A Compound TCP Approach for High-speed and Long Distance Networks

Kun Tan Jingmin Song Qian Zhang
Microsoft Research Asia
Beijing, China
{kuntan, t-jmsong, qianz}@microsoft.com

Murari Sridharan
Microsoft Corporation
One Microsoft way, Redmond, WA, USA
muraris@microsoft.com

***Abstract*—Many applications require fast data transfer over high speed and long distance networks. However, standard TCP fails to fully utilize the network capacity due to the limitation in its conservative congestion control (CC) algorithm. Some works have been proposed to improve the connection's throughput by adopting more aggressive loss-based CC algorithms. These algorithms, although can effectively improve the link utilization, have the weakness of poor RTT fairness. Further, they may severely decrease the performance of regular TCP flows that traverse the same network path. On the other hand, pure delay-based approaches that improve the throughput in high-speed networks may not work well in an environment, where the background traffic is mixed with both delay-based and greedy loss-based flows. In this paper, we propose a novel Compound TCP (CTCP) approach, which is a synergy of delay-based and loss-based approach. In CTCP, we add a scalable delay-based component into the standard TCP Reno congestion avoidance algorithm (i.e., the loss-based component). The sending rate of CTCP is controlled by both components. This new delay-based component can rapidly increase sending rate when network path is under utilized, but gracefully retreat in a busy network when bottleneck queue is built. Augmented with this delay-based component, CTCP provides very good bandwidth scalability with improved RTT fairness, and at the same time achieves good TCP-fairness, irrelevant to the windows size. We developed an analytical model of CTCP and implemented it on the Windows operating system. Our analysis and experiment results verify the properties of CTCP.**

Index Terms—TCP performance, delay-based congestion control, high speed network

I. INTRODUCTION

Moving bulk data quickly over high-speed data network is a requirement for many applications. For example, the physicists at CERN LHC conduct physics experiments that generate gigabytes of data per second, which are required to be shared among other scientists around the world. Currently, most of the applications use the Transmission Control Protocol (TCP) to transmit data over the Internet. TCP provides reliable data transmission with embedded congestion control algorithm [1] which effectively removes congestion collapses in the Internet by adjusting the sending rate according to the available bandwidth of the network. However, although TCP achieves remarkable success (maximizing the utilization of the link and fairly sharing bandwidth between competing flows) in the today's Internet environment, it has been reported that TCP substantially underutilizes network bandwidth over high-speed and long distance networks [3].

In high-speed and long distance networks, TCP requires a very large window, roughly equal to the *bandwidth delay production* (BDP), to efficiently utilize the network resource. However, the standard TCP takes a very conservative approach to update its window in congestion avoidance stage. Specifically, TCP increases its congestion window by one packet in every round trip time (RTT) and reduces it by half at a loss event. If BDP is too large, it requires an unreasonable time for TCP to expand its window to that value. As an example, Sally et. al. [3], pointed out that under a 10Gbps link with 100ms delay, it will roughly take one hour for a standard TCP flow to fully utilize the link capacity, if no packet is lost or corrupted. This one hour error free transmission requires a packet loss ratio around 10^{-11} using 1500-byte packets (one packet loss over 2,600,000,000 packet transmissions!). This requirement is far from reality of current network hardware.

Recent research has proposed many approaches to address this issue. One class of approaches modifies the increase/decrease parameters of TCP congestion avoidance algorithm (CAA) and makes it more aggressive. Like TCP, approaches in this category are loss-based that uses packet-loss as the only indication of congestion. Some typical proposals include HSTCP [3], STCP [4] [5], and BIC-TCP [6]. Another class of approaches, by contrast, is delay-based, which makes congestion decisions that reduce the transmission rate based on RTT variations, e.g., FAST TCP [5]. All aforementioned approaches are shown to overcome TCP's deficiencies in high bandwidth-delay networks pretty well to some extent. However, in this paper, we argue that for a new high-speed protocol, following requirements must be met before it can be really deployed into the Internet:

[Efficiency] It must improve the throughput of the connection to efficiently use the high-speed network link.

[RTT fairness] It must also have good intra-protocol fairness, especially when the competing flows have different RTTs.

