On Achieving Weighted Service Differentiation: An End-to-End Perspective [⋆]

Hung-Yun Hsieh, Kyu-Han Kim, and Raghupathy Sivakumar

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, 30332, USA
{hyhsieh@ece, khkim94@cc, siva@ece}.gatech.edu

Abstract. In this paper, we consider the problem of weighted rate differentiation using purely end-to-end mechanisms. Existing approaches to solving the problem involve changes in the AIMD congestion control mechanism used by TCP. However, such approaches either do not scale well to large weights, or make impractical assumptions. We use a new multi-state transport layer solution called *pTCP* to achieve end-to-end weighted service differentiation. A pTCP flow of weight *w* consists of *w* TCP *virtual flows* that collectively achieve *w* times the throughput of a default TCP flow. pTCP scales significantly better than approaches that change the AIMD congestion control mechanism of TCP. On the other hand, pTCP achieves more effective service differentiation and incurs less host overhead than the simplest form of a multi-state solution using multiple TCP sockets through application striping. We substantiate our arguments through simulations, and testbed experiments based on a user-level implementation of pTCP.

1 Introduction

Approaches to achieve relative service differentiation are inherently much simpler to deploy and manage. Hence, the paradigm of relative service differentiation has gained considerable attention over the last few years [1, 2]. In this work, we consider the specific problem of achieving weighted rate differentiation using purely end-to-end mechanisms. A solution to such a problem will have the added benefit of supporting scalable QoS without any infrastructure change.

An important instantiation of such an end-to-end weighted rate differentiation solution lies in the incorporation of weighted fairness within the TCP transport protocol design. Essentially, under a weighted fairness model, a TCP flow of weight *w* is to be provided with *w* times the throughput of a default TCP flow (with unit weight). In this context, several related works have been proposed to modify the AIMD (additive increase, multiplicative decrease) congestion control of TCP for achieving the desired throughput. For example, MulTCP [3] changes the AIMD parameters according to the weight of the flow. While such a weighted AIMD (WAIMD) scheme requires minimal changes to TCP, simulation and testbed results [3, 4] have shown that WAIMD can provide weighted rate differentiation only for a small range of weights (less than 10). For

[⋆] This work was funded in part by NSF grants ANI-0117840 and ECS-0225497, Motorola, Yamacraw, and the Georgia Tech Broadband Institute.

larger weights, it suffers from frequent packet losses, and does not provide consistent service differentiation. TCP-LASD [5] is an approach proposed to improve the performance of WAIMD by adapting the AIMD parameters to the packet loss rate as well as the flow weight. Although TCP-LASD exhibits higher scalability in terms of weight (up to 100), it relies on accurate loss estimation that is difficult to achieve using purely end-to-end schemes without any network support [5].

The fact that WAIMD maintains only one TCB (TCP control block) [6] per connection makes it vulnerable to deviation from the ideal behavior under severe loss conditions. Specifically, given an identical distribution of packet losses, a connection maintaining only one TCB will be impacted by timeouts far more than one that maintains multiple TCBs. When timeouts occur in the former, the whole connection will stall until the loss is recovered, but in the latter, only the affected TCB(s) will stall. While an application-striping approach using multiple TCP sockets (and hence multiple TCBs) has been studied and experimented in a different context [7–10], it has thus far not been considered as a potential solution for achieving weighted service differentiation due to the following reasons: (i) The throughput gain is not consistently proportional to the degree of parallelism (number of sockets used) even for a small number of sockets (around 10). Hence it is difficult to decide (in a distributed fashion) the number of parallel sockets to use for achieving the desired weighted fairness under any given network condition. (ii) Striping is usually done by dividing the application data into same-sized partitions equal to the number of sockets before transmission commences. Each partition of data is therefore transferred asynchronously by the corresponding streams (sockets) and is reassembled after all transmissions are complete (offline reassembly). This approach thus cannot be used by applications that require strict TCP semantics including in-sequence data delivery. The added complexity imposed on applications to perform the sophisticated striping techniques (e.g. data partition and reassembly) also renders this approach less desirable.

In this paper, we first study the performance of an application-striping approach when used as a solution to achieve weighted service differentiation. We show that it fares better than WAIMD in terms of scalability to increasing weights, but still does not achieve the ideal expected weighted service differentiation beyond small weight values. We provide insights into the performance limitations of application striping. We then use a multi-state transport layer protocol called **pTCP** (parallel TCP) that maintains multiple TCBs per connection but avoids the pitfalls of application striping, for achieving the desired weighted differentiation. pTCP was originally proposed in [11] to aggregate bandwidths on a mobile host with multiple network interfaces. We tailor the design of pTCP for the specific goal of achieving weighted service differentiation. A pTCP connection of weight w consists of w mini-flows called *TCP-v* (TCP-virtual) flows. TCP-v is a simple variation of default TCP that employs the same congestion control mechanisms, but does not deal with the actual application data. A central entity called the pTCP engine manages the send and receive socket buffers, and handles reliability and flow control for the pTCP connection. Through both simulation results and real-life evaluation of a user-level implementation of pTCP, we show that pTCP is able to achieve effective weighted service differentiation, exhibiting a much higher scalability to the range of weights than both WAIMD and application striping.

The rest of this paper is organized as follows: In Section 2 we explain the goals and scope of this paper, and in Section 3 we discuss related work that uses end-to-end mechanisms to achieve weighted service differentiation. In Section 4 we present the pTCP design and protocol. Section 5 presents simulation and prototype implementation results showing the performance of pTCP. Section 6 discusses some critical issues in the pTCP design, and finally Section 7 concludes the paper.

