TCP mechanisms for Diff-Serv Architecture

Wenjia Fang
wfang@cs.princeton.edu

Larry Peterson
llp@cs.princeton.edu

Department of Computer Science
Princeton University

Abstract

Work on Diff-Serv has demonstrated that it is possible to create differentiations in throughput among TCP connections during periods of network congestion. However, the effectiveness of such schemes is limited by the impreciseness and biases in TCP's window-based congestion control algorithm. More precisely, TCP's window-open mechanism has an intrinsic bias against long RTT connections, and its window-close mechanism adapts to the perceived network congestion optimal point only, which is not sufficient to meet the underlying premise of Diff-Serv architecture. In response to these two weaknesses, this paper proposes a set of new mechanisms for TCP's congestion control algorithm that are specifically tailored to the Diff-Serv architecture. While preserving TCP's "linear increase and multiplicative decrease" principle, these mechanisms make TCP more robust and precise in adjusting its sending rate to network congestion, as well as to a pre-defined service profile. Simulations and testbed implementations are used to qualitatively demonstrate the results. The results show that combined with Diff-Serv mechanisms in routers, the mechanisms in endhosts can allocate resource among TCP connections in a fair, precise and differentiated manner. The paper also discusses incremental deployment issues and proposes a deployment strategy for the proposed mechanisms.

1 Introduction

There has been a flourish of research efforts on Differentiated Services (Diff-Serv) [3] in the past few years. The Diff-Serv architecture distinguishes edge routers from interior routers, and requires only edge routers to maintain per-flow state. The interior routers treat traffic as an aggregate. The differentiation of traffic is achieved as follows. Diff-Serv defines a limited set of Per-Hop Behaviors (PHB) that are implemented in interior routers. Edge routers classify packets into flows and apply traffic conditioning mechanisms to flows, including metering, tagging, shaping, and policing. Edge routers tag packets [15] so as to indicate to the interior routers which PHBs should be applied to the packets. The interior routers do not need to classify packets, but instead apply the corresponding PHBs to packet aggregates. Thus, the service provided to a particular packet flow is a combination of the traffic conditioning at edge routers and a series of PHBs in interior routers.

There are currently two PHBs defined in the Diff-Serv architecture: Expedited Forwarding (EF) [3] and Assured Forwarding (AF) [3]. Expedited Forwarding provides the equivalent of a dedicated link of fixed bandwidth between two edge nodes. Assured Forwarding shares its root with the best effort service model. In AF, each customer specifies its expected level of service (e.g. targeted bandwidth) and the network provides a certain level of assurance in meeting the expectations of the customers [4]. The customer's application can then adapt to the expected service provided by the network. This paper focuses on mechanisms for Assured Forwarding only.

Instrumental to the Diff-Serv architecture is Service Level Agreement (SLA)—a contract between a customer and an Internet Service Provider (ISP) that specifies the forwarding service the customer should receive. The technical part of SLA specifies classifier rules and any corresponding traffic profiles and metering, tagging, shaping and dropping rules to be applied to the traffic streams selected by the classifier. An ISP is expected to provision its network to meet the agreed service requirements in SLA.

The Diff-Serv architecture changes the premise underlying the best-effort Internet service model. In the best-effort service model, congestion causes all participating to slow down. Thus, the received service of a customer depends on how many simultaneous users are currently sharing the bottleneck link. In contrast, the Diff-Serv architecture allocates resources based on some pre-defined policies embodied in the SLAs. With SLAs, an ISP can make better decisions about its provisioning, and customers know the services they expect to receive even during congestion. This implies that the customers can adopt mechanisms that help them to achieve the service they have already subscribed for. This change of premise reflects the now commercialized nature of the Internet. Since high level policies can only be exercised with support from underlying mechanisms, this paper focuses on the mechanisms that end hosts can adopt to achieve their allocated share of service.

Earlier work on Diff-Serv [5] proposed mechanisms in both edge routers and interior routers that, when deployed in a Diff-Serv domain, differentiate among TCP connections. The RIO algorithm is deployed in each interior router to discriminate between two types of packets during congestion: IN packets are those within the subscribed SLAs, and OUT packets are those beyond the subscribed SLAs. The tagging algorithm (TSW) is deployed at edge routers to mark packets as being either IN or OUT of the SLAs. Using simulations, this earlier work shows that such mechanisms can create differentiations among a wide range of TCP connections. However, the effectiveness of such schemes is rather limited by the impreciseness and biases in the window-based congestion control algorithms of TCP. More specifically, the rate adjustment scheme in the current Internet depends on a feedback loop completed by both TCP's congestion control algorithm and the router's congestion signals. Thus, by changing mechanisms in routers alone, the rate adjustment schemes are not very effective or precise in achieving the targeted SLAs.

This paper studies three Diff-Serv mechanisms which, when applied to current TCP's congestion control algorithm, can significantly improve TCP's performance in meeting the requirement of an SLA. Since customers know the SLAs prior to actual communication, the mechanisms we propose incorporate such knowledge as well. We believe the "linear-increase and multiplicative-decrease" principle in the current TCP is sound, and we do not propose to change it. In fact, we do not introduce any additional state variables to TCP's machinery, but merely make the observations that existing variables like *cwnd* and *ssthresh* can be used more effectively in meeting the requirements of SLAs. We use simulations to study these mechanisms in detail, and verify them with a testbed implemention. Our results show that combined with Diff-Serv mechanisms in routers – RIO and TSW –, the proposed TCP mechanisms can allocate resources among end-hosts in a fair, precise and differentiated manner. Furthermore, we discuss in detail the deployment of each mechanism and how to handle partial deployment—a situation in which only a portion of the network has been upgraded.

The idea of changing TCP to incorporate mechanisms that work with the Diff-Serv mechanisms have also appeared in [8], in which, Feng et al. proposed an adaptive marking algorithm to TCP's congestion control algorithm, which interpret and respond to congestion signals from the network. In comparison, our proposed scheme does not introduce additional states to TCP and is very simple to implement. It is applicable to other congestion control algorithms observing the same "linear increase, multiplicative decrease" method of TCP, such as "Congestion Manager" [2].

