TCP MaxNet
Background, Algorithms, Implementation and Experiments

Thesis by

Martin Suchara

In Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Science

California Institute of Technology
Pasadena, California

2006

Acknowledgement

I would like to acknowledge the support and valuable advice received from my research advisor Professor Steven Low. Also I would like to acknowledge Lachlan Andrew, Ryan Witt, and Bartek Wydrowski for their help with development of the many underlying theories, implementation of the protocol in the Linux kernel, and last but not least, the advice they provided.

Abstract

We present the first implementation of TCP MaxNet, a congestion control algorithm which uses multi-bit explicit congestion signal. While the theoretical properties of the MaxNet framework have been studied in depth, we have been lacking an implementation of the protocol that would confirm or refute what theory predicts; namely, stability, quick speed of convergence, max-min fairness, short router queues and low latency. The protocol consists of two components: an AQM algorithm and a source algorithm. The AQM algorithm resides in routers and calculates the congestion price. The source algorithm reacts to the calculated price and adjusts the sending rate accordingly. We provide pseudocode of both algorithms, and implement the protocol in the Linux kernel by modifying the TCP/IP stack. Implementation related issues, such as fractional calculations and evaluation of exponentials in the kernel are described. We show that our implementation, which uses a 23-bit TCP Option to encode and transmit the price, scales from 32 bits/sec to 1 Peta-bit/sec rates with sufficient accuracy. In the initial phase of data transmissions, we use a variant of the QuickStart algorithm, which allows the sender to start sending almost immediately at the available capacity, and prevents queue buildup. Using state of the art WAN-in-Lab equipment, we evaluate the performance of TCP MaxNet and confirm favorable properties of the protocol. We observe that within a few RTTs the sending rates converge to their max-min fair shares as dictated by the price. We show that the latency is much lower than that of the other TCP protocols, and the protocol operates with nearly empty router queues. Finally, since the protocol does not treat loss as congestion signal, it has superior performance under loss.

Contents

1 Introduction

The role of transmission control protocols (TCPs) [1] in computer networks is twofold. First, they are responsible for reliable and timely transfers of data in an environment where data may be corrupted or lost and packets may be received out of order or even duplicated. Second, they manage sharing of network resources to avoid congestion. Good congestion control provides high efficiency, fair sharing among competing users, and stable rates that reduce delay and jitter. While mechanisms that ensure reliable data transfers between the sender and receiver are well known and easy to implement, the distributed character of the congestion control algorithm, which is naturally imposed by the distributed organization of modern computer networks, makes congestion control an especially challenging problem.

1.1 Problems of the Current Congestion Control Algorithms

Congestion control algorithms in TCPs adjust the source rates based on a congestion signal obtained from the network. The congestion signal is obtained either passively by measuring the loss rate or delay, which is known to correlate with the level of congestion, or is calculated actively by an AQM (Active Queue Management). If the congestion level increases, the strength of the congestion signal increases as well. On the other hand, when the network is underutilized, the signal decreases. We can further distinguish between a binary and multi-bit congestion signal. A wealth of TCP protocols using passive, active, binary and multi-bit types of congestion level signaling has been suggested, analyzed and implemented.

The most widely used congestion control protocol today is TCP Reno [2] and its improved variant New Reno [3]. The algorithm is an improved version of TCP Tahoe, which dates to 1988 when it was suggested as a response to a series of congestion collapses in the Internet. Both algorithms are simple and

use loss of a packet as a binary congestion signal. The heart of the congestion control algorithm is AIMD (Additive Increase Multiplicative Decrease). The sender's congestion window[1] w is first increased exponentially in the slow start phase, and then adjusted in the congestion avoidance phase as follows:

$$on\ ACK:\ w = w + \frac{1}{w}, \tag{1}$$

$$on\ loss: w = \frac{1}{2}w. \tag{2}$$

TCP Reno has performed remarkably well considering the explosive growth of the Internet in the last two decades.

However, it is well known that the performance of the protocol degrades as the bandwidth delay product increases [4] [5] [6]. Equation (2) implies that in "equilibrium" the window of the sender oscillates between the maximal value it attains and one half of that value. In order to achieve full utilization, Reno requires buffer sizes equal to (at least) the bandwidth delay product of the links, in which case the buffer occupancy oscillates between 100% just after the packet loss and 0% when the window assumes its average value. As bandwidth and delay increase, the buffer sizes and the associated delay become prohibitively expensive.

Another issue is stability. From (1) and (2) the window of the sender increases by approximately $1/T$ packets per unit time and decreases by

$$\frac{4}{3}w\frac{1}{2}q\frac{w}{T} \tag{3}$$

packets per unit time, where q is the probability of packet loss, T is the round trip time, and w is the window size corresponding to the average sender's rate. Notice that $\frac{4}{3}w$ represents the maximum size attained by the congestion window, and $\frac{w}{T}$ is the number of packets sent per unit time. Therefore, the

[1]The details of operation and terminology used in connection with TCP protocols are provided in chapter 2.

change of the sender's window can be expressed as

$$\dot{v} = \frac{1}{T} - \frac{2}{3}\frac{qw^2}{T}, \tag{4}$$

and in equilibrium we have $\dot{v} = 0$. We have the following relation between the equilibrium loss probability q^* and the equilibrium window w^*, described in [4] [5] [6]:

$$q^* = \frac{3}{2}\frac{1}{w^{*2}}. \tag{5}$$

Consider, for example, 1 Gbps link with 50 ms RTT, a link we use in our experiments in chapter 5, and assuming the packet size is 1500 bytes, we calculate that the average sender's window is 4,167 packets, the peak window size in equilibrium is 5,556 packets, and the window is cut to 2,778 packets after each loss. As a result, it takes 139 seconds for the sender to recover from a single loss. In practice, the performance of the protocol would be exacerbated by loss caused by new flows probing the network and/or non-congestion based loss, making it nearly impossible to sustain full utilization of the network.

A number of extensions of the original packet loss based scheme, such as TCP BIC [7], HS [8], H [9], Scalable [10] and Westwood [11], have been suggested. These protocols often dramatically improve the performance of TCP Reno. However, some fundamental problems remain. Oscillations are a natural consequence of only one-bit congestion signal, and thus these proposals do not eliminate the need of large buffers. For example, TCP BIC achieves higher average throughput in high delay bandwidth product networks by increasing the sender's window more aggressively[2] after loss, but this results in longer router queues. TCP Vegas [12], and TCP FAST [13] [14] use a new approach and treat delay as an indicator of congestion. The biggest advantage

[2]TCP BIC halves its window after loss and then in each subsequent RTT increases its window by half of the difference of the current window and the window before loss.

of these delay-based proposals is that they are provably stable [15] [6] [14] and achieve excellent throughput and responsiveness. However, due to their delay based design, both protocols still require nonzero router queues, and thus introduce additional delay that grows with load.

Explicit congestion signal protocols provide yet another interesting method of congestion control. They calculate the congestion level actively in the routers, and use additional fields in the packet header to communicate the congestion level back to the sender. ECN [16], an extension of TCP Reno, marks a single bit in the header to indicate congestion. The higher the rate of packet marking, the higher the congestion level is. Even more interesting are XCP [17] and RCP [18], new protocols that use multi-bit explicit congestion signaling. XCP uses the routers to communicate the sender's window in the packet headers, while RCP communicates directly the sending rate. This multi-bit method of congestion signaling allows the protocols to improve the stability of the flow control and results in very short router queues, decreasing latency of the flows. However, recent research showed that even these implementations are not flawless.

XCP was shown to only achieve constrained max-min fairness, and one of its flows may be assigned an arbitrarily small rate [19]. Moreover, the protocol's stability has only been shown for a single link scenario shared by flows with identical RTTs. RCP faces another problem. It calculates d_0 [18], an estimate of the weighted average RTT of the flows, and performs rate updates with dynamics equal to this value. This has negative effect on dynamics of the system in an environment where RTTs of the flows vary. For example [20], in a router where 90% of the traffic is local LAN traffic with 1 ms RTT, and the remaining 10% of the bandwidth is used by one WAN flow with 500 ms RTT, d_0 is 50 ms. As a result, the LAN traffic is forced to operate with slow WAN dynamics. Another issue that pertains to both XCP and RCP is security. The routers trust the sources to honestly

advertise their RTTs. The problem is that a malicious source could advertise a very large RTT, and arbitrarily slow down the dynamics of the routers.

TCP MaxNet is a new congestion protocol that attempts to solve the aforementioned challenges. Namely, MaxNet aims to achieve stability, zero router queues, max-min fairness and fast dynamic properties in a range of environments.

1.2 TCP MaxNet

In this document we report on the first successful implementation of TCP MaxNet. The protocol, first introduced in [21] and [22] by Wydrowski, Andrew and Zukerman, uses a multi-bit congestion signal, which is actively calculated by routers. The main difference between MaxNet and other protocols, with the exception of XCP and RCP, is that the senders in MaxNet adjust their sending rate in response to the congestion signal from the most severely congested bottleneck on the end-to-end path they use. Other protocols use the sum of these signals. The framework provided by MaxNet has been shown to have many advantages. MaxNet is stable in networks of arbitrary sizes, speeds and topologies [21], and provides max-min fairness [22]; i.e., increase of a transmission rate of a sender does not result in the decrease of the transmission rate of a competing user that has equal or lower rate. Moreover, MaxNet has fast convergence properties [23]. Since the MaxNet sender decreases its sending rate as the price calculated by routers increases, proper selection of the rate control algorithms allows operation of the protocol with empty router queues, which is something other protocols that use passive congestion signal cannot achieve. Finally, since the protocol does not need to treat loss as a congestion signal, we show that it is vastly superior to the other popular protocols in the presence of loss.