2 Goals and Scope

– **Weighted Rate Differentiation:** The *DiffServ* framework supports absolute service differentiation with guarantees on absolute performance levels, as well as relative service differentiation with assurances for relative quality ordering between classes [1, 12]. Many approaches and architectures have been proposed to achieve relative service differentiation for different service parameters and applications [2, 5, 13–15]. It is shown in [1] that relative service differentiation requires a "tuning knob" to adjust the relative QoS spacing between classes, and hence the *proportional differentiation* model, where the performance experienced by a certain class is proportional to the service differentiation parameters, stands out for its ability to achieve *controllable* and *predictable* service differentiation. The goal of this paper is to provide such a "tuning knob" for achieving relative service differentiation. We consider data rate differentiation and hence a user (a TCP flow in particular) with weight or differentiation parameter w is to be provided w times the data rate of a user with unit weight.
– **End-to-End Approach:** We aim to provide the weighted rate differentiation using purely end-to-end mechanisms. While a complete model for providing relative service differentiation requires appropriate pricing and policing enforced at the edge routers, we do not rely on network support to achieve service differentiation. No assumptions are made in terms of network support except for a fair dropping mechanism such as RED. Note that the fair dropping requirement is also essential for TCP to achieve the default (non-weighted) proportional fairness [16–18].
– **TCP Friendliness:** We define TCP friendliness in the context of weighted service differentiation as follows: A weighted TCP flow of weight w will receive exactly the throughput that would have been enjoyed by w default TCP flows (which replace the weighted flow under the same network condition) in total. Note that given our TCP friendliness goal, the behavior of a TCP flow with weight w should be exactly that of an aggregation of w unit TCP flows. This in turn means that the weighted flow will still exhibit any undesirable property that TCP might have (such as RTT bias). While it is not the goal of this paper to change the behavior of a default TCP flow, any improved congestion control mechanism used by TCP should be easily incorporated by the weighted flow to achieve the desired service differentiation.
– **TCP Semantics:** We limit the scope of this paper to applications that require the end-to-end semantics of TCP in terms of reliability and in-sequence delivery. The conventional application-striping approach [7, 8] that partitions application data and transfers different portions through different TCP connections thus falls outside the scope of this paper, since it requires the receiving application to perform *offline processing* to reassemble the collected data portions.

3 Motivation

There are two broad classes of approaches that can be used to achieve end-to-end weighted rate differentiation: (i) *Weighted AIMD:* A single-state approach that uses a single TCP flow with a modified AIMD congestion control mechanism, and (ii) *Application Striping:* A multi-state approach that stripes across multiple default TCP sockets. The first class of approaches has been studied in quite some detail by related work [3, 5]. However, the latter class of approaches, although proposed in other contexts [7, 8], has not been investigated as a viable option for achieving weighted rate differentiation. In the rest of the section, we first outline the reasons for the *non-scalability* of vanilla WAIMD, and discuss the limitations of a variant called TCP-LASD. We then study the performance of application striping in the context of weighted rate differentiation, and provide insights into its benefits and limitations.

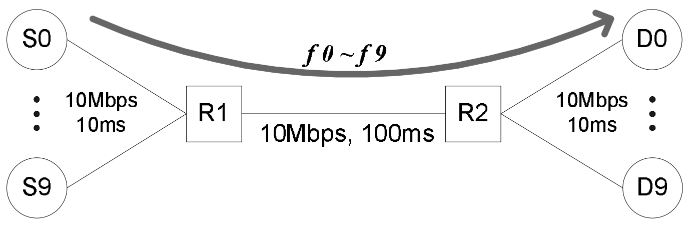


Fig. 1. Network Topology (Single Link)

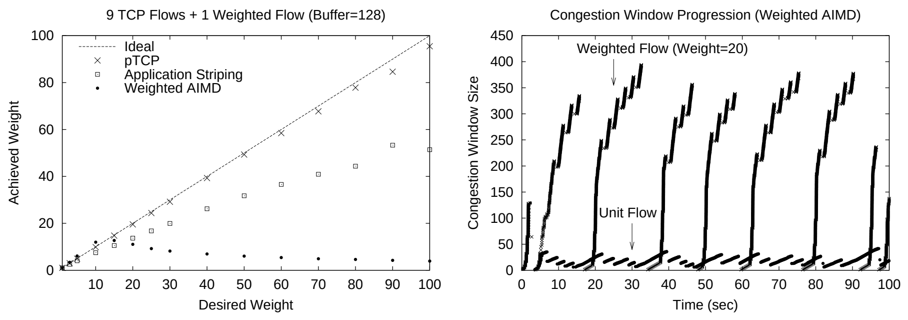
We use simulation results based on the *ns-2* network simulator [19] to illustrate and substantiate our arguments during the discussion. The topology used consists of a single link topology with 2 backbone routers and 20 access nodes as shown in Fig. 1 The end-to-end path traversed by the 10 TCP flows (flow i originates from S_i and terminates at D_i) has bottleneck capacity of 10Mbps and base round-trip time of 240ms. The backbone routers use the RED queue with buffer size of 320KB (approximately the bandwidth-delay product of the end-to-end path), and the access nodes use the DropTail queue. $f0$ is a weighted flow with weight w, and $f1$ through $f9$ are regular TCP flows with unit weight. We use TCP-SACK for all TCP flows.

3.1 Limitations of Weighted AIMD

Ideally, the throughput of a weighted flow with weight w should be w times that of the average throughput enjoyed by the unit flows. However, it is clear from Fig. 2(a) that[1] WAIMD does not scale beyond a weight of even 10. The reasons for such poor performance stem from the following two properties of WAIMD:

– **Burstiness:** For a flow with weight w, WAIMD modifies the increase (α) and decrease (β) parameters in AIMD to be αw, and $\frac{\beta}{w}$ respectively. It is clear that the

[1] The performance of pTCP shown in Fig. 2 and Fig. 3 will be discussed in Section 5.1.



(a) Scalability with weight (b) Congestion window progression

Fig. 2. Limitations of Weighted AIMD

congestion window progression for a weighted flow becomes more bursty with increasing weights. Burstiness induces packet losses due to queue overflows, which eventually result in timeouts if sufficiently large number of packets are lost from within one congestion window, or a retransmitted packet is lost. It is shown in [5] that the loss probability at a bottleneck link increases proportional to the *square of the sum of weights* of all contending flows. While the SACK option can ameliorate the impact of losses to some extent, due to the limited number of SACK blocks available in the TCP header, its benefits do not scale with the amount of losses. Hence, WAIMD becomes more bursty with increasing weights, and consequently experiences increasing chances of timeouts. Fig. 2(b) presents the congestion window evolution of the WAIMD flow. The burstiness is evident through the steep increases in the congestion window. Moreover, as the figure shows, 8 timeouts occur during the period of 100 seconds, substantiating our argument that the bursty nature of the traffic induces frequent timeouts in WAIMD.

– **Single State:** Timeouts in general degrade TCP's throughput performance. However, they drastically impact WAIMD's performance because of its single state strategy. Specifically, WAIMD maintains only one TCB per connection (just as TCP). Hence, when a timeout occurs, irrespective of the weight *w* of a flow, the flow is completely shutdown. In contrast, if the flow were really replaced with *w* default TCP flows, even if one of the flows experiences a timeout, the remaining flows would have been able to continue without interruption. However, WAIMD cannot emulate such a behavior because of its single state design.