2 Diff-Serv Mechanisms

2.1 RIO Algorithm

The RIO algorithm is based on the RED (Random Early Drop) algorithm, and is created with two sets of parameters for two types of packets: IN packets and OUT packets. The two sets of parameters are denoted as $(min\_in, max\_in, P_{max\_in})$ and $(min\_out, max\_out, P_{max\_out})$. min_in and max_in are the low and high thresholds for IN packets, and P_{max_in} is the maximum probability with which to drop an IN packet. Similarly, min_out and max_out are the low and high thresholds for OUT packets, and P_{max_in} is the maximum probability with which to drop an OUT packet.

The algorithm works as follows. When a packet arrives, RIO estimates two variables, avg_in_q, average IN packet queue and avg_q, average **total** queue, respectively. An arriving IN packet will contribute to the estimation of avg_in_q, as well as avg_q; an arriving OUT packet will only contribute to the estimation of avg_q. A dropping probability is calculated for each arriving packet depending on the current value of avg_in_q or avg_q. In the case of an IN packet, a dropping probability is calculated as $p = P_{max\_in} * (avg\_in\_q - min\_in)/(max\_in - min\_in)$. The intuition is that an IN packet represents the traffic that is to receive priority, therefore, whether it should be enqueued is dependent on the amount of IN packets the gateway received recently, and not affected by the OUT packets or the total number of packets (both IN and OUT). In the case of an OUT packet, a dropping probability is calculated as $p = P_{max\_out} * (avg\_q - min\_out)/(max\_out - min\_out)$. Since an OUT packet represents the lower priority traffic, it should yield to IN packets in terms of queuing, therefore, its dropping probability depends not only on other OUT packets in the queue but also on the number of IN packets in the queue, therefore, RIO uses avg_q, the average total queue, to calculate the probability for dropping an OUT packet.

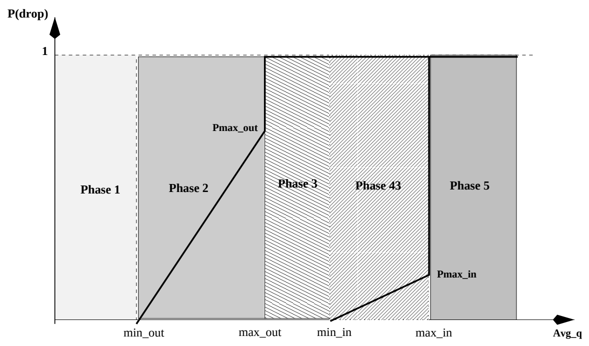


Figure 1: RIO algorithm

Graphically, RIO can be demonstrated in Figure 1. RIO divides up the gateway's congestion state in five phases, depending on the average queue length[1].

- Congestion free phase (phase 1)

  In this phase, the gateway is operating with no congestion: the amount of IN and OUT packets are well below its capacity. It sees very short instantaneous queue and very small average queue value. No packets are dropped.

- Congestion sensitive phase (phase 2)

  In this phase, the gateway suspects that the queue might be built up so it starts to drop packets as congestion signals, however, it drops OUT packets only. During this phase, the IN packets only see short instantaneous queue and they are never dropped.

- Congestion tolerance phase (phase 3)

  In this phase, all OUT packets are dropped, but no IN packets are ever dropped. During this phase, the average queue length is building up with the arriving IN packets. This is the buffering phase for the IN packets before gateways start dropping any IN packets.

- Congestion alarm phase (phase 4)

  In this phase, all OUT packets are dropped, in addition, the gateway starts to drop IN packets as a means to keep the queue from overflowing. This is an undesirable phase for ISP because it compromises the ISP's SLAs by dropping IN packets.

- Congestion control phase (phase 5)

  In this phase, the system is congested. The gateway drops both IN and OUT packets with probability 1. In this phase, the gateway has switched its primary goal from creating differentiations among two types of packets to congestion control. The gateway degrades into a drop-tail gateway, which has other undesirable consequences, e.g., dropping multiple packets from the same TCP stream and global synchronization, etc. If the gateway constantly operates in this phase, it is a sure sign that either the system is well under-provisioned or the parameters of traffic conditioners/RIO are not set correctly.

[1]The X-axis is the number of packets of avg_q, the estimate of average queue length.

Phases 2 and 3 are the ideal operating phases for a router because in this case, both instantaneous and average queue is short but the link is also highly utilized, the only dropped packets are OUT packets, which doesn't compromise the ISP's SLAs. When operating in phase 1, the router sees little congestion but the link capacity is not well utilized. When the input traffic is predictable, ISP should try to configure their system to avoid phases 4 and 5, and operate most in phases 1, 2 and 3.

2.2 Traffic Conditioners and Tagging

Traffic conditioners can be modeled as logical entities sitting on the forwarding path of an edge router. In an edge router, packets are first classified, and then fed through the corresponding traffic conditioners, which can choose to 1) passively monitor packet streams and tag packets, or 2) actively buffer and shape packet streams to obtain certain traffic properties before the packet streams enter the downstream Diff-Serv domain. We consider the simpler case of tagging packets.

Clark and Fang [5] proposed a tagging algorithm, call Time Sliding Window (TSW), that is specifically tailored to TCP traffic. A TSW tagger incorporates a probabilistic function that can reduce the likelihood of tagging consecutive packets within a window of packets, thus, it reduces the chance of multiple packet drops within a window. This keeps TCP operating in the "congestion avoidance" phase, thus making the rate adjustment scheme more controllable. Our simulations show that when TCP itself incorporates mechanisms suited to Diff-Serv, as proposed in the following section, the rate adjustment scheme is less dependent on the intricacies of tagging algorithms. Our results show that we could use a simple Token Bucket tagging scheme with a configured target rate for each TCP connection.

3 Proposed Mechanisms

3.1 TCP Congestion Control

TCP implements congestion control and avoidance mechanisms that interpret packet drops as congestion signals. The mechanisms are based on [13], and have incorporated many refinements [7, 12].