In chapter 2 we explain the fundamentals of transmission control protocols. Chapter 3 provides a detailed description of the MaxNet algorithm,

and chapter 4 describes our implementation in the Linux kernel. In chapter 5, we present the results of our experiments. We summarize our results in chapter 7 and provide directions for further research in chapter 6.

2 Basic Design and Functionality of Transmission Control Protocols

Rather than providing exhaustive descriptions and specifications of the algorithms, which can be found e.g. in [1], we focus on describing the components of the protocol that either play an important role in our performance considerations, or that need to be modified. Readers with knowledge of the TCP algorithm may safely skip this chapter.

Since transmitted data may be corrupted or lost, providing a framework for reliable and timely transfer of data is one of the most important goals of TCPs. To achieve this, TCPs assign a 32-bit sequence number to each octet of data that is being transmitted. The data is then split into packets (typically of sizes 1522 bytes or less), and transmitted over the network to the destination. Each packet contains a TCP header that includes the sequence number of the first octet of data in the packet. The format of a TCP header is depicted in Fig. 1. When the receiver receives the packet, it acknowledges its receipt by sending an acknowledgement (ACK) to the sender. The ACK contains the sequence number of the first octet of data that has not been received. Whenever the sender transmits a packet to the network, it places its copy on a retransmission queue, and starts a timer. If the data is not acknowledged before the timer times out, the data is assumed to be lost or corrupted, and it is retransmitted. If, on the other hand, the packet is acknowledged, the data is removed from the queue. This scheme ensures that all data is delivered to the receiver's TCP, which then reassembles it and passes it to the user's application.

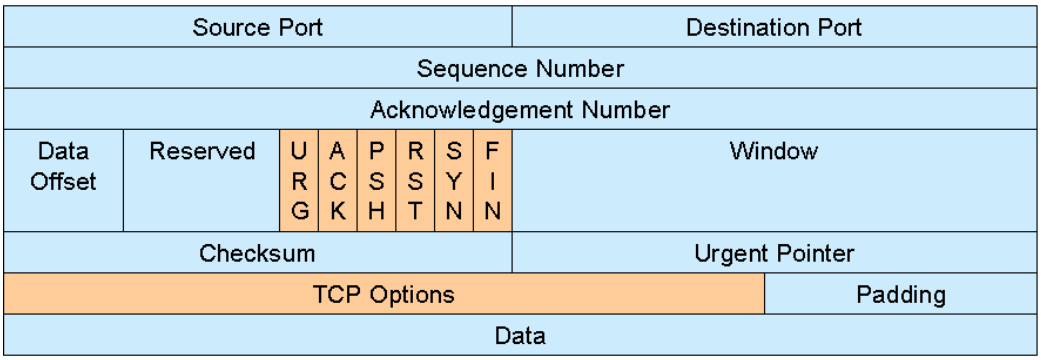


Figure 1: TCP header format. Sequence number identifies the first octet of data in the data part of the packet. Checksum is used to verify that the data has not been corrupted while transmitted. Window is the maximal number of packets that the sender of the packet is prepared to accept from the receiver.

To detect corruption of data, a checksum [24] algorithm is used. The sender calculates a 16-bit checksum for each packet, and places it in the checksum field of the TCP header. Upon receipt, the receiver validates the packet by recalculating the checksum of the packet. If the recalculated checksum does not match the checksum in the packet, the packet is rejected. The following algorithm is used to calculate the checksum. A 96-bit pseudo header containing the source address, the destination address, the protocol, and TCP length is prepended to the packet depicted in Fig. 1. The checksum field itself is substituted with zeros while the checksum is calculated. If the segment contains an odd number of octets, it is padded with one zero octet. Adjacent octets are paired so that they form 16-bit words. The checksum is calculated as a 1's complement sum[3] of the 16-bit words, and finally the 1's complement of the sum is placed in the checksum field of the TCP header. In our implementation, routers update price contained in the TCP header. Whenever the value changes, the checksum needs to be recalculated. An efficient way of performing an incremental update is described in section 4.2.

[3]One's complement addition is a binary addition where the carry-out bit is added to the least significant bit of the sum.

TCP options are used for negotiation between the sender and receiver. The first byte of each option contains the option type, the next byte encodes its length, and the next bytes are used to communicate multi-bit information to the receiver of the packet. Our implementation of MaxNet introduces a new TCP option which is used to communicate congestion price between the sender and the routers. Another example of a TCP option is SACK [25], an important and widely adopted extension of TCP Reno.

SACK, an extension of the TCP algorithm allowing selective acknowledgement of received segments of data [2], has been introduced because of poor performance of the standard algorithm in the presence of multiple packet losses. Fig. 2 shows a typical receiver queue when multiple packets were lost. The original TCP algorithm is only able to acknowledge the largest continuous block of data. Consequently, if some packet is missing, the sender must either wait for 1 RTT (round trip time) to learn that the next packet is missing, or retransmit all the data beginning with the first missing packet, which is inefficient. TCP SACK addresses this problem by allowing the sender to acknowledge a number of continuous blocks of data. TCP option is used to implement SACK. The SACK option consists of a one byte header field, one byte length field, followed by sequence numbers of the right and left edge of each of the acknowledged blocks. The length of the SACK option is $2 + 8n$ bytes where n is the number of blocks being acknowledged. Standard implementations usually allow selective acknowledgement of up to three blocks. Our implementation acknowledges at most two.

An important role of TCPs is to control the rate of data transmissions. The sender controls its rate by calculating a congestion window w, i.e., the maximum number of unacknowledged packets that can be outstanding in the network. Since RTT denotes the time it takes for a packet to be acknowledged, the sending rate x is

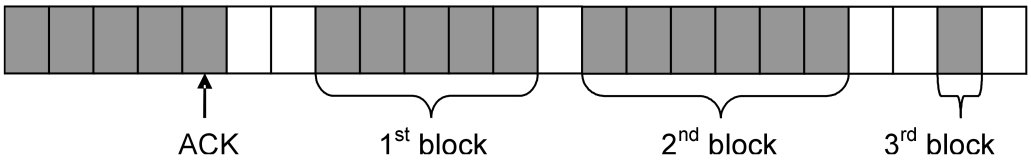


Figure 2: The shaded boxes represent packets that were received and the white boxes represent packets that were lost in transmission. Standard TCP may acknowledge only the largest continuous block of data, while TCP SACK notifies the sender that 3 more continuous blocks of data were received.

$$x = w/RTT. \tag{6}$$

The congestion window is typically calculated as a function of the congestion signal the sender receives. The main challenge for TCPs, including TCP MaxNet, is to calculate the window in each sender so that high utilization, stability, low loss, fair sharing of resources and low latency is achieved.

3 The Congestion Control Strategy of the MaxNet Algorithm

The MaxNet congestion control algorithm is comprised of two principal components - an AQM algorithm residing in the routers, and a rate control algorithm which is used on the sender side. This chapter describes both algorithms and their interaction.

Fig. 3 depicts the congestion signal feedback loop of MaxNet. For each link l routers periodically calculate price P_l which represents the most recent level of congestion. The prices increase with increasing loads of the links. The source adjusts its sending rate as a function of the maximum price on the end-to-end path. Section 3.1 describes how the congestion price is calculated. In addition, section 3.1 extends the basic MaxNet framework by describing how initial sending rates may be specified by routers. Section 3.2 provides

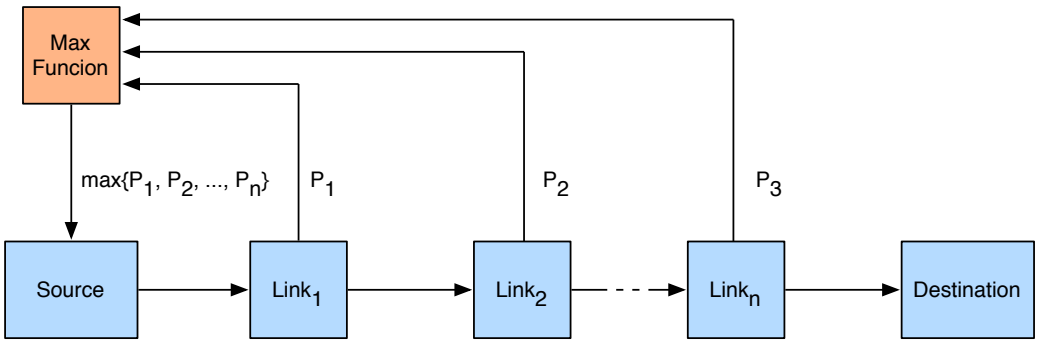


Figure 3: The congestion control feedback loop of MaxNet. The sender uses the maximum of the link prices P_1, P_2, ..., P_n to adjusts its congestion window.

the demand function of the source. Since the source only needs to learn the maximum of the prices on the end-to-end path, we argue in section 3.3 that a single 23 bit field in a packet suffices to signal the price with sufficient range and accuracy.