While WAIMD cannot achieve the desired service differentiation due to the inability to handle packet losses induced by its burstiness, in [5] the authors propose a *loss-adaptive* variant of WAIMD, called TCP-LASD, that adapts its increase and decrease parameters as a function of the loss rate (p) observed–e.g., by setting $\alpha \propto \frac{1}{p}$, or $\beta \propto p$. They show that TCP-LASD can scale considerably better than WAIMD if accurate loss rate information is available at all sources. However, such an assumption, even the authors agree, is not pragmatic in a distributed environment. This is due to the fact that even when network routers attempt to provide a *fair dropping probability*, they merely strive to achieve a fair expected loss probability for all packets in the *long term* (several round-trip times), and do not try to reduce the variance in the loss probabilities accorded to different flows. However, the fact that TCP-LASD relies on accurate loss estimation to drive the congestion window progression on a per-packet basis, makes its performance sensitive to even *short-term* loss rate estimation. While the authors argue that the problem can be offset with accurate loss rate information feedback from the network, this solution deviates from the scope of this paper in terms of providing service differentiation through purely end-to-end mechanisms.

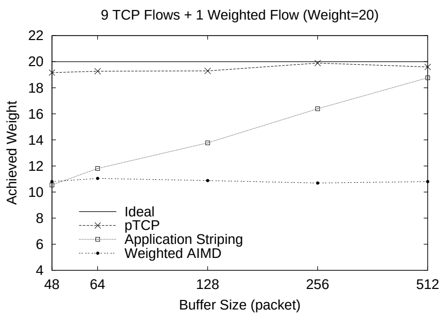
3.2 Limitations of Application Striping

In the application-striping approach, the application opens multiple sockets to the destination, and explicitly stripes data across the different sockets in an attempt to achieve better throughput performance. Specifically, we assume that a sending application will open w socket connections with the destination for achieving the desired weight of w. In keeping with the TCP semantics of in-sequence delivery, we assume that the receiving application will read only from the socket buffer that has the next expected application layer sequence number. The application can perform such an in-sequence read by setting the *peek* flag in the socket read options (which will prevent the last read packet from being dequeued from the socket buffer), and performing an actual read only when it knows that the socket buffer has the next expected packet.[2]

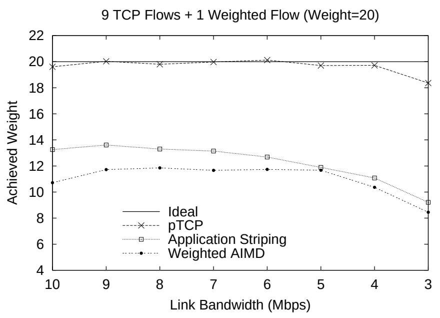
We observe from Fig. 2(a) that the performance of application striping is considerably better than that of WAIMD. This can be attributed to the *multiple-state* strategy adopted by the application-striping approach. In other words, in application striping, one TCB is maintained per unit TCP flow. Hence, even when one of the flows experiences a timeout, the other flows are still free to continue transmitting. The limiting factor, however, will be the *head-of-line blocking* at the receiver-side buffer. Since the application reads only in-sequence, although flows not experiencing a timeout can continue transmitting, the receiving application will not read from their socket buffers if the next in-sequence packet is carried by the timeout-stalled flow. When their buffers thus become full, the TCP receivers of those flows will advertise a window size of zero and force the TCP senders to enter the *persist* mode [20], eventually stalling the flows.

Note from Fig. 2(a) that the performance of application striping deviates from the ideal behavior for weights larger than 10. We now proceed to evaluate and explain its performance in terms of the buffer requirement, and when the per-unit-flow fair share of the bandwidth amounts to a small (and hence timeout-prone) congestion window. Fig. 3(a) studies the impact of the receive buffer size per TCB on the performance of the aggregate connection. Since the bandwidth-delay product for each TCB is approximately 24KB, it can be observed that application striping is able to achieve close to the ideal performance only for significantly large buffers (of 512KB or above for a weight of 20). In other words, close to *twenty times the bandwidth-delay product worth of buffering is required for each socket* for the performance of the application striping

[2] While we take such an approach to reduce the application complexity and overhead (in terms of the resequencing process required), note that any performance degradation due to the apparent lack of an application resequencing buffer can be compensated by using a large TCP socket buffer. We do study the impact of the buffer size at the receiving TCPs later in this section.



(a) Buffer requirement (per TCB)



(b) Scalability with fair share

Fig. 3. Limitations of Application Striping

to even reach close to the ideal performance (the total amount of buffering used by the application is the per-socket buffer size times the number of sockets open). Clearly, application striping fails to *scale* to large weights, from purely the standpoint of the amount of memory expended at the receiver. The dependence on such large buffers is due to the head-of-line blocking otherwise experienced, either (i) when striping is performed in a manner disproportionate with the instantaneous rate of the individual TCP flows by an unaware application, or more importantly (ii) when packet losses or timeouts occur.

Fig. 3(b) studies the impact of the fair share per TCB (unit flow) on the performance of the aggregate connection. Note that the congestion window size of a TCP flow is reflective of the fair share available. When the average congestion window size goes down below a value of *six*, it has been shown in [21] that TCP is more prone to timeouts due to the unavailability of sufficient data packets to trigger fast retransmit (3 duplicate ACKs) after a loss event. When timeouts do occur, although application striping does not suffer from the single-state problem in WAIMD, it is still vulnerable to the head-of-line blocking problem due to the filling up of the individual receive buffers (when one or more flows are recovering from a timeout). The results shown in Fig. 3(b) are obtained by varying the bandwidth of the bottleneck link from 10Mbps to 3Mbps, which in turn changes the average congestion window size per TCB (the queue size of RED routers and the buffer size per TCB scale accordingly). It is clear that while application striping achieves only about half the desired throughput when the bandwidth is 10Mbps (the average congestion window size is around 20), its performance degrades even more as the bandwidth reduces. Hence, application striping fails to *scale* to large weights when the available fair share per TCB is such that flows are prone to timeouts.

In summary, although application striping performs better than WAIMD by virtue of its multiple-state strategy, it still has the following limitations:

– **Disproportionate Striping:** When the application performs striping without transport layer support, only simple strategies such as *round-robin striping*, or *write until block* can be used. However, such simple strategies can result in the amount of data assigned to the individual TCP flows being disproportionate with the actual instantaneous data rates of the TCP flows. This will in turn result in out-of-order (in terms of application sequence numbers) packets being delivered at the receiver causing the receive buffers to fill up, and hence potentially resulting in some of the flows stalling due to buffer limitations.