There are two phases in TCP's window adjustment algorithm: the exponential increase (slow start) phase, and the linear increase (congestion avoidance) phase. TCP keeps two variables for its congestion control algorithm: the congestion window (*cwnd*) and the slow-start threshold (*ssthresh*). During the exponential increase phase, the TCP sender starts with a *cwnd* of one packet, and doubles this variable each RTT. When the congestion window hits a threshold, *ssthresh*, the sender switches to the congestion avoidance phase, and increases the congestion window linearly, probing the network capacity as it becomes available. TCP continues in the congestion avoidance phase until it receives a congestion signal—a packet drop or an Explicit Congestion Notification (ECN) in an acknowledgment packet—at which point the sender evokes a mechanism called Fast Retransmit and Fast Recovery to recover the lost packet. Additionally, TCP sets its *ssthresh* to be one half of the congestion window prior to the packet loss, and resets its *cwnd* to be the same as the new *ssthresh*.

In this scheme, *cwnd* indicates the amount of packets currently outstanding, and the instantaneous sending rate of TCP can be approximated as $cwnd/rtt$, where rtt is the round trip time including queuing delays. The choice of threshold reflects an estimation of the equilibrium operating point—a packet leaves the network as a sender puts a packet into the network—and is key to the performance of the algorithm. The algorithm, in adjusting *cwnd*, reflects the additive increase and multiplicative decrease rule which maintains stability in the network [13, 14].

3.2 Proposed Mechanisms

3.2.1 Fair Window Open-Up Algorithm

In both the slow start and congestion avoidance phases, TCP opens its window each round trip time (rtt). Let r_i denote source i's average round trip time, including queuing delays. In the congestion avoidance phase of TCP, node i's window is increased by roughly one packet every r_i seconds. Thus, node i's throughput is increased by $1/r_i$ packets/sec every r_i seconds, or by $1/(r_i)^2$ packets/sec every second. Therefore, it takes a long-rtt connection a significantly longer time to recover to its previous throughput than it does a short-rtt connection.

TCP's bias against long rtt connections has been known and studied in [9]. TCP adopts the current window open up algorithm for its simplicity in algorithm and implementation. However, as far as the fairness goal is concerned, the current increase-by-one window algorithm is not particularly good. We use the fairness index proposed in [14],

$$F = \frac{(\sum_{i=1}^{n} x_i)^2}{n(\sum_{i=1}^{n} x_i^2)}$$

where x_i is the resource allocation to the ith user. This fairness index ranges from 0 to 1, and is maximized when all users receive the same allocation. This index is k/n when k users equally share the resource, and the other $n-k$ users receive zero allocation. Examples of possible definition of resource allocation include response time, throughput, throughput times hops, and so on [14].

We propose two alternative window open-up algorithms, both fall in the categories of linear window open-up algorithm, which meet the criteria of fairness. In the first alternative, TCP increases $c * rtt$ packets per round trip time, where c is a constant, chosen as a scaling factor. Using this scheme, a connection that goes through k bottleneck gateways will share $1/k$ of a bottleneck link bandwidth as a connection which goes through one bottleneck gateway. This will meet the criteria of fairness index when the resource allocation is defined as throughput times the number of gateways. In the second alternative, TCP increases $c * rtt^2$ packets per round trip time. Using this scheme, when n connections are sharing a single bottleneck gateway, the window open-up algorithm allows all connections to each share $1/n$ of the bottleneck bandwidth, regardless of their rtt. This will maximize the fairness index when the resource allocation is defined as throughput of individual connections.

In the Diff-Serv architecture, each entity (potentially at the finest granularity of a single TCP connection) is associated with an SLA that defines a target throughput rate. Though the SLA definition has not been finalized by the IETF Diff-Serv working group, there are two potential definitions to choose from, both of which meet the criteria of fairness. Each definition will in turn determine the underlying window open-up algorithm. In the first definition, an SLA includes both a target throughput as well as a range of rtts within which the target throughput can be met (BW_{target}, (min_{rtt}, max_{rtt})). The longer the rtt, the smaller the corresponding target throughput. This definition of SLA is an interpretation of the fairness index if the underlying TCP window open-up algorithm is chosen to increase $c*rtt$ each round trip time. Alternatively, an SLA includes simply a target throughput (BW_{target}), which implies that the ISP is to assure the target throughput regardless of the rtts of the connection. This definition of SLA is an interpretation of the fairness criteria if the underlying TCP window open-up algorithm is chosen to increase window linearly $c * rtt^2$ each round trip time. The related SLA definitions and their corresponding window open-up policy and fairness criteria are tabulated in Table 1.

Table 1: SLAs and the corresponding TCP mechanisms to achieve fairness

	Policy 1	Policy 2
SLA	$[BW_{target}, (min_{rtt}, max_{rtt})]$	BW_{target}
Definition	Throughput * # of routers	Throughput
Win. Algo.	$c * rtt$	$c * rtt^2$

It should be noted that both alternatives to the current window algorithm of TCP still fall under the "linear increase" rule. Only that the "linear increase" is by c packets per rtt (the first policy), or by c packets per second (the second policy).

3.2.2 Setting *ssthresh* for TCP

As discussed in section 3.1, the value of *ssthresh* reflects the perceived network available bandwidth to a TCP. *ssthresh* is initially set to a default value and is readjusted after each packet drop to be one half of the *cwnd* before the packet drop. A packet drop is recovered either through a mechanism called Fast Retransmit and Fast Recovery, or through a timeout mechanism. When a single packet is lost, the Fast Recovery and Fast Retransmit mechanism recovers the lost packet successfully and both *cwnd* and *ssthresh* are reduced to one half of *cwnd* prior to the packet drop. At that point, TCP continues to operate in the linear window increase phase with a reduced *ssthresh*. When multiple packets are dropped within a window, current implementions of TCP (Reno) usually fail to recover all lost packets because the sender won't be able to put enough packets into the network to generate sufficient duplicated acknowledgments, which is needed to detect additional packet loss. Upon each detected successive packet loss, TCP reduces its *ssthresh* by one half, so when TCP eventually recovers from packet loss via a timeout mechanism, TCP operates with a much reduced *ssthresh*.

In the Diff-Serv architecture, bandwidth allocation is based on SLAs. The underlying premise is that each entity is assured of its target throughput specified in its SLA when congestion is experienced, and can exceed such profiles when there is no congestion. However, there will still be cases when either the ISP fails to provision properly or certain routers experience incipient congestion. In the Diff-Serv domain, when a TCP connection loses a packet, how should *ssthresh* and *cwnd* be set? The underlying Diff-Serv premise implies that the ideal behavior of TCP is to reduce its sending rate when congestion is experienced, but can recover to its target throughput robustly.

