.
ftp://gaia.cs.umass.edu/pub/MisraInfocom01-RED-Control.pdf

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss. *An Architecture for Differentiated Service.* RFC2475, December 1998.

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

[8] K. Ramakrishnan, S. Floyd, D. Black. *The Addition of Explicit Congestion Notification (ECN) to IP.* Internet draft intended to supercede RFC2481.
http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-ecn-03.txt

[9] W. Feng, D. Kandlur, D. Saha, K. Shin. Blue: *A New Class of Active Queue Management Algorithms*. U. Michigan CSE-TR-387-99, April 1999.
http://www.thefengs.com/wuchang/blue/CSE-TR-387-99.pdf

[10] J. Postel. *Internet Protocol.* RFC791, STD5, September 1981.

[11] K. Nichols, S. Blake, F. Baker, D. Black. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers.* RFC2474, December 1998.

[12] S. Deering, R. Hinden. *Internet Protocol, Version 6 (IPv6).* RFC2460. December 1998.

[13] J. Postel. *Transmission Control Protocol. RFC793, STD7,* September 1981.

[14] R. Braden, RFC Editor. *Requirements for Internet Hosts -- Communication Layers.* RFC1122, STD3, October 1989.

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

[16] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, ISBN 0-201-63346-9, December 1994.

[17] G. Wright, W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, ISBN 0-201-63354-X, January 1995.

[18] *Summary of The Linux v2.4.3 TCP Implementation.* Linux kernel, file net/ipv4/tcp_input.c, starting at line 1056 *http://lxr.linux.no/source/net/ipv4/tcp_input.c#L1056*

[19] J. Hoe. *Startup Dynamics of TCP's Congestion Control and Avoidance Schemes.* Master's Thesis, MIT, 1995. *http://www.psc.edu/networking/rate-halving/*

[20] S. Floyd, T. Henderson. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC2582, April 1999.

[21] M. Mathis, J. Mahdavi, S. Floyd, A. Romanov. *TCP Selective Acknowledgment Options.* RFC2018, October 1996.

[22] K. Fall, S. Floyd. *Simulation-based Comparisons of Tahoe, Reno, and SACK TCP.* Computer Communication Review, V. 26 N. 3, pp. 5-21, July 1996. *http://www.aciri.org/floyd/papers/sacks.pdf*

[23] M. Mathis, J. Mahdavi. *Forward Acknowledgement: Refining TCP Congestion Control.* SIGCOMM 96, August 1996.

[24] V. Jacobson, R. Braden, D. Borman. *TCP Extensions for High Performance.* RFC1323, May 1992.

[25] V. Paxson. *End-to-End Internet Packet Dynamics.* Proceedings of SIGCOMM ‘97, September 1997. *http://citeseer.nj.nec.com/paxson97endtoend.html*

[26] L. S. Brakmo, S. W. O'Malley, L. L. Peterson. *A TCP Vegas: New Techniques for Congestion Detection and Avoidance.* Proceedings of SIGCOMM'94, pp.24-35, August 1994.

[27] S. Floyd. *TCP and Explicit Congestion Notification.* ACM Computer Communication Review, V. 24 N. 5, October 1994, p. 10-23. *http://www.aciri.org/floyd/papers/tcp_ecn.4.pdf*

[28] *ECN timeout support in Linux TCP.* Discussion on the Linux Netdev Mailing List. April 2001. *http://oss.sgi.com/projects/netdev/mail/netdev/msg02708.html*

[29] K. Cho. *ALTQ: Alternate Queuing for BSD UNIX, Version 3.0.* December 2000. *http://www.csl.sony.co.jp/person/kjc/programs.html#ALTQ*

[30] S. Floyd. *RED: Discussions of Byte and Packet Modes.* March 1997. *http://www.aciri.org/floyd/REDaveraging.txt*

[31] S. Floyd, V. Jacobson. *Link-sharing and Resource Management Models for Packet Networks*. IEEE/ACM Transactions on Networking, Vol. 3 No. 4, pp. 365-386, August 1995. *http://www.aciri.org/floyd/cbq.html*

[32] W. Almesberger. *Linux Traffic Control – Implementation Overview.* November 1998. *ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz*

[33] G. Dhandapani, A. Sundaresan. *Netlink Sockets – Overview.* October 1999. *http://qos.ittc.ukans.edu/netlink/html/index.html*

[34] B. Hubert et al. *Linux 2.4 Advanced Routing & Traffic Control.* March 2001. *http://www.ds9a.nl/2.4Routing/*

[35] I. Bartók. *Implementation and Evaluation of the BLUE Active Queue Management Algorithm.* May 2001. *http://www.sch.bme.hu/~bartoki/projects/thesis/*

[36] R. Jones et al. *The Public Netperf Homepage.* *http://www.netperf.org/*

[37] J. H. Salim. *Tcpdump patch to print ECN info.* May 1998. *http://www.aciri.org/floyd/ecn/tcpdump.txt*