– **Inefficient Buffer Sharing:** While disproportionate striping by itself causes the undesirable phenomenon of more out-of-order delivery at the receiver, the problem is exacerbated due to the inefficient use of the aggregate receive-side buffer. Similar to the problem identified in [22], if the instantaneous rates of the individual TCP flows are different (due to window fluctuations), the constant buffer allocation at the receive side (per TCP flow) is clearly inefficient as the faster flows require a larger portion of the aggregate buffer. However, when application striping is used, no buffer sharing is possible between the individual TCP flows unless an explicit, dynamic buffer re-allocation technique like the one proposed in [22] is used.
– **Timeouts:** Due to the multiple independent states (one per unit TCP) maintained, application striping does not suffer from the drastic consequences that WAIMD faces upon the occurrence of timeouts. However, as explained earlier, the aggregate connection is still vulnerable to stalls because the unaffected TCP flows are also blocked due to unavailability of receive buffer during the course of a timeout. While over-allocation of buffer can reduce the intensity of the problem, it is clearly not a scalable solution.
– **Application Complexity:** Finally, another key drawback of application striping is the additional complexity that the application has to bear in terms of performing intelligent striping, and more importantly performing an elaborate resequencing process at the receiver in order to emulate TCP's in-sequence delivery semantics. Such a resequencing process will involve the use of application layer sequence numbers, intelligent reading from the socket buffer, and efficient maintenance of the application buffer. Requiring every application that needs weighted service differentiation to perform the same set of tasks is clearly undesirable.

4 The pTCP Protocol

pTCP is a multi-state transport layer protocol that maintains multiple TCBs per connection [11]. While it was originally proposed for achieving bandwidth aggregation on multi-homed mobile hosts with heterogeneous network interfaces, we discuss in this section how the basic principles of pTCP, combined with an appropriate tailoring of its mechanisms to the problem of weighted service differentiation (WSD), can help in achieving the desired goal.

4.1 Design Elements

– **Multiple States:** Similar to application striping, pTCP maintains multiple states for a weighted connection to avoid the performance degradation exhibited in WAIMD. A pTCP connection with a weight of *w* consists of *w* TCP-v mini-flows (pipes). The state that each TCP-v maintains (TCB-v) is identical to the TCB maintained by TCP, except that no real application data is stored or manipulated in the TCB-v. Note that while pTCP mimics application striping in maintaining multiple TCBs, the other design elements are targeted toward avoiding the pitfalls in application striping that we identified in Section 3.2.

– **Decoupled Functionality:** pTCP is a wrapper around TCP-v, and it controls the socket send and receive buffers across all TCP-v pipes. Each TCP-v performs congestion control and loss recovery just as regular TCP, but does not have access to the application data and has no control over which data to send. Any segment transmission by a TCP-v (called "virtual segment" that contains only the TCP header) is preceded by a function call to pTCP requesting for data binding. Therefore, TCP-v controls the *amount* of data that can be sent while pTCP controls *which* data to send. A retransmission in the TCP-v pipe does not necessarily mean a retransmission of the application data. pTCP thus decouples congestion control from reliability. By virtue of the design of decoupled functionality, pTCP is able to incur significant less overhead than application striping (we elaborate on this in Section 6.2).
– **Congestion Window Based Striping:** pTCP does not explicitly perform any bandwidth estimation along individual TCP-v pipes to decide the amount of data to distribute across pipes. Instead, pTCP reuses TCP's congestion window adaptation to perform intelligent striping. Specifically, in pTCP the application data is "bound" (assigned) to a TCP-v pipe for transmission only when there is space in its congestion window (recall that TCP-v controls the amount of data to be sent). As bandwidth along each pipe fluctuates, the congestion window of the concerned TCP-v adapts correspondingly, and the amount of data assigned to the pipe also varies, thus achieving packet allocation across multiple pipes. Note that when used for achieving bandwidth aggregation across different paths, the congestion window based striping needs to be complemented with packet scheduling algorithms to tackle the delay differential problem (where different paths exhibit RTT mismatches). However, such a sophisticated and high-overhead packet scheduling algorithm is not required for achieving WSD since all pipes share the same physical path.
– **Dynamic Reassignment:** An important design element in pTCP to avoid head-of-line blocking is the dynamic reassignment of data during congestion. We refer to the process that unbinds application data bound to a virtual segment, so as to rebind it to another virtual segment, as restriping. In [11], pTCP uses a restriping strategy that "blindly" unbinds data bound to virtual segments falling outside the congestion window of a TCP-v pipe that cuts down its congestion window due to congestion or probe losses. It then reassigns such unbound data to pipes that request to send more data. However, for the target environment, a more intelligent restriping strategy that incurs less overhead can be used. Note that since all segments dispatched by pTCP traverse the same physical path, pTCP can leverage the FIFO delivery nature of the single path to perform more intelligent loss detection. Essentially, pTCP can infer a loss as long as it receives 3 or more ACKs for any packets transmitted after the lost packet, irrespective of the specific TCP-v pipes they are bound to. This enables pTCP to "fast reassign" lost packets to another pipe, much before individual TCP-v pipes detect those losses. Therefore, when the fair share of each TCP-v pipe is low, even if individual TCP-v pipes might experience timeouts due to insufficient data to trigger fast retransmit, pTCP will not suffer.
– **Redundant Striping:** Another design element in pTCP to avoid head-of-line blocking is the redundant striping. We refer to the process of binding the same applica-

tion data to more than one virtual segment as redundant striping. In [11], a TCP-v pipe that cuts down its congestion window to one (after a timeout) will be assigned data that can be bound to another TCP-v pipe. In this way, the concerned TCP-v can keep probing for the state of the path it traverses, without stalling the progress of the whole pTCP connection (since it is possible that the path suffering a timeout is currently experiencing a blackout). However, for achieving WSD, we use a different redundant striping strategy. Not that there is no need to redundantly stripe the first segment of the recently-stalled pipe, since all segments experience the same dropping probability irrespective of which pipe they belong to. Instead, pTCP redundantly stripes any segment that is retransmitted due to the fast retransmit mechanism. This is because, in TCP, the loss of a retransmitted segment will cause a timeout. By redundantly striping the retransmitted segment, pTCP ensures no head-of-line blocking will occur even the concerned pipe might eventually experience a timeout. Although redundant striping might appear to be an overhead, the overhead is small when compared to the benefits such striping brings to the aggregate connection in terms of preventing a stall.