3.1 Router Algorithm

The role of the router code is two-fold. First, routers calculate congestion price for all links in order to control the sources. Second, routers use a variant of the QuickStart algorithm [26] to assign a fraction of the available free capacity to new flows joining the network.

3.1.1 Price Calculation

The congestion price of link l with capacity C is calculated by the AQM as:

$$\mathtt{p}_l(t+1) = p_l(t) + \frac{1}{C}(y_l(t) - \mu C), \tag{7}$$

where $y_l(t)$ is the amount of traffic which has been enqueued since the last price update and is destined to leave through link l, $\frac{1}{C}$ is a constant controlling the rate of convergence of the MaxNet algorithm [27], and μ is a constant. In steady state the price stabilizes and $y_l(t) = \mu C$. Choosing $\mu < 1$ results

1. On SYN arrival:

   $\texttt{QSrate} \leftarrow C(\mu + \epsilon) - \frac{\texttt{y}}{\texttt{dt}}$
   if $\texttt{QSrate} < \texttt{QSminrate}$ **then**
     $\texttt{QSrate} \leftarrow \texttt{QSminrate}$
   end if
   $\texttt{QSrateln} \leftarrow -ln\left(\frac{\texttt{QSrate}}{x_{max}}\right)$
   if $\texttt{QSrateln} > \texttt{pkt.QSrate}$ **then**
     $\texttt{pkt.QSrate} \leftarrow \texttt{QSrateln}$
   end if

2. Every `dt` seconds:

   $y \leftarrow \frac{y}{\texttt{dt}}$
   $\texttt{p} \leftarrow \texttt{p} + \left(\frac{\texttt{y}}{C} - \mu\right) \texttt{dt}$
   if $\texttt{p} < 0$ **then**
     $\texttt{p} \leftarrow 0$
   end if
   $\texttt{y} \leftarrow 0$

3. On packet arrival:

   $\texttt{y} \leftarrow \texttt{y} + \texttt{pkt.size}$
   if $\texttt{p} > \texttt{pkt.price}$ **then**
     $\texttt{pkt.price} \leftarrow \texttt{p}$
   end if

Variables and parameters:

`QSrate`	calculated QuickSrart rate
`QSminrate`	minimal rate assigned to new flows
μ	target utilization, typically ≈0.95
ϵ	new flow allocation overshot
`C`	link capacity
`y`	data arrival rate
`dt`	price update interval
`pkt.price`, `pkt.QSrate`	price and QuickStart rate advertised in packets

Figure 4: Pseudocode of the router algorithm

in less than 100% utilization of the link and prevents queue buildup. Our implementation of the price calculation is described in step 2 of Fig 5. To avoid interrupts, routers perform calculation (7) every dt seconds, where dt is typically smaller than the smallest expected RTT. The resulting calculation is efficient even at high speeds, as only one simple addition - increment of $y_l(t)$ - per packet is required. The resulting price p_l is saved in the price option of a packet that uses link l only if the price contained therein is lower than p_l.

3.1.2 QuickStart Rate Calculation

The router side implementation of the QuickStart algorithm is described in step 1 of Fig 5. The available capacity at link l is $C - \frac{y}{dt}$ where $\frac{y}{dt}$ is the capacity used up by the background traffic. The initial target rate $QSrate$ is given by $C(\mu + \epsilon) - \frac{y}{dt}$, where ϵ guarantees that the link will be slightly overutilized after the new flow joins. Since the available rate may be zero or even negative, we always assign at least $QSminrate$ to each new flow. $QSrate$ is encoded in a passing packet only if it is the first packet of a joining flow, i.e., SYN packet, and if the $QSrate$ contained in the packet is either uninitialized or larger than the calculated value.

3.2 Source Algorithm

A simplified pseudocode of the source algorithm is provided in Fig. 5. The goal of the source algorithm is to calculate new congestion window $estWnd$ based on the price and initial sending rate, which are obtained from incoming ACKs. The congestion window needs to be calculated so that all the favorable properties of the protocol we desire are achieved.

3.2.1 QuickStart Rate Control Mode

The source operates in two modes: QuickStart mode and price controlled mode. The sender enters the QuickStart mode at the beginning of the trans-

mission after the first ACK with the QuickStart rate is received, and immediately starts sending at this rate. The window size corresponding to this rate is saved in $QSWnd$. $QSWnd$ is exponentially increased over time. The consequent incoming ACKs contain prices which are saved in q, and the sender periodically calculates the sending rate based on this price. The window corresponding to this rate is saved in $PriceWnd$, but this value is not used yet to control the transmission rate. Only when the price-calculated rate is smaller than the QuickStart rate do we exit the QuickStart mode and enter the price controlled mode. The reason for using the QuickStart mode in the initial phase is twofold. First, it allows the sender to start transmissions at as high a sending rate as can be immediately accommodated. Second, entering directly in the price controlled mode is not convenient because the initial price obtained from the routers shortly after a flow joins is not close enough to the equilibrium. For example, after a single flow joins the network, the price converges to some value p. When a second flow joins, it receives p in the second packet. This price would result in too high a sending rate of the second flow because p does not take into account the additional traffic of the second flow. The following paragraphs describe the window calculation in the two modes in greater detail.

After arrival of the first ACK, the sender decodes the QuickStart rate. This is described in step 1 of the pseudocode. `QSfinished = false` indicates that we entered the QuickStart rate control mode. As long as we are in the QuickStart mode, $QSrate$ contains the target sending rate. Since the conditions in the network may change with time, e.g., a flow may leave the network, the $QSrate$ may become too small, and the price q may stop increasing. Since the price controlled mode needs to be reached, $QSrate$ is increased. It is increased by factor $rateinc$ every RTT whenever q is not increasing quickly enough. This is described in step 4 of the algorithm.

1. On first ACK arrival obtain QuickStart rate:
   $\texttt{QSrate} \leftarrow x_{max} \times exp\,(\texttt{-pkt.QSrate})$
   $\texttt{QSfinished} \leftarrow false$

2. On second ACK arrival estimate ξ:
   $\xi \leftarrow q\left(\frac{\alpha}{\texttt{baseRTT}} - \frac{1}{T}\right)$

3. Every `dt` seconds update ξ and calculate `estWnd`:
   $\xi_{new} \leftarrow \xi + \frac{\eta \times dt}{\texttt{baseRTT}^2}\left(\left(\frac{T \times \alpha}{\texttt{baseRTT}} - 1\right) \times q - T \times \xi\right)$
   if $\xi < q\left(\frac{\alpha}{\texttt{baseRTT}} - \frac{1}{T}\right) < \xi_{new}$ **or** $\xi_{new} < q\left(\frac{\alpha}{\texttt{baseRTT}} - \frac{1}{T}\right) < \xi$ **then**
     $\xi \leftarrow q\left(\frac{\alpha}{\texttt{baseRTT}} - \frac{1}{T}\right)$
   else
     $\xi \leftarrow \xi_{new}$
   end if

   $\texttt{PriceWnd} \leftarrow \texttt{baseRTT} \times x_{max} \times exp\left(\xi - \frac{q \times \alpha}{\texttt{baseRTT}}\right)$
   if `QSfinished` **then**
     $\texttt{estWnd} \leftarrow \texttt{PriceWnd}$
   else
     $\texttt{QSWnd} \leftarrow \texttt{QSrate} \times \texttt{baseRTT}$
     $\texttt{estWnd} \leftarrow \texttt{QSWnd}$
     if $\texttt{QSWnd} > \texttt{PriceWnd}$ **then**
       $\texttt{QSfinished} \leftarrow true$
     end if
   end if

4. Every `baseRTT` seconds:
   if $\texttt{QSWnd} < \texttt{cwnd}$ **then**
     $\texttt{deltaq} \leftarrow \frac{\texttt{q-lastq}}{\texttt{baseRTT}}$
     $\texttt{lastq} \leftarrow \texttt{q}$
     if $q = 0$ **or** $\texttt{deltaq} < \texttt{mindeltaq}$ **then**
       $\texttt{QSrate} \leftarrow \texttt{QSrate} \times \texttt{rateinc}$
     end if
   end if

Variables and parameters:

`cwnd`	current value of congestion window
`dt`	window update interval
`estWnd`	new calculated congestion window
$\texttt{x}_{max}$	maximal supported transmission rate
ξ	state variable used in window calculation
`q`	price received in the most recent packet
α	convergence parameter
η	convergence parameter
`T`	parameter that determines speed of convergence
`baseRTT`	minimum RTT measurement of the flow
`QSrate`	QuickStart rate obtained from first ACK
`QSWnd`	window corresponding to `QSrate`
`PriceWnd`	window corresponding to `q`
`QSfinished`	true after exit from QuickStart mode
`mindeltaq`	determines that price does not increase
`rateinc`	factor for rate increase

Figure 5: Pseudocode of the source algorithm

3.2.2 Price Rate Control Mode

The window in the price controlled mode is given by the following demand function:

$$PriceWnd = baseRTT \times x_{max} \times exp(\xi - \frac{q \times \alpha}{baseRTT}), \tag{8}$$

where ξ is a state variable. This rate control law ensures that the sender achieves the maximum rate x_{max} when $q = 0$, and the rate is inversely proportional to q. ξ counterbalances the dependence of the window on RTT so that multiple senders with different RTTs achieve the same throughput. ξ is calculated according to the following update rule [20]:

$$\xi = \xi + \frac{\eta \times dt}{\texttt{baseRTT}^2} \left(\left(\frac{T \times \alpha}{\texttt{baseRTT}} - 1 \right) \times q - T \times \xi \right). \tag{9}$$

The window calculation and update of ξ is described in step 3 of the algorithm. ξ is updated every dt seconds, which is typically much smaller than $baseRTT$. This ensures sufficient dynamics of the control system. Andrew et al. show in [20] that for fixed q the value of ξ converges to

$$\xi = q(\frac{\alpha}{baseRTT} - \frac{1}{T}). \tag{10}$$

In order to speed up the convergence of ξ in the initial phase, we initialize ξ according to (10) as soon as q is available. When dt is too large due to a delay of the call of the update rule (9), the value of ξ may be adjusted incorrectly because the control system overreacts. We solve this problem by not allowing ξ to exceed the value given by (10).