Abbreviations

ACK	Acknowledgment
AF	Assured Forwarding
ARP	Address Resolution Protocol
AQM	Active Queue Management
ATM	Asynchronous Transfer Mode
BE	Best Effort
BECN	Backward Explicit Congestion Notification
BSD	Berkeley Software Distribution
CBQ	Class Based Queuing
CE	Congestion Experienced
CPU	Central Processing Unit
CWND	Congestion Window
DoS	Denial of Service
DS	Differentiated Services
DSCP	Differentiated Services CodePoint
DSP	Digital Signal Processor
dupack	Duplicated Acknowledgment
ECN	Explicit Congestion Notification
ECT	ECN Capable Transport
EF	Expedited Forwarding
EWMA	Exponentially Weighted Moving Average
FACK	Forward Acknowledgment
FECN	Forward Explicit Congestion Notification
FIFO	First In First Out
GNU	GNU's Not Unix
HTTP	Hypertext Transfer Control Protocol
IETF	Internet Engineering Task Force
IHL	Internet Header Length
ISP	Internet Service Provider
IP	Internet Protocol
IPv6	Internet Protocol version 6
ISN	Initial Sequence Number

LAN	Local Area Network
LAPB	Link Access Procedure, Balanced
MSS	Maximum Segment Size
NAT	Network Address Translation
NTP	Network Time Protocol
PC	Personal Computer
PSTN	Public Switched Telephone Network
qdisc	Queuing Discipline
RED	Random Early Detection
RFC	Request For Comments
RTP	Real-time Transport Protocol
RTT	Round Trip Time
RTTM	Round Trip Time Measurement
SACK	Selective Acknowledgment
ssthresh	Slow Start Threshold
TC	Traffic Control
TCP	Transmission Control Protocol
ToS	Type of Service
UDP	User Datagram Protocol
UTP	Unshielded Twisted Pair
VoIP	Voice over IP
WRR	Weighted Round Robin

Appendix A – Adding ECN support to tcpdump

This modification is mostly based on [37]. My additions are:

- Display also the ECN CWR flag in TCP packets.
- This patch applies cleanly to the Debian tcpdump-3.4a6 package source.

```
diff -u --recursive --new-file tcpdump-3.4a6/print-ip.c tcpdump-3.4a6-
ecn/print-ip.c
--- tcpdump-3.4a6/print-ip.c    Sat May 26 21:01:44 2001
+++ tcpdump-3.4a6-ecn/print-ip.c        Fri Apr 27 21:02:32 2001
@@ -493,8 +493,15 @@
        } else if (off & IP_DF)
                (void)printf(" (DF)");

-       if (ip->ip_tos)
+       if (ip->ip_tos) {
                (void)printf(" [tos 0x%x]", (int)ip->ip_tos);
+
+               /* ECN bits */
+               if (ip->ip_tos &0x01)
+                       (void)printf(" [CE] ");
+               if (ip->ip_tos &0x02)
+                       (void)printf(" [ECT] ");
+       }
        if (ip->ip_ttl <= 1)
                (void)printf(" [ttl %d]", (int)ip->ip_ttl);

diff -u --recursive --new-file tcpdump-3.4a6/print-tcp.c tcpdump-3.4a6-
ecn/print-tcp.c
--- tcpdump-3.4a6/print-tcp.c   Sat May 26 21:01:44 2001
+++ tcpdump-3.4a6-ecn/print-tcp.c       Fri May 25 18:40:27 2001
@@ -75,6 +75,9 @@
 #define TCPOPT_CCECHO          13      /* T/TCP CC options (rfc1644) */
 #endif

+#define ECE_ON                 0x40    /* ECN CE Notify */
+#define CWR_ON                 0x80    /* ECN CE Notify */
+
 struct tha {
        struct in_addr src;
        struct in_addr dst;
@@ -146,6 +149,12 @@
                        putchar('P');
        } else
                putchar('.');
+
+       if (flags & ECE_ON)
+               printf(" [ECE]");
+
+       if (flags & CWR_ON)
+               printf(" [CWR]");

        if (!Sflag && (flags & TH_ACK)) {
                register struct tcp_seq_hash *th;
```

Appendix B – Hardware configuration of the computers used for the measurements

Machine	BLUE	SEND, RECV
CPU	466 MHz Intel Pentium3	350 MHz AMD K6-3D
Cache	L1 Instruction 16 kB	L1 Instruction 32 kB
	L1 Data 16 kB	L1 Data 32 kB
	L2 256 kB	
Motherboard Chipset	Intel 440 BX	ALi M1541
RAM	128 MB SDRAM	32 MB SDRAM
Eth0	3Com PCI 3c905B Cyclone	Intel 82557 PCI (Ethernet Pro 100)
Eth1	3Com PCI 3c905C Tornado	Intel 82557 PCI (Ethernet Pro 100)
Eth2	3Com PCI 3c905C Tornado	–