[TCP fairness] It must not reduce the performance of other regular TCP flows competing on the same path. This means that the high-speed protocols should only make better use of free available bandwidth, but not steal bandwidth from other flows.

For existing loss-based high-speed solutions, it is essential to be highly aggressive to satisfy the efficiency requirement. However, this aggressiveness also causes severe RTT unfairness and TCP unfairness. On the other hand, for delay-based approaches, although they can achieve high efficiency and good RTT fairness in a network where the majority flows are delay-based, it is difficult to meet the third requirement if most competing flows

are loss-based, e.g. TCP-Reno. In this situation, the delay-based approaches may suffer from significant lower throughput than their fair share. The reason is that delay-based approaches reduce their sending rate when bottleneck queue is built to avoid self-induced packet losses. However, this behavior will encourage loss-based flows to increase their sending rate as they may observe less packet losses. As a consequence, the loss-based flows will obtain much more bandwidth than their share and delay-based flows may even be starved [9]

In this paper, we propose a new congestion control protocol for high-speed and long delay environment that satisfies all aforementioned three requirements. Our new protocol is a synergy of both delay-based and loss-based congestion avoidance approaches, which we call it *Compound TCP* (CTCP). The key idea of CTCP is to add a scalable delay-based component to standard TCP[1]. This delay-based component has a scalable window increasing rule that not only can efficiently use the link capacity, but can also react early to congestion by sensing the changes in RTT. If a bottleneck queue is sensed, the delay-based component gracefully reduces the sending rate. This way, CTCP achieves good RTT fairness and TCP fairness. We have developed analytical model of CTCP and performed comprehensive performance studies on CTCP based on our implementation on Microsoft Windows platform. Our analysis and experimental results suggest that CTCP is a promising algorithm to achieve high link utilization and while maintaining good RTT fairness and TCP fairness.

The rest of paper is organized as follows. In next section, we elaborate the background and existing approaches in detail. We review these schemes against the three properties we mentioned before. Then, we propose our design of CTCP in Section III. Analytical analysis of CTCP is presented in Section IV. We describe the implementation in Section V, and experiment results of CTCP are presented in Section VI. We conclude the paper in Section VII.

II. Background And Related Work

The standard TCP congestion avoidance algorithm employs an *additive increase and multiplicative decrease* (AIMD) scheme. When there is no packet loss detected (by means of three duplicate-ACKs or retransmission timeout), the *congestion window* (cwnd) is increased by one Maximum Segment Size (MSS) every RTT. Otherwise, if a packet loss is detected, the TCP sender decreases cwnd by half. In a high-speed and long delay network, it requires a very large window, e.g. thousands of packets, to fully utilize the link capacity. Therefore, it will take the standard TCP many RTTs to recover the sending rate upon a single loss event. Moreover, it is well-known now that the average TCP *congestion window* is inversely proportional to the square root of the packet loss rate, as shown in (1) [8][12],

$$W = \frac{1.22 \cdot \mathrm{MSS}}{\sqrt{p}}, \qquad (1)$$

where W is the average TCP window and p is the average packet loss rate. It requires extremely small packet loss rate to sustain a large window. With the packet loss rate in real life networks, a standard TCP sender may never open its window large enough to fully utilize the high-speed link resource.

One straightforward way to overcome this limitation is to modify TCP's increase/decrease control rule in its congestion avoidance stage. More specifically, in the absence of packet loss, the sender increases cwnd more quickly and decreases it more gently upon a packet loss.