The algorithm presented here is especially attractive because the only per packet operation required is update of q. ξ and the congestion window $estWnd$ is updated at most every dt seconds, and the calculations do not severely burden the sender as its sending rate increases. Implementing an interrupt that performs step 3 every dt seconds is not required either. Since no new information is conveyed to the source unless a new packet is received, step 3 is performed only when a new packet is received and more than dt seconds have elapsed from the last update.

3.3 Price and QSrate Encoding

This section describes how price and $QSrate$ is encoded in the packets so that the MaxNet algorithm achieves sufficient range and accuracy of rate control. We start by analyzing the price encoding.

Since price is quantized, rates are also quantized. If the sender receives prices q_A and q_B, the corresponding congestion windows are $D(q_A)$ and $D(q_B)$ where $D(q)$ is the demand function in (8). We require that the increase of the rate from one quantization level to another is small, i.e., $D(q_A) < \alpha D(q_B)$

where α is a small constant. This requirement ensures that accurate rate control can be achieved. For the demand function (8) which can be rewritten as $D(q) = c_1 \times exp(c_2 - c_3 \times q)$ where $c_3 < 1$ the requirement simplifies to $q_B - q_A = \Delta q < ln(\alpha)$. In order to achieve the desired rate accuracy between the minimum supported sending rate x_{min} and maximum supported sending rate x_{max}, we need an encoding with fixed fractional part and fixed integer part. The integer part of the encoding must be long enough to accommodate $D^{-1}(x_{max})$, and the fractional part must be long enough so that the rounding error is less than $ln(\alpha)$.

Let B_{int} denote the bit width of the integer part and B_{frac} denote the bit width of the fractional part. We have $B_{int} = log_2(D^{-1}(x_{min}))$ and $B_{frac} = -log_2(ln(\alpha))$. Choosing x_{max} = 1e15 bits/sec, x_{min} = 32 bits/sec and α = 1.00001, we have $B_{int} = 5$ and $B_{frac} = 17$. We conclude that a fixed point encoding with 5 bit integer and 17 bit fractional part allows for huge dynamic range of the protocol, yet is compact enough. In our implementation the price could be encoded in many different ways. We decided to encode the price in a MaxNet TCP Option. Chapter 4 describes in detail how this is done.

$QSrate$ is encoded in the packets as $-ln\left(\frac{QSrate}{x_{max}}\right)$. This allows us to represent sending rates in the x_{min} to x_{max} range. This would not be possible if the rate was encoded directly as it would require up to 50 bits.

4 Implementation in the Linux Kernel

There are several ways of assessing the performance of a new TCP protocol, ranging from simulation to implementation and testing in an operating system. We chose to implement the protocol in the Linux kernel on top of the existing TCP/IP framework. On the one hand, using the existing kernel codebase speeds up the development of the protocol, and the results repre-

sent the actual performance of the protocol very accurately. On the other hand, implementing a rather complicated TCP protocol, such as MaxNet, in the kernel is very challenging. The kernel only supports simple integral arithmetic operations, and thus much care has to be taken to perform calculations with sufficient accuracy and range. This chapter describes our implementation in the kernel, all parameters and their values, as well as some challenges we had to overcome. First, we explain how non-integral calculations were performed. Next, we describe implementation of the router and sender code, and all parameters. We conclude by justifying some implementation specific changes we had to make in order to achieve good performance.

4.1 Non-integral Calculations in the Kernel

Performing non-integral calculations in the kernel is simple, albeit tedious. Our approach is best illustrated by an example. During the initialization phase the source calculates $\xi = q(\frac{\alpha}{baseRTT} - \frac{1}{T})$ where, e.g., $\alpha = 0.3$, $baseRTT$ = 0.028 sec, $T = 0.05$ and q = 181,000. Since we may only use integral variables, our encoding multiplies α, $baseRTT$ and T by a factor of 1,000, i.e., the variables initially contain α = 300, $baseRTT$ = 28, T = 50 and q = 181,000. The desired calculation is then relatively straightforward:

```
xi=(__s64)q*alpha;
do_div(xi, baseRTT);   // xi now contains q×alpha/baseRTT
temp = 1000*(__s64)q;  // temp now contains 1000 × q
do_div(temp, T);       // temp now contains q/T
xi -= temp;            // xi finally contains q(α/baseRTT − 1/T)
```

Other calculations are performed in a similar fashion.

4.2 Router Code

The router code, described in section 3.1., is implemented as `iproute2` dynamically loadable module. The module is loaded using the `tc` program and

it accepts several parameters. Typical values of these parameters, assuming capacity of the link 1 Gbit/sec, are: $C = 1$ Gbit/sec, $\mu = 0.94$, $dt = 1000$ μs, $QSminrate = \frac{C}{128}$, $\epsilon = 0$, $msgdt = 100{,}000$ μs and $p\_min = \frac{181{,}083}{2^{18}}$. If the capacity of the output link is different, the value of C and p_min has to be adjusted accordingly. The value dt was chosen small enough so that we can guarantee favorable convergence properties for flows with short RTTs, yet the value is large enough so that small bursts are averaged, and the calculations are not too frequent. The last two parameters, $msgdt$ and p_min, are specific to our implementation, and deserve detailed description.

Parameter $msgdt$ specifies the maximum frequency with which our code reports statistical information, such as price or y. The code reports this information using `printk`, and the output is then readily collected in `/var/log/messages`. If the prints occurred too frequently the performance of the router would suffer.

Parameter p_min specifies the minimal price the router attempts to signal when its links are underutilized. The rate of the sender is given by the demand function $D(q)$ in (8), and $D(0) = x_{max}$, where in our case $x_{max} =$ 1e15 bits/sec. When the router signals zero price, the sender will send data at a speed of up to 1e15 bits/sec. If the link has a lower capacity than x_{max}, signaling such a low price is undesirable. A valid price should never drop below the value corresponding to the maximal capacity of the link C. The value p_min is given by:

$$p\_min = -T \times ln(\frac{x}{x_{max}}). \tag{11}$$

When the router code changes the price contained in a packet, the TCP checksum needs to be updated. We implement an efficient incremental procedure that accepts the old checksum c, old price p, and new price p', and outputs the updated checksum c'. The new checksum is calculated according to [24] and [28] as $c' = \sim (\sim c + \sim p + p')$.

4.3 Sender Code

Implementation of the source algorithm, described in section 3.2., involves integrating the new TCP protocol in the existing TCP code. Newly introduced system control variables allow us to switch between TCP MaxNet and other congestion control algorithms as well as pass parameters to the MaxNet code at runtime. The following table summarizes the new system control variables we introduced, and provides typical values.

Variable	**Value**	**Purpose**
`tcp_maxnet`	1	Switches TCP MaxNet on / off.
`tcp_maxnet_debug_mode`	0	Allows printing debug messages, but degrades performance.
`tcp_maxnet_alpha`	0.3	α from section 3.2.
`tcp_maxnet_eta`	0.14	η from section 3.2.
`tcp_maxnet_wnd_update_freq`	1	The congestion window is recalculated after receiving this number of ACKs.
`tcp_maxnet_QS_enabled`	1	Switches QuickStart on / off. When switched off only price rate control mode is used.
`tcp_maxnet_max_wnd`	65535	Maximum allowed size of the window in packets.
`tcp_maxnet_dpdt_threshold`	16000	*dpdt* from section 3.2.
`tcp_maxnet_message_dt`	40000 μs	How often experimental data is printed. Too frequent prints degrade performance.
`tcp_maxnet_T`	0.05	T from section 3.2.

The main part of the algorithm serves to calculate the congestion window, and is called from `tcp_ack()`, a function that processes incoming ACKs. The algorithm retains values of the per connection variables, such as ξ and q, between the calls in `struct str_maxnet`, which is declared within `struct`

`tcp_sock`. The main challenge of the algorithm is accurate calculation of an exponential with a real parameter. Since the kernel only allows simple integer calculations, we solved this problem by hard-coding a lookup table. Our lookup table consists of approximately 64,000 records, and the individual elements of the table are addressed by the value of the exponent. Such an implementation allows both sufficient range and accuracy, and is more efficient than approximating the exponential by an iterative method.