– **Simplified Connection Setup:** When used on multi-homed mobile hosts with heterogeneous network interfaces, the peers of a pTCP connection need to exchange the number of interfaces to use and the corresponding IP addresses during connection establishment. As shown in [11], it takes at least two round-trip times to fully establish all TCP-v pipes (the first round-trip time is needed to convey the IP addresses used by subsequent TCP-v pipes). However, for the target environment, all TCP-v pipes of a pTCP connection terminate at the same pair of IP addresses, and hence only a field in the packet header that carries the desired weight (hence the number of TCP-v pipes to open) is needed. The connection setup time thus is the same as a normal TCP connection. Note that in [11] a new IP layer socket hashing function is necessary to map incoming segments with different IP addresses to the same pTCP socket. Such hashing function is not required for achieving WSD.

4.2 Protocol

Fig. 4 provides an architectural overview of the pTCP protocol. pTCP acts as the central engine that interacts with the application and IP. When used with a weight of *w*, pTCP spawns *w* TCP-v pipes. TCP-v is a slightly modified version of TCP that interacts with pTCP using the 7 interface functions shown in Fig. 4. The *open()* and *close()* calls are same as the default TCP ones to enter or exit its state machine [20]. The *established()* and *closed()* interfaces are used by TCP-v to inform pTCP when its state machine reaches the `ESTABLISHED` and `CLOSED` state respectively. The *receive()* call is used by pTCP to deliver virtual segments to TCP-v, and the *send()* call is used by TCP-v to send virtual segments to pTCP which will then bind the segments to real data for transmission. Finally, the *resume()* call is used by pTCP to throttle the amount of data each TCP-v can send.

pTCP controls and maintains the send and receive data buffers for the whole connection. Application data writes are served by pTCP, and the data is copied onto the `send_buffer`. A list of active TCP-v pipes (that have space in the congestion window to transmit) called `active_pipes` is maintained by pTCP. A TCP-v pipe is placed in

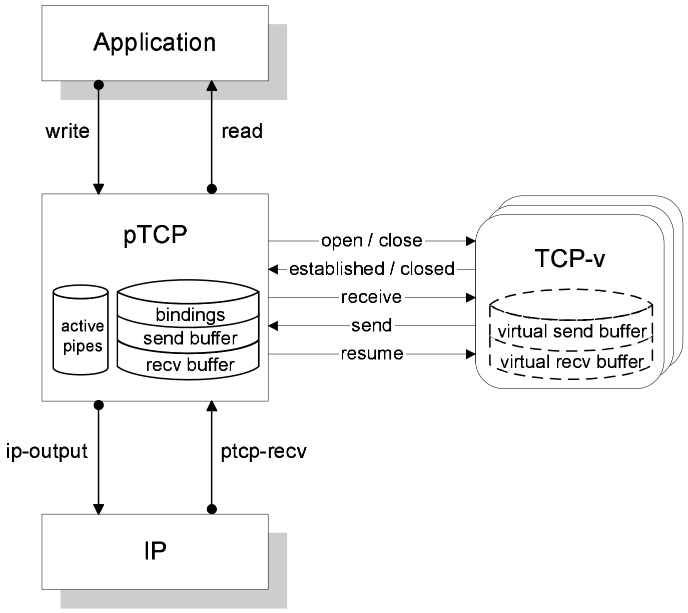


Fig. 4. pTCP Architecture

`active_pipes` initially when it returns with the *established()* function call. Upon the availability of data that needs to be transmitted, pTCP sends a *resume()* command to the active TCP-v pipes, and remove the corresponding pipes from `active_pipes`. A TCP-v pipe that receives the command builds a regular TCP header based on its state variables (e.g. sequence number) and gives the segment (sans the data) to pTCP through the *send()* interface. pTCP binds an unbound data segment in the `send_buffer` to the "virtual" segment TCP-v has built, maintains the binding in the data structure called `bindings`, appends its own header and sends it to the IP layer. A TCP-v pipe continues to issue *send()* calls until there is no more space left in its congestion window, or pTCP responds back with a `FREEZE` value (note that the TCP-v needs to perform a few rollback operations to account for the unsuccessful transmission). When pTCP receives a *send()* call, but has no unbound data left for transmission, it returns a `FREEZE` value to freeze the corresponding TCP-v, and then adds the corresponding pipe to `active_pipes`.

When pTCP receives an *ACK*, it strips the pTCP header, and hands over the packet to the appropriate TCP-v pipe (through the *receive()* interface). The TCP-v pipe processes the *ACK* in the regular fashion, and updates its state variables including the virtual send buffer. The virtual buffer can be thought of as a list of segments that have only appropriate header information. The virtual send and receive buffers are required to ensure regular TCP semantics for congestion control and connection management within each TCP-v pipe. When pTCP receives an incoming data segment, it strips both the pTCP header and the data, enqueues the data in the `recv_buffer`, and provides the appropriate TCP-v with only the skeleton segment that does not contain any data. TCP-v treats the segment as a regular segment, which is then queued in the virtual receive buffer.

5 Performance Evaluation

In this section, we compare the performance of pTCP against that of weighted AIMD and application striping using both simulation and testbed results.

5.1 Single Link Configuration

We first evaluate the performance of pTCP using the network topology and scenario described in Section 3 that we used for evaluating WAIMD and application striping. We observe in Fig. 2(a) that pTCP scales considerably better than WAIMD and application striping with increasing *w*, and follows the ideal curve closely even when *w* scales to 100. While it is clear that WAIMD suffers from its single-state design, pTCP outperforms application striping due to its design elements to prevent head-of-line blocking explained in Section 4.1.

Considering the scalability of pTCP in terms of buffer requirement and bandwidth fair share (i.e. congestion window size), we find in Fig. 3(a) and Fig. 3(b) that pTCP also achieves much better performance than WAIMD and application striping. Note that in Fig. 3(a), the bandwidth-delay product is around 24KB, and hence the ideal (minimum) buffer requirement is 48KB [22]. The reason for the marginal decrease of pTCP in the achieved weight can be explained by the coupling of the individual TCP-v pipes. Although pTCP attempts to mask such coupling to the maximum extent possible, it may still fail in some cases given a tight buffer allocation. However, even when the buffer allocation is significantly reduced, pTCP's performance reduces more gracefully than the other two schemes as shown in Fig. 3(a). Similarly, in Fig. 3(b) we observe that the performance of pTCP does not degrade until the link bandwidth is reduced to 4Mbps at which point the average congestion window size is approximately 8, while both WAIMD and application striping exhibit performance degradation from their already lower performance, at a much earlier point. When the bandwidth is eventually reduced to 3Mbps (the average congestion window size is around 6), individual unit TCP flows become prone to frequent timeouts [21]. However, the performance of the aggregate pTCP connection does not degrade by much even under such circumstances because of its restriping and redundant striping strategy.