STCP [4] alters TCP's AIMD congestion avoidance scheme to MIMD (*multiplicative increase and multiplicative decrease*). Specifically, STCP increases cwnd by 0.01 MSS on every received ACK and reduces cwnd to its 0.875 times upon a packet loss. HSTCP [3], on the other hand, still mimics the AIMD scheme, but with varying increase/decrease parameters. As cwnd increases from 38 packets to 83,333 packets, the decrease parameter reduces from 0.5 to 0.1, while the increase parameter increases accordingly. HSTCP is less aggressive than STCP, but is far more aggressive than the standard TCP. As pointed in [6], these aggressive schemes suffer great RTT unfairness. From simple analysis based on synchronized loss model, Lisong *et. al.*, [6] shows that for any loss-based congestion control protocol with steady state throughput in the form of $Th = \frac{C}{R \cdot p^{d}}$, where R is the RTT, p is the packet loss rate, C and d are constant, the throughput ratio between two flows with different RTT should be inversely proportional to 1/(1-d)th power of their RTT ratio. Or, formally, $\frac{Th_2}{Th_1} = \left(\frac{R_1}{R_2}\right)^{\frac{1}{1-d}}$. This value d for HSTCP and STCP is 0.82 and 1, respectively. Thus, the RTT unfairness of HSTCP and STCP is 5.56 and infinite, respectively.

In [6], BIC-TCP is proposed to mitigate this RTT unfairness by using binary increase scheme and switching to AIMD with constant parameters when cwnd is large. BIC-TCP has similar RTT-fairness to TCP-Reno when the window is large and then the sender has switched to AIMD mode (window > 16000 packet and in this time d is near 0.5). However, if the protocol works at lower window range, d could increase to near 1 and BIC-TCP may have similar RTT unfairness to STCP.

Besides the RTT-unfairness, the above approaches also have TCP-unfairness concerns when they are working in a mixed network environment with standard TCP flows and these enhanced flows. When an aggressive high-speed variant flow traverses the bottleneck link with other standard TCP flows, it may increase its own share of bandwidth by reducing the throughput of other competing TCP flows. The reason is that the aggressive high-speed variants will cause much more self-induced packet losses on bottleneck links, and therefore push back the throughput of the regular TCP flows. Indeed, there is a mechanism present in the existing aggressive high-speed variants to revert to standard TCP congestion avoidance algorithm if the window is smaller than a pre-defined threshold (*low_window* as defined in [3], which is typically set to 38 packets). This, although may prevent collapses in heavy congested cases, can not provide satisfactory TCP-fairness in general network situations. The aggressive behavior of these enhanced flows may severely de-

[1] In this paper, terms of "regular TCP" and "standard TCP" all refer to TCP-Reno.

grade the performance of regular TCP flows whenever the network path is already highly utilized. As an illustration, we conducted the following simulation in NS 2 [22]. The network topology was a simple dumbbell topology, as shown in Figure 1. The capacity of the bottleneck link was set to 100Mbps and the round-trip delay was 100ms. We tested the cases where one enhanced high-speed flow (i.e. HSTCP, STCP, BIC-TCP[2]) was competing with five regular TCP flows. We also compared with the case where there were six identical regular TCP flows. The results are summarized in Figure 2. With this network setup, six regular TCP flows already fully utilized the network path. As clearly shown in Figure 2, all the three aggressive high-speed flows occupied over 60% of link bandwidth in its testing, and the throughput of a regular TCP was reduced to only 30% compared to that if there was no aggressive high-speed flow. In other words, each aggressive high-speed flow increased its throughput only by reducing throughputs of other regular TCP flows, and therefore caused TCP unfairness. Note that BIC-TCP caused even severe TCP-unfairness than HSTCP in this case. According to [6], BIC-TCP may have better TCP-fairness property than HSTCP when the window is large. But in the window size in this example, the aggressiveness of BIC-TCP was between of HSTCP and STCP.

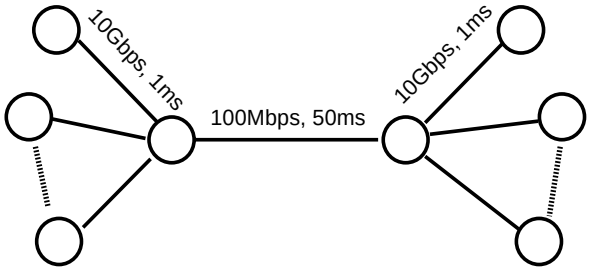

Figure 1. The network topology with simple bottleneck link.