Once the target congestion window is calculated, MaxNet provides this value to the operating system by overwriting the *snd_cwnd* variable in the `tcp_sock` structure. It is important to notice that the standard TCP framework in Linux only implements loss based congestion control algorithms and the congestion window is halved after each loss event. This decrease is not necessary with TCP MaxNet, because MaxNet decreases its sending rate as soon as the congestion price increases, and thus the loss the MaxNet sender experiences is not caused by congestion. For this reason, our algorithm overrides the standard behavior of the operating system, and enforces its calculated congestion window even after loss. This behavior enables the protocol to achieve excellent performance under loss. We report on this in the experimental section.

4.4 The MaxNet TCP Option

We introduce a new TCP option [29] that uses 6 bytes in each packet header to carry information about the price or QuickStart rate. The option format is depicted in Fig. 6. The purpose of the first byte is to advertise that this is TCP MaxNet option and the second byte advertises the length of the remaining fields. We suggest that, for performance reasons, the length of the option does not exceed 8 bytes (including the 2 leading bytes), which leaves 6 bytes divided between forward and echo field. Both the echo and forward fields carry either price or QuickStart rate whose length does not exceed 23 bits. This is a sufficient bit width as explained in section 3.3. The 24-th

bit in the forward and echo field is used to distinguish between packets that carry price and QuickStart rate.

The communication of the price and QucikStart rate is organized as follows. When a packet is sent, the forward field is initialized. The routers modify the forward field of the packet according to our algorithm, and when the packet is received, the receiver places the value from the forward field in the echo field. The TCP Option is then placed in an ACK, and when the original sender of the packet receives the ACK, it finds the price corresponding to the forward path in the echo field. This scheme also works with duplex connections.

The maximum total length of the TCP Options in the Linux kernel is limited to 60 bytes. 20 bytes are used for the TCP header, 12 bytes are used for the timestamp option and 28 bytes remain for selective acknowledgements (SACKs) and TCP MaxNet options. In standard TCP implementations, one can have up to 3 SACK blocks per header. They use 4 bytes plus 8 bytes per block, requiring up to 28 bytes and not leaving any space for TCP MaxNet options. Therefore, we are forced to change the standard SACK implementation and allow at most 2 blocks per SACK. This leaves 8 bytes for TCP MaxNet options. It is important that the MaxNet option does not use up more than 8 bytes, as availability of at least 2 blocks per SACK is crucial for the protocol's performance under loss. The reduction of the number of blocks per SACK did not have any negative impact on our experiments.

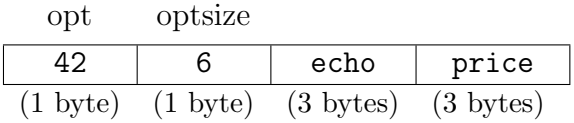


Figure 6: MaxNet option format.

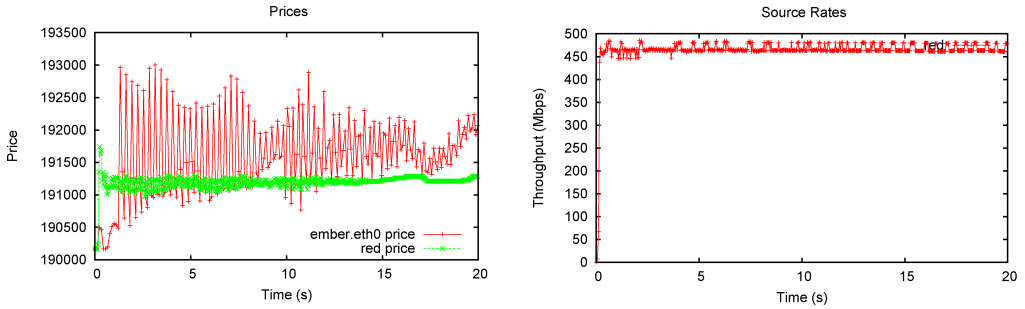
4.5 Robustness and Error Resilience

The MaxNet algorithm described in chapter 3 is based on a fluid traffic model. However, this model does not reflect reality accurately. Real traffic is packetized, packets may be reordered, lost, and may arrive in bursts. The senders, routers and receivers have limited memory and computational power, and thus the operations they perform are sometimes delayed or fail. In this section, we will discuss these limitations and suggest solutions.

Packetization and burstiness of traffic is the main challenge the fluid model faces. In chapter 3 we explained that the price update interval in the router code must be small so that the price reflects the immediate load of the links. However, when traffic arrives in bursts, the immediate load fluctuates, and the congestion signal is not stable. We solved this problem by averaging the incoming price at the sender side. Since the periodicity of any bursts must be lower or equal than one RTT, the source code was modified to average the incoming price over one RTT period, and the algorithm was modified to take into account the additional feedback delay.

The following experiment was conducted to demonstrate the qualitative difference between the immediate and averaged price. Our sender and receiver were connected by 1 Gbit/sec link controlled by a router. We used a token bucket filter which capped the transmission at 500 Mbit/sec at the router, but it allowed small bursts through. While the transmission was unstable when no price averaging was used, the throughput stabilized at 460 Mbit/sec when the price was averaged. Fig. 7(a) shows samples of the immediate price calculated by the router (red color), and the averaged value used by the source (green color). The sending rate of the sender when averaging was used is depicted in Fig. 7(b).

Another source of instabilities is change of base RTT estimate during runtime. Since the demand function (8) depends exponentially on $\frac{1}{baseRTT}$,



(a) Price calculated by the router (red), and averaged value (green).

(b) Throughput with 500 Mbit/sec bottleneck when averaging is used.

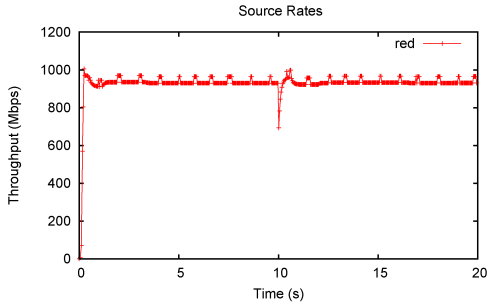
Figure 7: Price averaging over one RTT period at the sender side is required to stabilize the sending rates.

even a small change of base RTT causes a huge change of the congestion window. The size of the congestion window is later readjusted as ξ converges to its proper value. However, the initial mismatch of the window size is often so large so that it results in a temporary instability and oscillations. This instability can be avoided by changing the value of ξ to counterbalance the change in base RTT. When the base RTT estimate changes from $baseRTT_{old}$ to $baseRTT_{new}$, we need to change the value of ξ as follows:

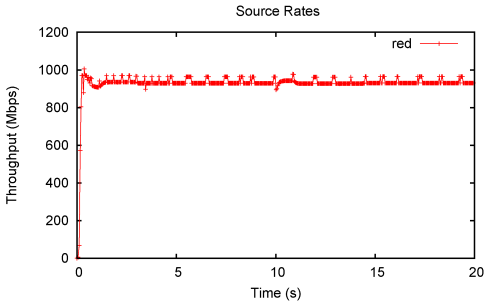
$$\xi = \xi + q\alpha(\frac{1}{baseRTT_{new}} - \frac{1}{baseRTT_{old}}). \tag{12}$$

We performed an experiment to compare the behavior of the system before and after the suggested change. We performed a simple 20 second data transfer and decreased the RTT estimate 10 seconds after the start of the experiment by 1 ms . Behavior of the system before the change is depicted in Fig. 8(a), 8(c) and 8(e). Fig. 8(a) depicts the throughput, Fig. 8(c) the base RTT estimate and the actual RTT, and Fig. 8(e) depicts the convergence of ξ to its target value. Results of the same experiment after the change was incorporated are depicted in Fig. 8(b), 8(d) and 8(f).

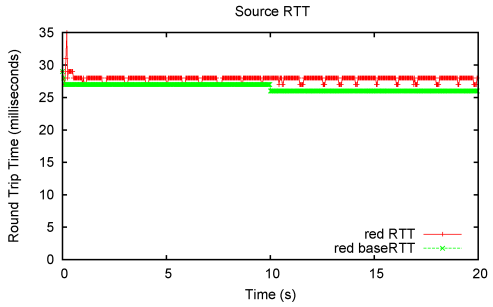
Minimizing processing overhead is an important issue in high speed net-



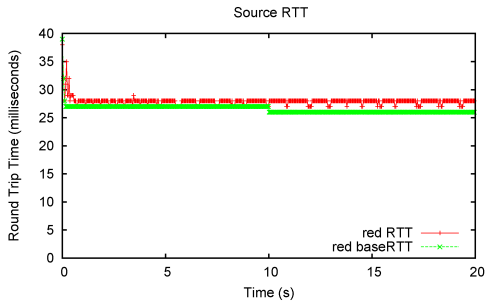
(a) Throughput before change.



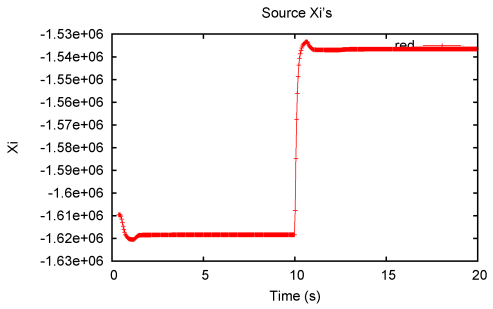
(b) Throughput after change.