5.2 Multiple Link Configuration

In this section, we extend the simulation scenario to a more sophisticated network topology with multiple backbone routers as shown in Fig. 5. The multi-link topology consists of 5 backbone routers and 28 access nodes (access nodes for flows $f10$ to $f13$ are not shown for clarity). Flows $f0$ to $f9$ are "long" flows with round-trip time of 240ms, while flows $f10$ to $f13$ are "short" flows with round-trip time of 90ms and are used as background traffic. As shown in Fig. 5 we also introduce 60 on-off UDP traffic ($c0$ to $c59$) in both directions to emulate the flash crowds in the Internet. Each UDP traffic is generated using the Pareto distribution, where the mean burst time is set to 1s, the mean idle time is set to 2s, the data rate during the burst time is set to 200Kbps, and the shape parameter is set to 1.5. Finally, we introduce another UDP traffic using CBR source on

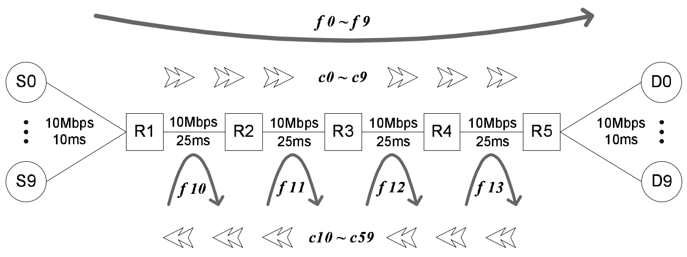


Fig. 5. Network Topology (Multiple Links)

the bottleneck link (from R1 to R2) to explicitly control bandwidth fluctuations experienced by TCP flows along the bottleneck link. The data rate of the background UDP traffic varies from 500Kbps to 3.5Mbps and fluctuates in a 1-second interval throughout the duration of the simulation (600 seconds).

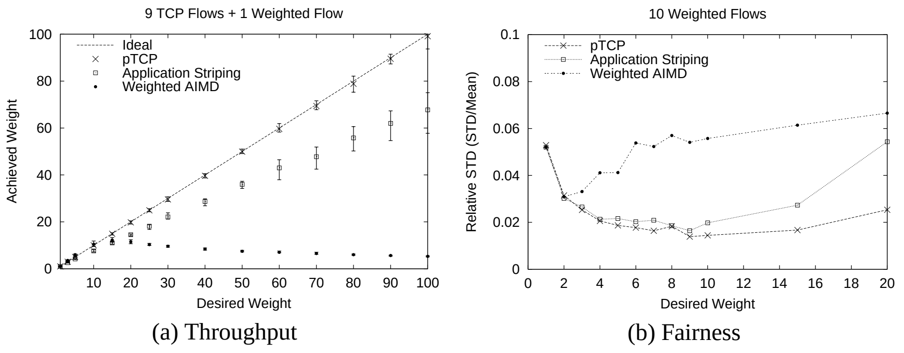


Fig. 6. Performance of pTCP using Multiple Link Topology

In Fig. 6(a) we show the throughput ratio between the weighted flow ($f0$) and the average of the other 9 unit flows ($f1$ to $f9$)–recall that the weighted rate differentiation is with respect to flows along the same path. It is clear that pTCP achieves a much higher scalability than WAIMD and application striping as in the single bottleneck topology, despite the sophisticated traffic distribution and bandwidth fluctuations. In Fig. 6(b), we increase the number of weighted flows to 10 ($f0$ to $f9$), and vary the weight of individual flows from 1 to 20 (all weighted flows have the same weight) to study the fairness property of the 3 schemes. We obtain the throughputs of individual flows, and use the relative standard deviation as a metric to measure the “unfairness” experienced by all flows. As Fig. 6(b) shows, WAIMD has the highest variance for large weights due to frequent timeout occurrences. While application striping achieves similar performance to pTCP when the weight is small, its performance degrades significantly with larger weights. This again is attributed to the head-of-line blocking at the receiver (note that the per-socket buffer requirement in application striping increases with weight).

5.3 Testbed Results

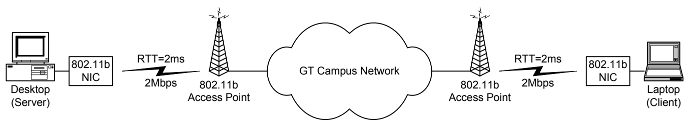


Fig. 7. Testbed Topology

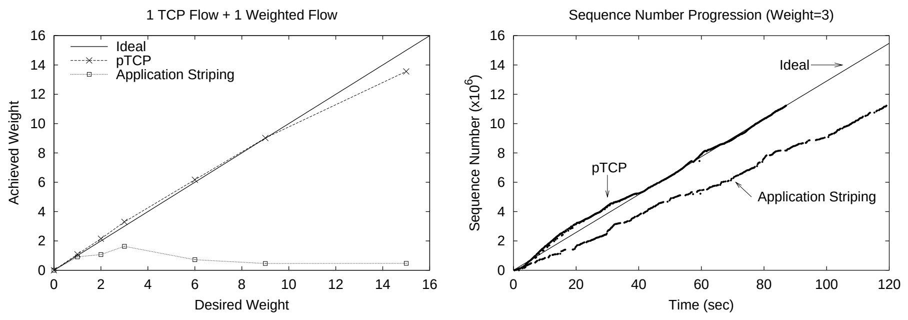
We now present evaluation results for a prototype implementation of pTCP tested over a real-life campus network. We use the campus network shown in Fig. 7 as the testbed. In order to avoid sub-millisecond round-trip times, the client and the server communicate through a campus-wide wireless LAN. The server is a DELL Optiplex GX110 desktop with a Pentium III 733Mhz CPU and 256MB RAM, and the client is an IBM Thinkpad T-20 laptop with a Pentium III 700Mhz CPU and 128MB RAM. Both the client and the server are equipped with the Orinoco IEEE 802.11b network card operating at a data rate of 2Mbps, and run the RedHat 7.3 Linux operating system. The round-trip time between the client and the server is approximately 15ms. The background traffic in the network is not under our control.