We propose the following changes to reflect the change in the underlying premise from a purely best-effort service model to a Diff-Serv model. We set the initial value of *ssthresh* to be the minimum of the default value and the byte-equivalent of the target rate as defined in the SLA. This also "gauges" the operating point of TCP. Additionally, we propose that TCP sets its *ssthresh* to be the byte-equivalent of the target throughput, when congestion is detected. TCP reduces *cwnd* to be one-half of the previous value before the packet drop, as it would in current implementations. This has the effect of reducing instantaneous sending rate of TCP connections to alleviate temporary congestion, but allows each TCP connection to quickly throttle back to its target operating point.

3.2.3 ECN-enabled TCP in Diff-Serv Domain

Some recently proposed changes to TCP include the use of Explicit Congestion Notification (ECN) mechanisms in both TCP and RED gateways [10]. In this proposed scheme, RED routers mark an ECN bit in a packet's header instead of dropping the packet, and TCP responds to the explicit congestion notifications instead of inferring congestion from duplicated acknowledgments. This mechanism has the advantage of avoiding unnecessary packet drops and unnecessary delay for packets from low-bandwidth delay-sensitive TCP connections. A second advantage of the ECN mechanism is that TCP doesn't have to rely on coarse granularity of its clock to retransmit and recover packet losses.

Similar mechanisms could be deployed in the Diff-Serv architecture. Instead of dropping packets, the RIO gateway can also take advantage of the ECN mechanism by marking them. A RIO gateway can apply its preferential algorithm in which it marks an OUT packet as experiencing congestion with a much higher probability than an IN packet [5]. The ECN bit will be copied by the transport-layer receiver and relayed back to the sender. The TCP sender has to be able to recognize the two types of packets (IN and OUT), and respond to ECN bits in them differently.

When an OUT packet arrives back to the TCP sender with the ECN bit marked, it indicates that the RIO gateway is operating in the congestion sensitive phase (phase 2 in Section 2.1). When a RIO gateway deploys the ECN mechanism as the only mechanism for notifying the transport-layer protocols to retract its congestion window, the window reduction should be no more aggressive than the recommended guidelines for ECN mechanisms [10]. We recommend that TCP reduces its *cwnd* to be one half of the current *cwnd* value, and resets *ssthresh* to be the byte-equivalent of the target throughput. Depending on the value of *cwnd* and *ssthresh* prior to receiving the ECN signal, TCP can be operating in either linear increase mode or exponential increase mode again. In either case, the reduction in the window size will induce a temporary reduction in TCP's sending rate to alleviate congestion, but still keep TCP operating close in the targeted operating point.

When an IN packet arrives back to the TCP sender with the ECN bit marked, it indicates that the RIO gateway operates in the congestion control phase (phase 5 in Section 2.1), meaning that the gateway has seen persistent long queues and is forced to mark both IN and OUT packets with probability 1. When such packet is received, the TCP sender should react to the congestion signal more drastically. We recommend that TCP reduces its reduce its *cwnd* to be one packet, and reset *ssthresh* to be the byte-equivalent of the target throughput. This is the same window reaction as in the current implementation when a packet has been dropped but TCP starts in its slow start phase with a configured *ssthresh*. This will cause a more drastic reduction in TCP's sending rate, but since the new *ssthresh* will be greater than the new *cwnd*, TCP will quickly recover to the target operating point using exponential increase window increase algorithm.

4 Simulation and Testbed Results

This section presents simulation results and some preliminary results from a testbed implementation. We find that when endhost incorporates Diff-Serv mechanisms to respond to feedbacks from the network (both congestion information and IN/OUT information), the combined scheme is precise and effective in allocating network capacities among connections with different SLAs.

4.1 Simulation Setup

We use the network simulator *ns* [1], with a simple topology (Figure 2) to evaluate bulk-data transfers. We use six FTP transfers with two sets of rtts: 80ms and 30ms. Each simulation run has four different phases. The first phase is the *start-up* phase in which all six FTP/TCP connections reach their respective operating points. The second phase is a *congested* phase, in which, a constant bit rate (CBR) connection starts, running at $1/4$ of the bottleneck bandwidth. This will cause heavy congestion in the router and TCP connections will back off during this phase. The third phase is the *recovery* phase, in which the CBR source stops and all FTP/TCP connections will recover to their respective operating points. The fourth phase is the *over-provisioned* case, during which, one of the FTP/TCP connections (TCP1) stops sending, and the available bandwidth is shared among the rest of five FTP/TCP sources. Each individual phase lasts for 25 seconds. All packet sizes are set to 1000 bytes. We use TCP-Reno, and receiver windows are large enough to not be a constrain on the congestion window.

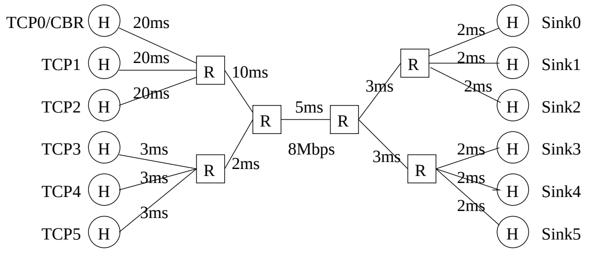


Figure 2: Simulation Topology

Table 2: Configurations of TCP connections

	RTT (ms)	R_t (Mbps)
TCP0	80	2
TCP1	80	2
TCP2	80	1
TCP3	30	1
TCP4	30	0.6
TCP5	30	0.6
CBR	80	2

The parameters for RED and RIO gateways are set comparably. The bottleneck speed is 8Mbps. The low threshold (min_th) for RED is the byte-equivalent of 5ms of queue delay, the high threshold (max_th) is the byte-equivalent of 10ms of queuing delay,[2] and the dropping probability P_max is 0.1. The comparable parameters for RIO are (5, 10, 0.5) for OUT packets, and (10, 20, 0.02) for IN packets. To save space, we use tables to represent the time average throughput of three representative connections during different phases. Each setup is run three times with a different random seed, and the data presented in the tables are averages of the three runs. For each scenario, we show the throughput of 1) a long-rtt FTP/TCP (with and w/o a target throughput of 2Mbps); 2) a short-rtt FTP/TCP (with and w/o a target throughput of 0.6Mbps); 3) a CBR connection with sending rate at 2Mbps during the congested phase. The constant c in TCP's window open-up algorithm is chosen to be 100, which is equivalent of increase one packet each 100ms.