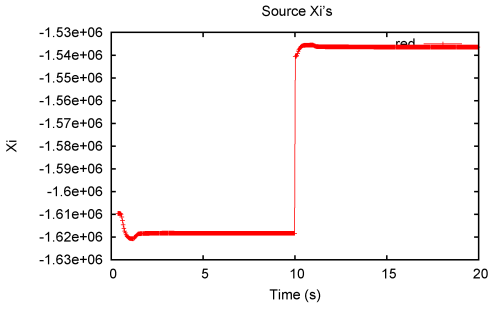
(c) Base RTT and RTT before change. We simulated a drop of the base RTT value at time 10 seconds.



(d) This is the same measurement as in c after the change.



(e) Convergence of ξ before the change.



(f) Convergence of ξ after the change is immediate.

Figure 8: Performance of the protocol when RTT drops. The figures on the left were obtained using the protocol as described in chapter 3. The figures on the right were obtained using a version that performs calculation given by equation (12) when RTT drops.

works. We optimized the source code and tested the protocol successfully at speeds up to 1 Gbit/sec. Input/output operations, such as printing debugging or status messages are the most time consuming operations. Thus, we only allowed message prints at fixed time intervals. Chapter 6 provides some suggestions how the computational overhead of the protocol could be further reduced.

5 Experimental Results

Performance evaluation of TCP MaxNet was conducted on a WAN network using WAN-in-Lab [30]. WAN-in-Lab is a wide area network consisting of an array of reconfigurable routers, servers and clients. The backbone of the network is connected by a 1600 km OC-48 link introducing 14 ms of one-way propagation delay. Our experimental setup consisting of two Linux routers and four Linux end hosts is depicted in Fig. 9.

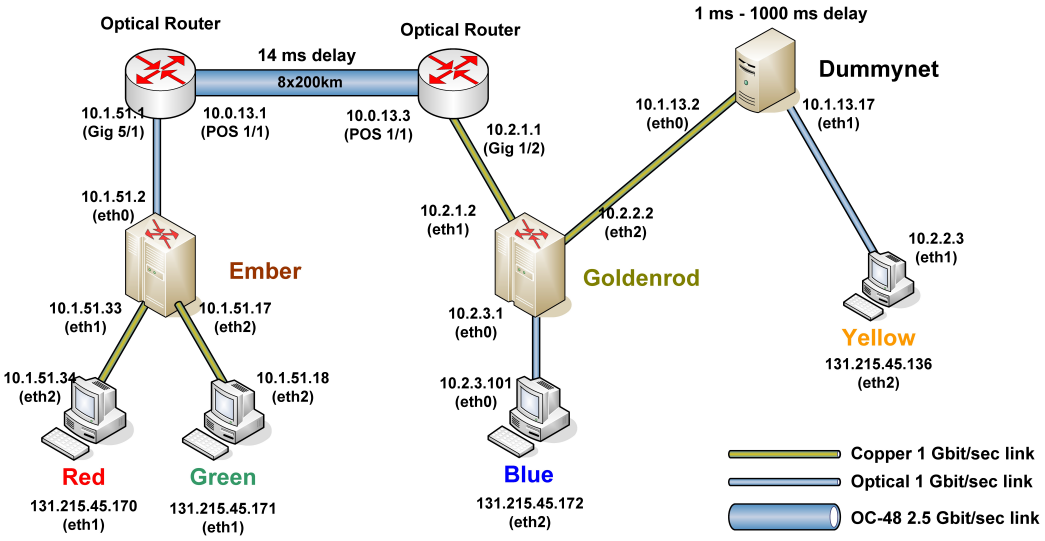
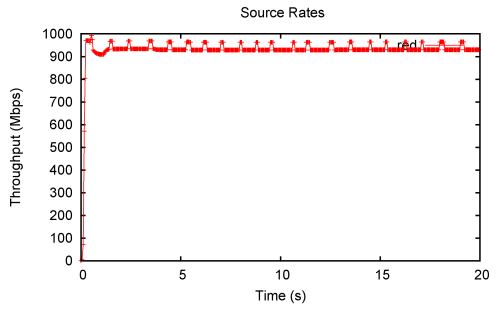


Figure 9: The topology of the Wan-in-Lab.

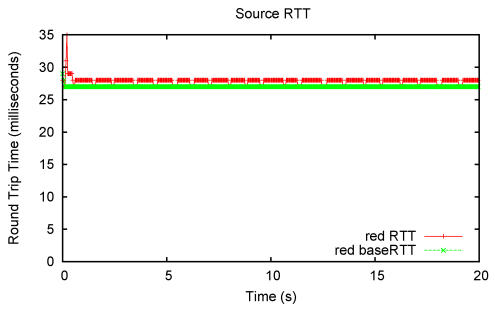
A dummynet [31] machine resides on the link between Goldenrod and Yellow. This allows to introduce propagation delays between Goldenrod and Yellow in both directions, simulating a real link. TCP MaxNet router code is installed on the routers (labeled Ember and Goldenrod) and the client side code is installed on the end hosts (labeled Red, Green, Blue and Yellow). This configuration allows us to test the protocol under various conditions, ranging from low delay low speed to high delay high speed environments. The capacity of the links in low speed tests is reduced by a token bucket filter, which is installed on the routers. The buffer sizes of the routers equal approximately one bandwidth delay product. Traffic in our experiments was generated by `netperf`.

5.1 Simple Data Transfer

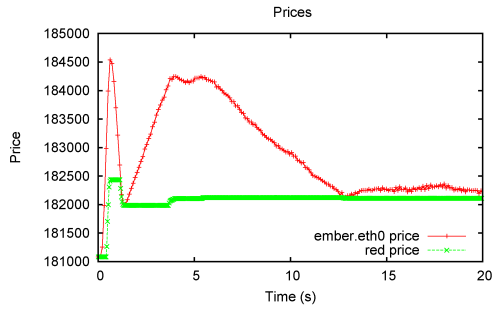
In the first experiment, we observe the behavior of the system with only one sender. Traffic is sent from Red to Blue over the 1 Gbit/sec link with a 14 ms one-way propagation delay. During the experiment, we record the immediate raw throughput of the protocol as measured by Red and Ember, window size, RTT, drop rate, queue length at Ember, and price and ξ at Red in periodic intervals. The raw throughput is calculated by Red as the number of packets in flight divided by the RTT, and the other variables are obtained directly from the kernels. Fig. 10 depicts results of the experiment. Fig. 10(a) shows the throughput measured by the sender, and Fig. 10(b) shows the corresponding RTT. While the protocol achieves nearly full utilization of the link, the RTT stays very close to the two-way propagation delay of the link, the theoretical minimum. We show in [29] that this is something that other loss or delay based protocols, such as BIC and FAST, cannot achieve. Both loss and delay based protocols need to keep the buffers at least partially filled, which results in much higher latency. Our measurements of router queues confirmed that queues do not build up, and no packets are dropped. Fig. 10(c) and Fig. 10(d) depict price and ξ. Both price and ξ quickly converge to their theoretical values corresponding to a



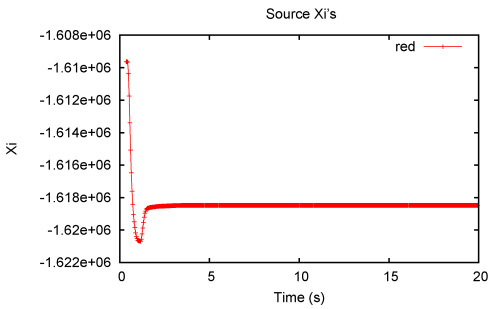
(a) Transfer rate of the sender.



(b) RTT of the flow stays around the two-way propagation delay of the link.



(c) Price calculated by the bottleneck router and averaged price used by the sender to adjust its congestion window.



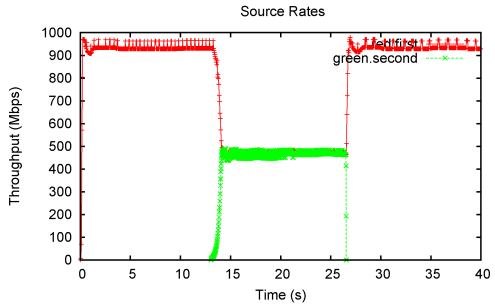
(d) ξ calculated by the sender.

Figure 10: Simple data transfer at 1 Gbit/sec between two endhosts.

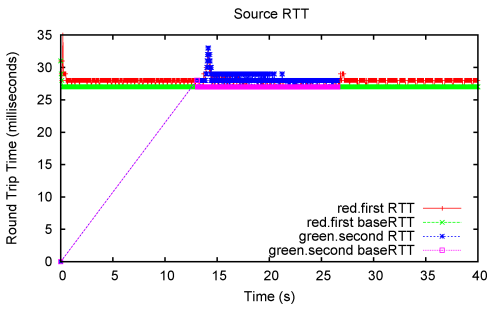
1 Gbit/sec transmission over a link with 28 ms base RTT.