We build pTCP upon a user-level implementation of TCP,[3] and follow closely the design and architecture presented in Section 4. The socket interface exported by pTCP resembles that of TCP, except that it allows the specification of the desired weight w. A simple client-server file transfer application is implemented, and the client and the server use pTCP to communicate. We also use a modified version of the file transfer application that explicitly opens w TCP sockets of unit weight and performs striping over the w sockets. We use this to evaluate the performance of application striping, and compare its performance against that of pTCP. We start two FTP applications between the server and the client: one with unit TCP flow and one with weighted flow using either pTCP or application striping.

In Fig. 8(a), we present the ratio of the throughput enjoyed by the weighted flow to the throughput enjoyed by the default TCP flow, for different values of w. We show both the results for pTCP and application striping. The ideal curve is also shown for comparison. It can be observed that pTCP scales significantly better than application striping for all weights, except for $w = 15$, where the ratio observed is 14:1. The reason for the deviation from the ideal behavior is due to the implementation of pTCP at the user-level. The maximum throughput of a user-level process is limited by the CPU scheduling policy in the kernel. Since the server serves the two flows using two independent user-level processes, each process obtains approximately equal share of the CPU cycles. While this does not serve as a limitation for smaller weights, for the weight of 15, the CPU cycles become a bottleneck resulting in the marginally lower performance. We note that such an overhead will not exist in a true kernel implementation.

On the other hand, we observe that the application striping results in Fig. 8(a) are significantly worse than the ideal performance. Upon close inspection, it is determined

[3] We thank Jia-Ru "Jeffrey" Li for sharing his user-level implementation code of TCP with us.



(a) Scalability with weight (b) Sequence number progression

Fig. 8. Testbed Results

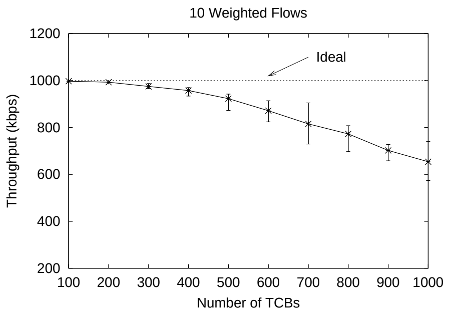
that severe head-of-line blocking is consistently observed due to both wireless channel losses and congestion losses, resulting in the individual flows stalling repeatedly. Fig. 8(b) substantiates this observation where the sequence number plot of application striping exhibits multiple timeouts, and has a much smaller slope than that of the ideal curve. Note that pTCP, although tested in the same wireless environment, does not exhibit performance degradation owing to the design elements described in Section 4.1.

6 Discussion

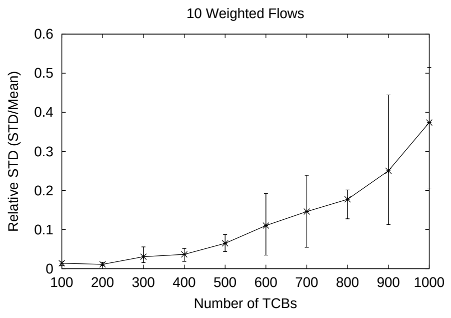
6.1 Scalability Limit

As we have shown in Section 5, pTCP exhibits a much higher scalability than weighted AIMD and application striping. An application that requires weighted rate differentiation of *w* can simply open a pTCP socket with *w* TCP-v pipes and enjoys the desired service (subject to the policing mechanism of the service model). An interesting question that arises is: *what is the upper bound on the maximum weight that pTCP can support for weighted rate differentiation?*

The answer is determined by the *network storage* defined as the sum of the queue length of the bottleneck router and the bandwidth-delay product of the end-to-end path. Since a weighted pTCP flow with weight *w* essentially consists of *w* TCP mini-flows, the maximum weight that pTCP can support is in fact limited by the maximum number of TCP connections the network can support. It is shown in [21] that since TCP's fast retransmit mechanism cannot recover from a packet loss without a timeout if its window size is less than 4 packets, on average TCP must have a minimum send rate of 6 packets per round-trip time. The network hence needs to store roughly 6 packets per active connection, which places an upper limit on the number of connections that can share the bottleneck link. If the number of connections increases beyond this limit, timeouts become the norm and TCP will experience a high degree of delay variation and unfairness. Although pTCP is designed to function even when some of the component mini-flows experience timeouts (pTCP will not stall as long as one of the mini-flow is not stalled), nonetheless it is undesirable to use pTCP in such a scenario.



(a) Throughput



(b) Fairness

Fig. 9. Setting Weight Beyond the Network Limit

To illustrate the phenomenon when the network is operated beyond its limit, we use the same network topology as shown in Fig. 1. We introduce 10 pTCP flows with the same weight to study their throughput performance when the total number of TCBs open is greater than the network limit. For the network considered, the bandwidth-delay product is 300 packets and the router's buffer size is 320 packets, so the network can store at most 620 packets at any instant. If we roughly apply the "6 packets per connection" rule, then the maximum number of TCBs the network can support is around 100. When the number of TCBs used is beyond this limit, Fig. 9(a) shows the average throughput enjoyed by each flow and Fig. 9(b) shows the throughput unfairness among them. We observe that the average throughput degrades and the unfairness increases due to the frequent timeout occurrences. In fact, when the number of connections is greater than network storage (in terms of packets), each connection on average cannot send more than 1 packet per round-trip time, which inevitably will induce timeouts.

A first look into the limitation of maximum weight might indicate that pTCP's design of maintaining one TCB per unit weight causes the problem, since a pTCP flow with weight w needs at least $6w$ packets of storage in the network. However, we contend that for any TCP-based protocol, the maximum weight achievable will indeed be a function of the network storage. Recall that a flow with weight w should achieve throughput w times the throughput of a flow with unit weight. Since TCP has a minimum send rate of 6 packets per round-trip time, the flow with weight w needs to send at the rate of at least $6w$ packets per round-trip time to achieve the desired service differentiation. Hence, as long as the unit flows are TCP-based, the maximum achievable weight for a single flow (or alternately, the maximum of the sum of the weights of contending flows) will be bounded by $\frac{\eta}{6}$, where η is the amount of network storage available.

Note that the design of pTCP by itself does not preclude the use of any other congestion control mechanism that is different from that of TCP. Conceivably, if a congestion control mechanism does not require transmission of at least one packet every round-trip time, the limitation imposed by the network storage on the maximum weight achievable can be eliminated. Such a scalable congestion control scheme can be used in tandem with pTCP through the well-defined interface described in Section 4.2.