The total allocated throughput is 7.2Mbps, or 90% of the bottleneck link. The details of simulation set up are listed in Table 2.

4.2 Impact of Mechanisms

We separate the mechanisms into two groups: Diff-Serv mechanisms to be applied in the end hosts (combinations of all the mechanisms proposed in Section 3) and Diff-Serv mechanisms to be applied in the router (RIO and TSW algorithms). We consider four different scenarios: 1) standard TCP-reno algorithm with RED gateways; 2) Diff-Serv enhanced TCP with RED gateways; 3) standard TCP with RIO and TSW gateways; and 4) Diff-Serv enhanced TCP with RIO and TSW gateways. Table 3 lists the results from four different scenarios.

Scenario 1 is our basis for comparison, representing the current best effort model. It illustrates two well-known behaviors: 1) short-RTT TCP connections have advantage over long-RTT connections when sharing the same bottleneck (first body row vs. second body row); and 2) a non-congestion controlled source has a detrimental effect on TCP connections (second body column), in which, TCP0 and TCP3 throughput dropped by 30% when CBR starts. In this case, the CBR source gets almost all its packets through a RED gateway at the expenses of other TCP connections' throughput.

Scenario 2 illustrates the effect of the mechanisms incorporated into TCP. With configured knowledge of target throughputs, TCP could robustly recover to its target rate after packet losses. The proposed window open-up algorithm also corrects the bias against long-RTT connections, e.g., in the Start-up and Recovery phases, TCP0, with an *rtt* of 80ms, doesn't suffer from network bias and gets close to its allocate target rate (1.86Mbps or 93%). However, in the presence of a non-congestion controlled source, all TCP sources will suffer, e.g., a drop in TCP0 and TCP3's throughput (30%) when CBR starts. The RED gateway is not capable in discriminating against an "out-of-profile" source.

Scenario 3 shows the results of applying the mechanisms in the routers only. Compared to scenario 2, the RIO algorithm discriminates against "out-of-profile" sources to limit the detrimental effect OUT packets have on IN packets during congestion. In this case, the CBR source is getting 89% of its packets through vs. 96% of its packets in scenario 2. (The bottleneck link has enough available bandwidth to accommodate 50% of the CBR packets.) The service differentiation among TCP connections with varying RTTs is the most pronounced during congestion (body column 2). When the network is well-provisioned, the service discrimination effect of RIO is dampened by the TCP's window algorithm. Short-RTT connections obtain most of the available bandwidth in the over-provisioned situation. In other words, when free of congestion, the innate TCP biases can override the targeted bandwidth allocation created by the Diff-Serv mechanisms in routers.

Scenario 4 illustrates the effects of the mechanisms in both the end host TCP and in routers. Compared to scenario 2, the improvement lies in the congested phase, in which the RIO algorithm is able to shield IN packets from the interference of OUT packets. In this case, the CBR source is able to get 50% of its packets through (body column 2), which is roughly what the router can accommodate besides all its pre-allocated resources. Compared to scenario 3, the improvement lies in allocation of bandwidth according to each connection's profile regardless its *rtt* and the network conditions. When the network is congested, each TCP receives close to its targeted throughput; when the network is well-provisioned, the allocation of extra available bandwidth is fair among all TCP connections.

In summary, we observe that by incorporating Diff-Serv mechanisms in endhosts, the combined scheme can allocate resources fairly, precisely and differentially among connections, regardless of network conditions. In fact, if the endhost TCP has incorporated the Diff-Serv mechanisms, the RIO algorithm in routers can be configured to create strong differentiation among classes of packets,

[2] In simulations, we translate this in terms of the number of packets queued.

Table 3: Comparison of Diff-Serv mechanisms applied to routers and endhost TCP; Modified TCP = Standard TCP + Three mechanisms. All measured in Mbps

		Start-Up phase	Congested Phase	Recovery Phase	Over-provision Phase
Standard TCP+RED (Scenario1)	TCP0 (80ms, no target)	0.676768	0.491638	0.723149	0.832894
	TCP3 (30ms, no target)	1.622382	1.126404	1.585279	1.804911
	CBR		1.978168		
Modified TCP+RED (Scenario2)	TCP0 (80ms R_t=2Mbps)	1.86133	1.31369	1.81553	2.30319
	TCP3 (30ms R_t=1Mbps)	1.11268	0.84987	1.12360	1.42987
	CBR		1.92003		
Standard TCP +RIO+TSW (Scenario3)	TCP0 (80ms R_t=2Mbps)	1.43707	1.32511	1.40382	1.49129
	TCP3 (30ms R_t=1Mbps)	1.05836	0.90249	1.11443	1.37187
	CBR		1.78891		
Modified TCP RIO+TSW (Scenario4)	TCP0 (80ms R_t=2Mbps)	2.02678	1.89689	2.02658	2.36111
	TCP3 (30ms R_t=1Mbps)	1.04109	0.91049	1.04853	1.33992
	CBR		1.00350		

therefore, more effectively shield traffic that within SLAs from those that are outside SLAs.