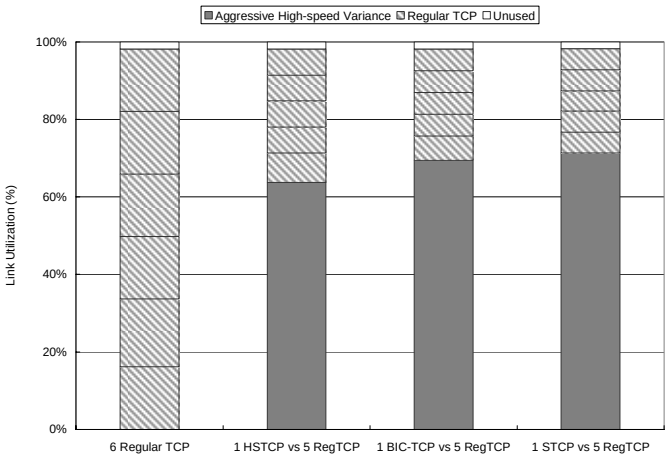

Figure 2. Bandwidth allocation between competing flows.

Another class of high speed protocols, like FAST TCP [5], instead of making simple modifications on TCP's increase/decrease parameters, chooses to design a new congestion control scheme which takes RTT variances as congestion indicator. These delay-based approaches are more-or-less derived from seminal work of TCP Vegas [11]. One core idea of delay-based congestion avoidance is that the increase of RTT is considered as early congestion, and the sending rate is reduced to avoid self-induced buffer overflow. Another interpretation of this delay-based behavior is that it tries to maintain a fixed buffer occupation. In this way, they will not cause large queueing delay and reduce packet losses. FAST TCP can be regarded as a scaled version of TCP Vegas. FAST TCP incorporates multiplicative increase if the buffer occupied by the connection at the bottleneck is far less than some pre-defined threshold α, and switch to linear increase if it is near α. Then, FAST tries to maintain the buffer occupancy around α and reduces sending rate if delay is further increased. Theoretical analysis and experiments show that delay-based approaches have better properties than pure loss-based approaches, such as higher utilization, less self-induced packet losses, faster convergence speed, better RTT fairness and stabilization [16][5]. However, previous work also reveals that delay-based approaches may not be able to obtain fair share when they are competing with loss-based approaches like standard TCP [9]. This can be explained as follows. Consider a delay-based flow, e.g. Vegas or FAST, shares a bottleneck link with a standard TCP. Since the delay-based flow tries to maintain a small number of packets in the bottleneck queue, it will stop increasing its sending rate when the delay reaches some value. However, the loss-based flow will not react to the increase of delay, and continues to increase the sending rate. This, observed by the delay-based flow, is considered as congestion indication and therefore the sending rate of the delay-based flow is further reduced. In this way, the delay-based flow may obtain far less bandwidth than its fair share. One possible way to remedy this is to design a dynamic scheme to choose α to ensure it is TCP-compatible [26]. However, designing such a scheme is simply not trivial, since the correct α is a function of buffer size and number of concurrent connections, which are generally unknown in a real world network. To our best knowledge, this is no design of such a dynamic scheme for this purpose.

Besides the protocols discussed above, there is recently a proposal of TCP Africa [27]. Similar to CTCP, it also proposes to incorporate delay information to improve the RTT and TCP fairness. However, the key difference of this proposal to CTCP lies in that, in TCP Africa, the delay-information is used only as a trigger to switch TCP Africa from "fast mode" (aggressive increase) and "slow mode" (additive increase), but never used to reduce the window size. Therefore, TCP Africa is intrinsically a loss-based approach. When competing with other regular TCP flows, it will still steal much bandwidth from them, though better than HSTCP. As we will show later in our design and experiments, reducing window based on delay information is essential to prevent stealing bandwidth from other regular TCP flows in a mixed network environment (see SectionVI.B.6)).

III. THE COMPOUND TCP

After understanding the existing approaches and their limitations, we revisit the design of a high-speed congestion control algorithm that fulfills all three requirements listed in Section I.

[2] We chose the parameters of BIC-TCP according to [6].

The key idea is that if the link is under-utilized, the high-speed protocol should be aggressive in increasing sending rate to obtain available bandwidth more quickly. However, once the link is fully utilized, being aggressive is no longer good, as it will only cause problems like TCP unfairness. We note that delay-based approaches already have this nice property of adjusting its aggressiveness based on the link utilization, which is observed by the end-systems from the increase in the packet delay. However, as mentioned in previous section, the major weakness of delay-based approaches is that they are not competitive to loss-based approaches. And this weakness is difficult to be remedied by delay-based approaches themselves.