6.2 Host Overhead

pTCP, similar to application striping, maintains one TCB per unit weight. This will incur a large overhead in terms of the kernel memory spent at the sender when the weight for a connection is large. However, in pTCP, since the TCP-v pipes merely act on virtual buffers, and act merely as congestion window estimators, the overheads incurred are much lower than in application striping. Moreover, the TCB sharing [23] technique can be used between different TCP-v pipes to further reduce the overheads such as RTT estimation. Another promising approach to reduce the overhead incurred by pTCP is to leverage the scalability of WAIMD for small weights by using a combination of pTCP and WAIMD, where each TCP-v pipe uses WAIMD with a small weight (instead of using the default TCP congestion control with unit weight). Hence, the number of TCP-v pipes used in pTCP to achieve weight w can be reduced to at least $\frac{w}{2}$ (since WAIMD will scale to a weight of 2 easily under all conditions), thus reducing the overheads and complexity in pTCP. Mixing different congestion control schemes within pTCP is possible due to its design of decoupled functionality.

7 Future Work and Conclusions

While we present pTCP as a solution to achieve weighted service differentiation, the pTCP design can have a wider range of applications. Some examples are: (i) Each TCP-v flow can traverse a different path, and hence the performance of pTCP will not be limited to a single bottleneck path. (ii) For an environment where the network can provide absolute QoS assurances, achieving bandwidth aggregation of QoS guarantee service and best-effort service is a non-trivial problem [24]. pTCP can address such a problem by dedicating one TCP-v for the guaranteed service, and one TCP-v for the best-effort service, and effectively providing the application with the desired service.

In this paper, we use a transport layer protocol called pTCP to achieve weighted rate differentiation. A pTCP flow of weight w consists of w TCP-v mini-flows that collectively achieve throughput w times the throughput of a TCP flow. pTCP achieves more effective service differentiation and incurs less host overhead than an approach using multiple sockets. On the other hand, pTCP avoids the pitfalls of the approach that changes the AIMD congestion control of TCP to resemble aggregate TCP flows, i.e. propensity to timeouts and poor scalability. We present evaluation results showing that pTCP is able to achieve end-to-end weighted service differentiation with better scalability and fairness than those presented in related work.

References

1. Dovrolis, C., Ramanathan, P.: A case for relative differentiated services and the proportional differentiation model. IEEE Network **13** (1999) 26–34
2. Banchs, A., Denda, R.: A scalable share differentiation architecture for elastic and real-time traffic. In: Proceedings of IWQoS, Pittsburgh, PA, USA (2000)
3. Crowcroft, J., Oechslin, P.: Differentiated end-to-end internet services using a weighted proportional fair sharing TCP. ACM Computer Communication Review **28** (1998) 53–69

4. Gevros, P., Risso, F., Kirstein, P.: Analysis of a method for differential TCP service. In: Proceedings of IEEE Globecom, Rio de Janeiro, Brazil (1999)
5. Nandagopal, T., Lee, K.W., Li, J.R., Bharghavan, V.: Scalable service differentiation using purely end-to-end mechanisms: Features and limitations. In: Proceedings of IWQoS, Pittsburgh, PA, USA (2000)
6. Postel, J.: Transmission control protocol. IETF RFC 793 (1981)
7. Allman, M., Kruse, H., Ostermann, S.: An application-level solution to TCP's satellite inefficiencies. In: Proceedings of Workshop on Satellite-Based Information Services (WOSBIS), Rye, NY, USA (1996)
8. Sivakumar, H., Bailey, S., Grossman, R.: PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In: Proceedings of IEEE Supercomputing (SC), Dallas, TX, USA (2000)
9. Lee, J., Gunter, D., Tierney, B., Allcock, B., Bester, J., Bresnahan, J., Tuecke, S.: Applied techniques for high bandwidth data transfers across wide area networks. In: Proceedings of Computers in High Energy Physics (CHEP), Beijing, China (2001)
10. Hacker, T., Athey, B., Noble, B.: The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In: Proceedings of IPDPS, Fort Lauderdale, FL, USA (2002)
11. Hsieh, H.Y., Sivakumar, R.: A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In: Proceedings of ACM MobiCom, Atlanta, GA, USA (2002)
12. Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., Weiss, W.: An architecture for differentiated services. IETF RFC 2475 (1998)
13. Dovrolis, C., Stiliadis, D., Ramanathan, P.: Proportional differentiated services: Delay differentiation and packet scheduling. In: Proceedings of ACM SIGCOMM, Cambridge, MA, USA (1999)
14. Nandagopal, T., Venkitaraman, N., Sivakumar, R., Bharghavan, V.: Delay differentiation and adaptation in core stateless networks. In: Proceedings of IEEE INFOCOM, Tel-Aviv, Israel (2000)
15. Shin, J., Kim, J.G., Kim, J.W., Kuo, C.C.: Dynamic QoS mapping framework for relative service differentiation-aware video streaming. European Transactions on Telecommunications **12** (2001) 217–230
16. Kelly, F., Maulloo, A., Tan, D.: Rate control for communication networks: Shadow prices, proportional fairness and stability. Journal of the Operational Research Society **49** (1998) 237–252
17. Massoulie, L., Roberts, J.: Bandwidth sharing: Objectives and algorithms. In: Proceedings of IEEE INFOCOM, New York, NY, USA (1999)
18. Kunniyur, S., Srikant, R.: End-to-end congestion control schemes: Utility functions, random losses and ECN marks. In: Proceedings of IEEE INFOCOM, Tel-Aviv, Israel (2000)
19. The Network Simulator: ns-2. http://www.isi.edu/nsnam/ns (2000)
20. Wright, G.R., Stevens, W.R.: TCP/IP Illustrated, Volume 2. Addison-Wesley Publishing Company, Reading, MA, USA (1997)
21. Morris, R.: Scalable TCP congestion control. In: Proceedings of IEEE INFOCOM, Tel-Aviv, Israel (2000)
22. Semke, J., Mahdavi, J., Mathis, M.: Automatic TCP buffer tuning. In: Proceedings of ACM SIGCOMM, Vancouver, Canada (1998)
23. Touch, J.: TCP control block interdependence. IETF RFC 2140 (1997)
24. Feng, W., Kandlur, D., Saha, D., Shin, K.: Understanding and improving TCP performance over networks with minimum rate guarantees. IEEE/ACM Transactions on Networking **7** (1999) 173–187