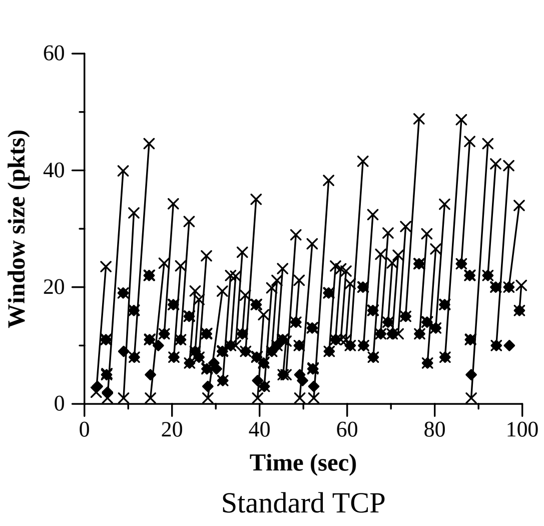
4.3 Robust Recovery from Losses

This section focuses on the details of TCP's window behaviors before and after incorporating the Diff-Serv mechanisms. We illustrate the effects in Figure 3. The left graph shows TCP0's *cwnd* and *ssthresh* throughout the entire 100 seconds of simulation (scenario 1 setup in Table 3, in which TCP uses standard Reno algorithm). The right graph shows TCP0's *cwnd* and *ssthresh* throughout time (scenario 4 setup in Table 3, in which TCP incorporates all three Diff-Serv mechanisms). The most pronounced and visible difference lies in how *ssthresh* is adjusted in the two graphs: in the left graph, the *ssthresh* adjustment is according to the perceived network conditions and can be drastic and unpredictable. For example, from time 25 to 50 seconds, when there is a CBR source keeping the networks in a congested state, the TCP sources usually detect this and run at a much reduced operating point. There are several cases in which *ssthresh* is adjusted multiple times, each for a packet drop within the same window. (Not visible given the granularity of the graph.) From time 80 second and onwards, the network is in a over-provisioned state, and the rate adjustments are infrequent and the *ssthresh* is high. In contrast, in the right graph, the *ssthresh* is set by the targeted throughput, so after a packet drop, TCP's *cwnd* is reduced but not its *ssthresh*. Note that *ssthresh* is adjusted if the estimated RTT changes, because the *ssthresh* is set to be byte-equivalent of target-rate delay product. This is shown in the graph as a few discreet values of *ssthresh*: 40, 45 and 50 packets, etc. By keeping the *ssthresh* near its target operating point, TCP can quickly recover from its packet losses and not being affected by worsened network conditions caused by non-congest control sources.

Another difference between the two graphs lies in the rate at which TCP adjusts its window, or the slope of each discrete segment of TCP window adjustments. During the congestion control phase of window increase, the enhanced TCP using a constant c of 100 opens up its window *slower* than its counterpart before incorporating the Diff-Serv mechanisms. The right graph appears to have a fast rate of increase because most of the time, it operates in the "Slow Start" phase after a packet drop because *ssthresh* is greater than *cwnd*. This is also a sign that TCP is not meeting its targeted throughput. On the other hand, in the left graph, TCP operates in the congestion avoidance phase after a packet drop because *ssthresh* and *cwnd* are both readjusted.

4.4 Testbed Implementation

We have implemented the first two mechanisms (TCP window open-up algorithm and setting *ssthresh*) in a testbed. The testbed currently has edge routers that implement the TSW tagging algorithms, and a RIO algorithm with three dropping preferences, conforming to the Diff-Serv WG standard. The endhosts use Linux RedHat 2.3.39 distribution, which has the standard TCP-Reno algorithms. We incorporated the first two mechanisms in endhost kernel, and run some initial test experiments. By the time of paper



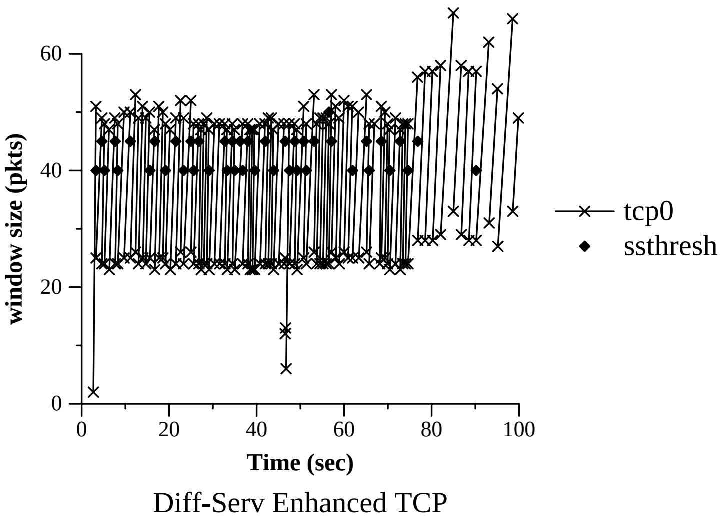


Figure 3: TCP window algorithm before and after incorporating Diff-Serv mechanism

Table 4: Effect of C in a testbed environment (throughput measured as Mbps)

	TCP0 (30ms)	TCP1 (80ms)
Both using Reno	2.7	2.0
C=50	2.7	2.0
C=100	2.5	2.3
C=200	2.4	2.4
C=500	2.3	2.6
C=1000	2.2	2.7

submission, we have only conducted a few simple test experiments. This is on-going work and we will have some new results to report later this year.

In a simple test case to study the effect of constant *c*, we have two TCP connections, one has incorporated the new mechanisms and another doesn't. Both are sharing a 5Mbps bottleneck connection. The standard TCP (TCP0) connection has an RTT of 30ms, and the TCP connection with new mechanisms (TCP1) has an RTT of 80ms. When both TCPs use TCP-reno, we observe the network bias against long-rtt connections (2.7Mbps for TCP0 and 2.2Mbps for TCP1). Then we configure TCP1 with increasingly large value of *c* and observe the effect of having an increasingly aggressive window open-up algorithm by changing the value of *c*. TCP1, with the new window algorithm, can gradually overcome the network bias. Eventually, the effect of an aggressive window open-up algorithm (*c* = 1000) is limited because the actual sending window is limited by the receiver's window, instead of the congestion window. The results are summarized in Table 4.

5 Discussion

5.1 Choice of c

Another way of viewing the change in TCP's linear window open-up algorithm is the following: instead of increasing TCP's congestion window by one packet each round trip time, the proposed mechanism opens up *cwnd* by one packet during a certain standard unit of time. If all TCP implementations adopt such algorithm, then they will all increase their window at the same rate regardless of their rtts. Thus, the choice of c, which determines the value of such standard unit of time is a crucial one. For example, if c is chosen to be 100, then, the standard unit of time is implicitly set to be 100ms $(100 * (0.1)^2 = 1pkt)$. In other words, all TCP implementing the above proposed mechanism will be increasing their congestion windows at the same rate as a current TCP implementation with an rtt of 100ms. Essentially, this algorithm make those TCP connections with rtt less than 100ms less aggressive than the current implementations, and those with rtt greater than

Table 5: Choice of c in TCP fair window algo.

rtt range	Constant C	Equivalent rtt
(0, 50ms)	1024	31.2ms
(50,100ms)	256	62.5ms
(100,200ms)	64	125ms
(200ms, ∞)	4	500ms

100ms more aggressive than the current implementations.