5.2 Multiple Flows

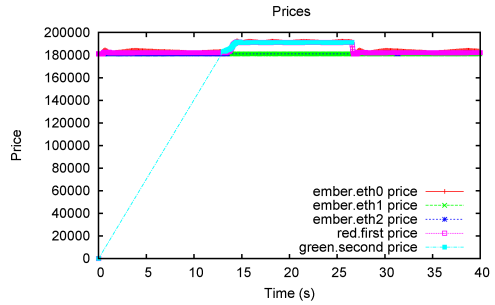
In this experiment we study the dynamic properties of the protocol when a flow enters and leaves the system. Traffic is sent for 40 seconds from Red to Blue over a 1 Gbit/sec link with 14 ms one-way propagation delay. 13 seconds after the start of the experiment a second flow from Green to Blue joins the network. The second flow leaves the network at time 27 seconds after the start of the experiment. We measure the same observables as in the simple data transfer experiment. The results are depicted in Fig. 11. We



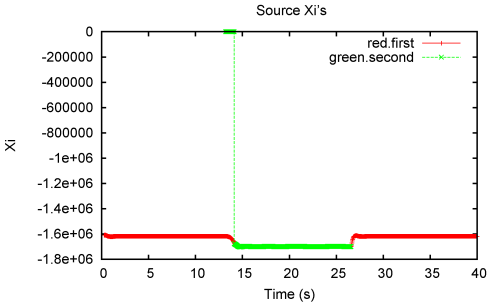
(a) Throughput of the two flows.



(b) Base RTT stays around the theoretical minimum.



(c) Price calculated by the router and received by the sources.



(d) ξ used by the senders.

Figure 11: 1 Gbit/sec link shared by two flows. The second flow joins the network at time 13 sec and leaves at time 27 sec.

observe that the second flow acquires half of the available bandwidth quickly after joining, and the first flow promptly reduces its sending rate. After the second flow leaves, the first flow almost immediately claims the remaining free capacity. Besides fair sharing, this experiment demonstrates several crucial properties of TCP MaxNet.

First, the operation of the QuickStart algorithm is demonstrated. When the second flow joins the network, the router communicates the maximum of the available capacity and QSminrate to the source of the second flow. The source then exponentially increases this rate. In the meantime, the rate of the first flow decreases, and when the sending rates equalize, the source of

the second flow switches to the price controlled mode.

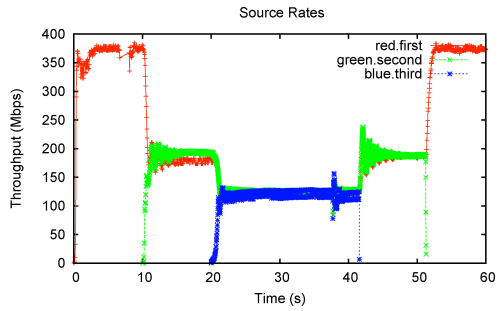
Second, the experiment demonstrates excellent behavior of the protocol in the price rate control mode. When the second flow ramps up, the price calculated by the router increases. We observe that the first sender, which is in the price rate controlled mode, promptly decreases its sending rate. It is important to notice that the RTTs of both flows stay around the base RTT value throughout the duration of the experiment, and virtually no queues build up in the routers. The dynamics of the price controlled mode is also tested when the second flow leaves the network. In this case, the price signaled by the router sharply drops, and the source of the first flow converges to the target 1 Gbit/sec utilization within a few RTTs. Again, the price control mode handles the situation properly and the rate stabilizes at 1 Gbit/sec, and no queues build up.

The absence of router queues is a significant advantage over the other available TCP protocols. In [29] we show that while other commonly used protocols are usually able to share bandwidth among two flows between end hosts with identical propagation delay fairly, they require long router queues which introduce additional delay.

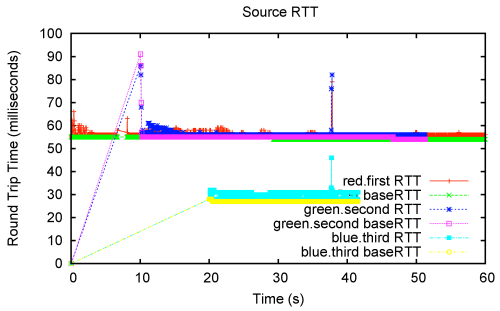
5.3 Testing Fairness

It is well known that stability requirements of commonly used congestion avoidance algorithms impose dependence of the target throughput on RTT [29]. Therefore, TCP protocols usually do not share bottleneck capacity fairly among flows with different RTTs. In this experiment we show that MaxNet is able to share bottleneck capacity fairly even when the RTTs of the participating flows vary.

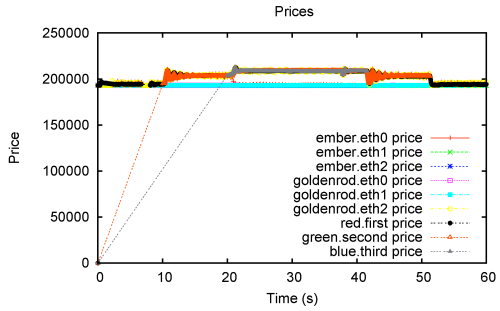
This experiment uses three flows. The first two flows connect Red to Yellow and Green to Yellow respectively. The dummynet is set to provide



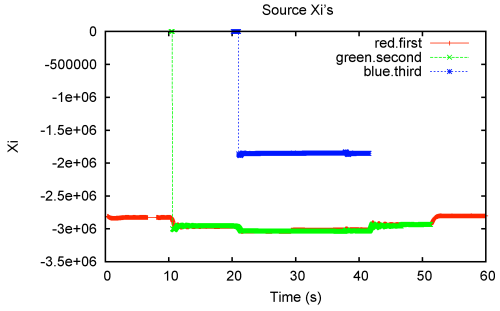
(a) The sending rates of the three flows.



(b) All RTTs stay around the theoretical minimum.



(c) Prices calculated by the routers.



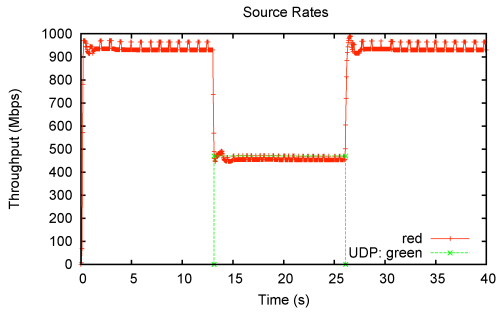
(d) Values of ξ in the senders.

Figure 12: The three flows share the bottleneck capacity fairly despite the fact that the RTT of the third flow is lower than the RTT of the first two flows.

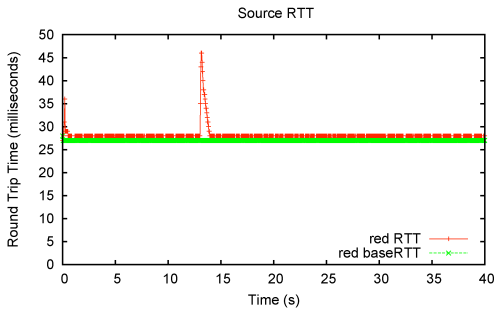
14 ms of one way propagation delay. Thus, the two-way propagation delay on the links that the two flows use is about 56 ms. The third flow connects Blue and Yellow, and has a two-way propagation delay of about 28 ms. Since dummynet is not able to run at 1 Gbit/sec with our hardware, this experiment was performed at 400 Mbits/sec. The result is depicted in Fig. 12. We observe that when the third flow joins the network the rates are equalized and fair sharing is achieved.

5.4 Sharing Bandwidth with a UDP Flow

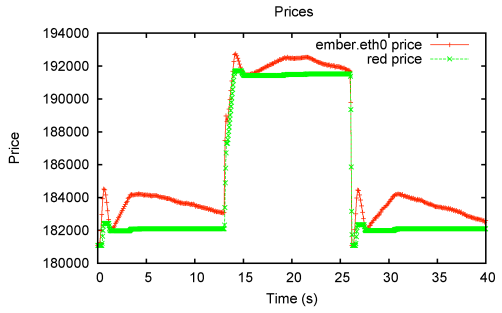
This experiment demonstrates the dynamic properties of the price rate controlled mode of MaxNet. The experiment is essentially the same as the one



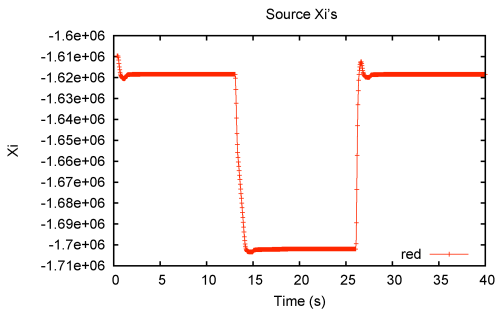
(a) Sending rates of the MaxNet and UDP flows.



(b) RTT of the MaxNet flow.



(c) Price in the router and in the MaxNet sender.



(d) ξ of the MaxNet flow.