There are two potential problems arise from this. The first is how to choose a value that can be universally agreed upon. The technical merits of the proposed mechanism have been argued, but the ultimate choice lies in the policies by which the choice of c makes sense. One problem of choosing a relatively small c (less than 100ms, for example) is that for long-rtt connections, the new algorithm will result in an effective rate increase even greater than that during the slow start phase. For example, if a 1sec TCP connection uses the proposed algorithm, it means it will open up its window at the rate of one packet each 100ms, which is 10 packets each rtt. Depending on the current number of packets outstanding, this rate can be greater than that during the slow start phase. The problem with choosing a relatively large c is that this makes TCP window increase algorithm very slow and if this algorithm is universally adopted, it might result in low utilization of link bandwidth immediately after a congestion epoch.

One possible solution is to define a set of inclusive rtt ranges, and within which, modified TCP connections will open up their window at the same rate, but each range has a different window open rate. A reasonable heuristic is that the longer rtt, the slower the window increase rate is because the longer the connection, the more resources (buffer space, or packets in the pipe) it would take. Such ranges of rtts can be easily specified in the SLAs as the ISP will set a lower expected throughput for longer connections. The range of rtts can be chosen to reflect actual market concerns. For example, we could define four ranges of RTTs, inclusive. (0, 50ms) for LANs and WAN range of connections; (50ms, 100ms) for intra-continental connections; (100ms, 200ms) for inter-continental connections; and (200ms, ∞) for non-tether connections. Of course, such policies have to be universally agreed upon and standardized. These ranges define the particular algorithm and the corresponding values for c.

Another problem with the choice of c lies in incremental deployment of such algorithm. When TCPs with different implementations operate in a heterogeneous environment, TCPs observing the fair algorithms might be at a disadvantage. Fortunately, Diff-Serv router mechanisms offer a solution for migrating TCPs to the fair algorithms. See Section 6.2 for a detailed discussion of this point.

5.2 Interactions with Tagging Algorithms

In [5], Clark and Fang propose specific tagging algorithms for entities using TCP as a transport layer protocol, and used a probabilistic function to reduce the likelihood of multiple packets being tagged and dropped within a TCP window. We find through our simulations that when TCP itself has incorporated Diff-Serv mechanisms, the end-to-end performance relies less on the intricacies and accuracies of tagging algorithms. TCP would perform well without using a probabilistic tagging function. This switches the role of tagging schemes in edge routers from tagging a TCP connection accurately to managing a number of connections sharing a common SLA. This is the topic of future research.

6 Deployment Issues

6.1 Backward Compatibility

Among the above three proposed mechanisms, the first and second mechanisms require only the TCP sender to change its window adjustment algorithm, and does not require the receiver's cooperation.

The second mechanism need some policy servers that keep information about SLAs, and additionally, a signaling protocol for communicating between the transport layer at the end host and the edge router if the profile is changing in real time. The information kept in policy servers is used to configure TCP with its initial *ssthresh* value and the *ssthresh* value after each packet drop.

The third mechanism requires TCP to be aware of the IN/OUT bit (or TOS field) of the IP header. This mechanism can be deployed at the same time as the ECN field. The mechanism works as follows: a TCP sender always sends out packets with IN/OUT bit as "OFF". A packet goes through a traffic conditioner, which in turn tags the packet's TOS field as either "ON" or "OFF". A RIO and ECN capable gateway will mark packets differentially, and turn on ECN field for those packets if necessary. The transport layer at the receiver side has to copy both the ECN field and the TOS field of the IP header in the due acknowledgment packet. The sending TCP will react to a packet with both ECN and TOS bits (an IN packet) set differently from that with only ECN bit set (an OUT packet).

Table 6: Heterogeneous Deployment of TCP mechanisms, measured in Mbps. **Mech1** is the fair window open up algorithm, **new** include all three mechanisms

		Start-Up phase	Congested Phase	Recovery Phase	Over-provision Phase
Standard TCP+RED (Scenario1)	TCP0 (80ms, Reno)	0.676768	0.491638	0.723149	0.832894
	TCP3 (30ms, Reno)	1.622382	1.126404	1.585279	1.804911
	TCP5 (30ms, Reno)	1.541346	1.122553	1.610749	1.850088
	CBR		1.978168		
Mixed TCP algorithms +RED (Scenario2)	TCP0 (80ms, w/ mech1)	0.851694	0.499243	0.898498	0.90222
	TCP3 (30ms, w/ mech1)	0.950140	0.584215	0.792326	1.3719
	TCP5 (30ms, Reno)	1.893473	1.462454	1.845942	2.018788
	CBR		1.986283		
Uniform TCP algorithms +TSW+RIO (Scenario3)	TCP0 (80ms, R_t=2, new)	2.02678	1.89689	2.02658	2.36111
	TCP3 (30ms, R_t=1, new)	1.04109	0.91049	1.04853	1.33992
	TCP5 (30ms, R_t=0.6, new)	0.659625	0.533941	0.629245	0.969653
	CBR		1.00350		
Mixed TCP algorithms +TSW+RIO (Scenario4)	TCP0 (80ms,R_t=2, new)	1.984425	1.917876	1.991106	2.18545
	TCP3 (30ms,R_t=1, new)	0.993548	0.924187	0.991756	1.182006
	TCP5 (30ms, R_t=0.6, Reno)	0.602984	0.424578	0.591179	0.940206
	CBR		1.151985		

6.2 Heterogeneous Environments

Among the mechanisms we proposed, the first mechanism has been studied in a context of improving fairness for TCP connections with varying rtts [11]. One important problem pointed out by [11] lies not in the algorithm itself, but its interaction with the standard TCP algorithm when they both exist in a heterogeneous network environment. As discussed before, the fair algorithm makes all TCP connections open up their windows at the same rate. With a chosen constant c corresponding to some standard unit of time, this algorithms makes any TCP connections with rtt shorter than the standard unit *less aggressive* than their current implementation, and any TCP connections with rtt longer than the standard unit *more aggressive* than their current implementations. As a result, if two TCP implementations co-exist in a heterogeneous network environment and their rtts are both shorter than the standard unit of rtt, the connection with the current implementation will be more aggressive than the connection with the fair algorithm implementation. This takes away any incentives for people to deploy the fair algorithm [3].

The first half of the Table 6 illustrates this case. We include another 30ms TCP connection (TCP5). Scenario 1 is the case when all TCPs use the standard algorithm and RED is used by routers as the queuing discipline. The two 30ms TCP connections have a clear advantage over the 80ms TCP connection, as expected from the current TCP window algorithm. Scenario 2 illustrates the case when TCP0 and TCP3 have upgraded to use the new and fair window algorithm whereas TCP5 remains the same. The constant c is chosen to be 100, which makes both TCP0 and TCP3 *less* aggressive than their counterparts in scenario 1. We see that TCP0 and TCP3 achieve comparable results, (0.85Mbps and 0.95Mbps) whereas TCP5 has gained an advantage over both (1.89Mbps). TCP0 performs slightly better than its counterpart in scenario 1 (0.67Mbps), but TCP3 performs much worse (1.62Mbps).

Fortunately, we find that the Diff-Serv mechanisms in routers can be used to assist in such migration. We find that when the Diff-Serv router mechanisms are deployed first and TCPs incorporate all three proposed mechanisms, the allocation of bandwidth is according their respective SLAs (for those TCPs which have respective SLAs), and there is no clear advantage for standard TCP over enhanced TCP. Scenarios 3 and 4 in Table 6 illustrate this. In scenario 3, all TCPs have upgraded to incorporate the Diff-Serv mechanisms, and the allocation of resources is according to their respective service profiles regardless the state of the network. When the network is over-provisioned, the available bandwidth is equally distributed among all connections. In

[3]Of course, connections with rtt longer than the standard unit rtt will be more aggressive than their current implementation, and there would be incentives for people to deploy such algorithm.

scenario 4, TCP4 (not shown) and TCP5 both use the standard TCP window open-up algorithm. The results show that there is no clear advantage of the current TCP algorithm over the fair TCP algorithm in the Diff-Serv environment. This preserves the incentives for customers to update their TCP algorithms to incorporate the fair algorithm.

7 Conclusions

The rate adjustment scheme in the current Internet relies on congestion control mechanisms in both transport-layer TCP and congestion signals in gateways. When the premise of resource allocation has changed from the best-effort model of the current Internet to a defined-service model in Diff-Serv architecture, the underlying mechanisms have to be changed to support it as well.

Early work in Diff-Serv have mostly focused on the mechanisms to be deployed in routers to provide differentiations among TCP connections. A logical extension of that is to devise mechanisms to be deployed in transport-layer TCP to support the change in premise. This is the focus of our paper. Without introducing any new state variables into the existing TCP machinery, we propose three simple mechanisms to make TCP operate significantly better in a Diff-Serv domain. These mechanisms preserve the "multiplicative decrease and linear increase" principle of the TCP congestion control algorithm and can be applied to similar congestion control algorithms preserving the same principle. We use simulations and testbed implementations to qualitatively verifying our ideas. Our study shows that the proposed mechanisms complement the Diff-Serv router mechanisms in changing the rate adjustment scheme of the current Internet. By incorporating those mechanisms in endhosts, the combined scheme can allocate resources fairly, precisely and differentially among connections. We also discuss the partial deployment issue.

The study reported in this paper can be extended in a number of ways. We are currently working on implementations and experiments on a testbed. One possible future direction is to investigate whether router mechanisms (TSW and RIO) can be combined and implemented as a collective admission control mechanism in edge routers. This is related to how an ISP would configure and provision its network given the information from all its contracted SLAs. Another possibility is to devise TCP Diff-Serv mechanisms for other proposed Diff-Serv services, e.g., Proportional Diff-Serv [6], or to study how well the proposed mechanisms will integrate with other proposed Diff-Serv queueing disciplines.

References

[1] *Ns network simulator*. Available via http://www-nrg.ee.lbl.gov/ns/.

[2] BALAKRISHNAN, H., RAHUL, H., AND SESHAN, S. An integrated congestion management architecture for internet hosts. In *Proceedings of SIGCOMM '99* (1999), vol. 29.

[3] BLACK, D., BLAKE, S., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An architecture for differentiated services. In *IETF RFC 2475*. IETF, December 1998.

[4] CLARK, D. D. Internet cost allocation and pricing. In *Internet Economics* (1997), J. B. L. McKnight, Ed., MIT Press, pp. 215–253.

[5] CLARK, D. D., AND FANG, W. Explicit allocation of best effort packet delivery service. *IEEE/ACM Transactions on Networking 6*, 4 (1998).

[6] DOVROLIS, C., STILIADIS, D., AND RAMANATHAN, P. Proportional differentiated services. In *Proceedings of SIGCOMM* (October 1999), vol. 29.

[7] FALL, K., AND FLOYD, S. Simulation-based comparisons of tahoe, reno and sack tcp. In *Computer Communications Review* (July 1996), vol. 26, pp. 5–21.

[8] FENG, W., KANDLUR, D., SAHA, D., AND SHIN, K. Underderstanding and improving tcp performance over networks with minimum rate guarantees. *IEEE/ACM Transactions on Networking 7*, 2 (April 1999), 173–187.

[9] FLOYD, S. Connections with multiple congested gateways in packet-switched netwokrs part 1: One-way traffic. *Computer Communication Review 21*, 5 (October 1991), 30–47.

[10] FLOYD, S. Tcp and explicit congestion notification. In *Computer Communication Review* (October 1995), vol. 24.

[11] HENDERSON, T. R., SAHOURIA, E., MCCANNE, S., AND KATZ, R. On improving the fairness of tcp congestion avoidance. In *Proceedings of IEEE Globecom '98* (Sydney, 1998).

[12] HOE, J. Improving the start-up behaviors of a congestion control scheme for tcp. In *Proceedings of ACM SIGCOMM* (Stanford, CA, 1996).

[13] JACOBSON, V. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM* (Stanford, CA, 1988).

[14] JAIN, R., CHIU, D., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared systems. Tech. rep., Digital Equipment Corporation, 1984.

[15] NICHOLS, K., BLAKE, S., BAKER, F., AND BLACK, D. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. In *IETF RFC 2474*. IETF, December 1998.