Figure 13: TCP MaxNet and UDP share a 1 Gbit/sec link.

in section 5.2, but the second flow is substituted by a UDP connection with a fixed bandwidth of 470 Mbit/sec. The first flow is a MaxNet flow connecting Red to Blue and the second flow, which joins at time 13 seconds and leaves at time 27 seconds, connects Green and Blue. The results of the experiment are depicted in Fig. 13. Although the UDP flow starts its transmission much more aggressively than a TCP connection would, we observe that the MaxNet source is able to reduce its sending rate almost immediately in order to accommodate the UDP flow. Although the RTT of the first flow increases briefly after the second flow joins, the queue length quickly decreases to zero. After the UDP flow leaves the network, we again observe that the MaxNet price control handles the situation as desired and increases the sending rate of the flow to 1 Gbit/sec.

5.5 Performance under Loss

Performance of TCP MaxNet under loss is evaluated by measuring the raw throughput of a TCP flow between Red and Blue. Loss between 0% and 0.25% is introduced in Ember on all its interfaces. This simulates loss in both directions on the backbone link between Ember and Goldenrod. Average throughput of the protocol for each loss level is measured with five consecutive `iperf` flows, each running for 20 seconds. Fig. 14 shows the average throughput of TCP MaxNet, TCP Reno, TCP BIC and TCP Westwood. TCP MaxNet outperforms the other protocols because it does not reduce its congestion window when loss occurs. The standard protocols assume that loss is a symptom of congestion, and thus they decrease the sending rate severely when non-congestion loss is introduced.

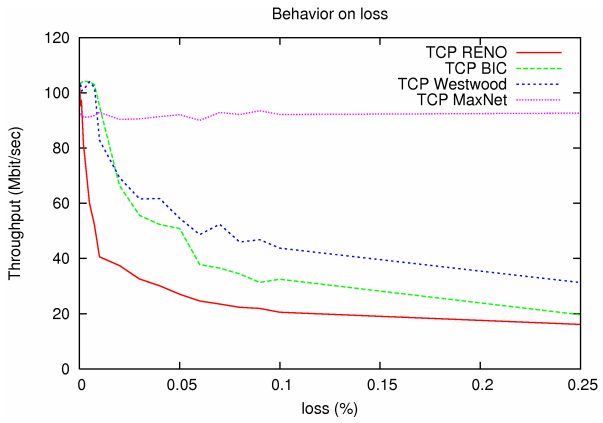


Figure 14: TCP MaxNet achieves much better throughput than other protocols when non-congestion based loss is present. Loss level as low as 0.1% has devastating consequence for the standard protocols.

6 Future Research

Our experiments show that TCP MaxNet achieves excellent performance. However, we would like to point out the limitations of our work, and suggest further improvements, some of which will be necessary before the protocol is adopted.

The main problem which will need to be addressed is fair sharing of bandwidth with other TCP protocols and UDP. Since the sending rate of TCP MaxNet depends on the available spare capacity in the routers, TCP MaxNet is penalized in systems where other protocols, such as the ubiquitous TCP Reno, try to aggressively push data through the network, and disobey the price signal. We suggest separating the MaxNet queue in the routers, and using a variant of selective packet dropping method, such as CHOKE [32], to enforce approximate fairness. Separating the MaxNet queue from the standard queue seems to be advantageous for the following two reasons. First, the separate queue allows MaxNet to enjoy low latency and other favorable properties. Second, the selective dropping algorithm would be able to accurately adjust the aggregate sending rate of the MaxNet flows simply by changing the target utilization constant μ of the MaxNet queue in runtime. Although the issue of fair sharing in heterogeneous environments has not received much attention in connection with other recent TCP implementations, we believe that this challenge can be overcome by using the approximate fair sharing schemes presented e.g. in [32], [33] and [34].

Although our implementation could theoretically scale up to 1 Petabit/sec, increasing CPU utilization will become a performance bottleneck at higher speeds. In order to save CPU cycles, we suggest reducing the frequency of packet marking at higher speeds. For example, one could mark every tenth packet at speeds exceeding 100 Mbits/sec, and every hundredth packet at speeds exceeding 1 Gbit/sec. This would reduce the load of both the senders and the routers. We believe that speeds exceeding 10 Gbits/sec should be achievable with minimal changes of the code.

7 Conclusion

We are the first to provide working implementation of TCP MaxNet and evaluate its performance. Our contribution is twofold. First, we provide solutions

to the implementation related challenges. We show how to use a variant of the QuickStart algorithm to accommodate new flows that join the network, and we demonstrate at 1 Gbit/sec speeds that it only takes a fraction of a second for the algorithm to bring the system to equilibrium. We improve the price rate controlled algorithm to be more robust. We provide detailed pseudocode of the MaxNet algorithms that are readily implementable. We show that the algorithms we use scale with speeds and delays with desired accuracy. We provide a method of evaluating exponential and carry out fractional operations in the Linux kernel, and we point out changes imposed by the implementation, such as price averaging, that are needed to achieve good performance. Our second contribution is performance evaluation of the protocol. Our results confirm previous theoretical predictions and show that TCP MaxNet is for many reasons an attractive protocol. The dynamic properties of the protocol are excellent. For example, when a UDP flow at 470 Mbits/sec joins a 1 Gbit/sec MaxNet flow, the speed of the MaxNet flow decreases by 470 Mbits/sec within a few RTTs, while the router queues stay empty. Next, we demonstrate that TCP MaxNet is capable of fair sharing of bottleneck capacity even when the propagation delays of the competing flows vary. Moreover, we demonstrate that TCP MaxNet achieves extraordinary performance in lossy environments. Finally, we observe that the router queues stayed empty during all of our experiments, and we conclude that MaxNet has very low latency and high responsiveness.

References

[1] J. Postel. RFC 793: Transmission control protocol, September 1981.

[2] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[3] S. Floyd and T. Henderson. RFC 2582: The NewReno modification to TCPs Fast Recovery algorithm, April 1999.

[4] T. V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, 1997.

[5] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communications Review*, 27(3), 1997.

[6] C. Jin, D. Wei, and S. Low. FAST TCP: Motivation, architecture, algorithms, performance. Caltech CS Technical Report, December 2003.

[7] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fastlong-distance networks. In *Proceedings of IEEE Infocom*, March 2004.

[8] S. Floyd. RFC 3649: Highspeed TCP for large congestion windows, March 2003.

[9] D. Leith and R. Shorten. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDnet*, Argonne, 2004.

[10] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Computer Communication Review*, 33(2):83–91, 2003.

[11] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297, 2001.

[12] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[13] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *Proceedings of IEEE Infocom*, March 2004.

[14] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, to appear in 2007.

[15] S. H. Low, L. L. Peterson, and L. Wang. Understanding TCP vegas: a duality model. In *SIGMETRICS/Performance*, pages 226–235, 2001.

[16] K. K. Ramakrishnan and S. Floyd. RFC 2481: Proposal to add explicit congestion notification (ECN) to IP, January 1999.

[17] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[18] H. Balakrishnan, N. Dukkipati, N. McKeown, and C. Tomlin. Stability analysis of switched hybrid time-delay systems – analysis of the rate control protocol. Stanford University Technical Report, June 2004.

[19] S. Low, L. Andrew, and B. Wydrowski. Understanding XCP: Equilibrium and fairness. In *Proceedings of IEEE Infocom*, Miami, Florida, March 2005.

[20] L. Andrew, M. Suchara, R. Witt, and B. Wydrowski. MaxNet: From theory to implementation. *In preparation for publication.*

[21] B. Wydrowski, L. L. H. Andrew, and M. Zukerman. MaxNet: A congestion control architecture for scalable networks. *IEEE Communication Letters*, 7:511–513, October 2003.

[22] B. Wydrowski and M. Zukerman. MaxNet: A congestion control architecture for MaxMin fairness. *IEEE Communication Letters*, 6:512–514, November 2002.

[23] B. Wydrowski, L. L. H. Andrew, and I. M. Y. Mareels. MaxNet: Faster flow control convergence. *Networking 2004*. Springer Lecture Notes in Computer Science LNCS 3042.

[24] B. Braden, D. Borman, and C. Partridge. RFC 1071: Computing the internet checksum, September 1988.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgement options, October 1996.

[26] A. Jain, S. Floyd, M. Allman, and P. Sarolahti. Quick-start for TCP and IP. Internet draft, September 2004.

[27] F. Paganini, Z. Wang, J. C. Doyle, and S. H. Low. Congestion control for high performance, stability, and fairness in general networks. *IEEE/ACM Transactions on Networking*, 13(1):43–56, 2005.

[28] A. Rijsinghani. RFC 1624: Computation of the internet checksum via incremental update, May 1994.

[29] M. Suchara, R. Witt, and B. Wydrowski. TCP MaxNet - implementation and experiments on the WAN in Lab. In *Proceedings of the Joint IEEE International Conference on Networks (ICON 2005)*, pages 901–906, Malaysia, November 2005.

[30] WAN in Lab, available online: http://wil.cs.caltech.edu.

[31] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[32] R. Pan, B. Prabhakar, and K. Psounis. CHOKE, a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proceedings of IEEE Infocom*, volume 2, pages 942–951, 2000.

[33] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 118–130, 1998.

[34] W. Feng, D. D. Kandlur, D. Saha, and K. G. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *Proceedings of IEEE Infocom*, pages 1520–1529, 2001.