Having made this observation, we propose to adopt a synergic way that combines a loss-based approach with a delay-based approach for high speed congestion control. For easy understanding, let's imagine application *A* communicates application *B* simultaneously using two flows. One is a standard loss-based TCP flow, and the other is a delay-based flow. When the network is underutilized, *A* can get an aggregated communication throughput, with *B*, which is the sum of both flows. With the increase of the sending rate, queue is built at the bottleneck, and the delay-based flow gradually reduces its sending rate. The aggregated throughput for the communication also gradually reduces but is bound by the standard TCP flow.

Then, there comes the core idea of our novel *Compound TCP* (CTCP), which incorporates a scalable delay-based component into the standard TCP congestion avoidance algorithm. This scalable delay-based component has a rapid window increase rule when the network is sensed to be under-utilized and gracefully reduces the sending rate once the bottleneck queue is built. With this delay-based component as an auto-tuning knob, *Compound TCP* can satisfy all three requirements pretty well:

1) CTCP can efficiently use the network resource and achieve high link utilization. In theory, CTCP can be very fast to obtain free network bandwidth, by adopting a rapid increase rule in the delay-based component, e.g. multiplicative increase. However, in this paper, we choose CTCP to have similar aggressiveness to obtain available bandwidth as HSTCP. The reasons are two-fold. On one hand, HSTCP has been tested to be aggressive enough in real world networks and is now an experimental IETF RFC. On the other hand, we want to bound the worst case behavior of CTCP as HSTCP if the network is poorly buffered, as we will elaborate in Section VI.B.5).

2) CTCP has similar or even improved RTT fairness compared to regular TCP. This is due to the delay-based component employed in the CTCP congestion avoidance algorithm. It is known that delay-based flow, e.g. Vegas, has better RTT fairness than the standard TCP [19].

3) CTCP has good TCP-fairness. By employing the delay-based component, CTCP can gracefully reduce the sending rate when the link is fully utilized. In this way, a CTCP flow will not cause more self-induced packet losses than a standard TCP flow, and therefore maintains fairness to other competing regular TCP flows.

A. Architecture

As explained earlier, CTCP is a synergy of a delay-based approach with a loss-based approach. This synergy is implemented by adding a new scalable delay-based component in the standard TCP congestion avoidance algorithm (also called loss-based component). To do so, a new state variable is introduced in current TCP Control Block (TCB), namely, *dwnd* (Delay Window), which controls this delay-based component in CTCP. The conventional congestion window, *cwnd*, remains untouched, which controls the loss-based component in CTCP. Then, the CTCP sending window now is controlled by both *cwnd* and *dwnd*. Specifically, the TCP sending window (called *window* hereafter) is now calculated as follows:

$$win = \min(cwnd + dwnd, awnd),$$

where *awnd* is the advertised window from the receiver.

The update of *dwnd* will be elaborated in detail in next sub-section, while the update of *cwnd* is in the same way as in the regular TCP in the congestion avoidance phase, i.e., *cwnd* is increased by one MSS every RTT and halved upon a packet loss event. However, here CTCP may send (*cwnd +dwnd)* packets in one RTT. Therefore, the increment of *cwnd* on arrival of an ACK is modified accordingly:

$$cwnd = cwnd + 1/(cwnd + dwnd). \quad (2)$$

CTCP keeps the same Slow-Start behavior of regular TCP at the start-up of a new connection. It is because that we believe slow-start, which exponentially increases window, is quick enough even for fast and long distance environment that we target at [3]. We initially set *dwnd* to zero if the connection is in slow-start state, and the delay-based component is effective only when the connection is working at congestion avoidance phase.

B. Design of delay-based congestion avoidance

In this sub-section, we focus on the algorithm used in the aforementioned delay-based component in CTCP, which is enabled when the connection is in congestion avoidance phase. This delay-based algorithm should have the following properties. Firstly, it should have an aggressive, scalable increase rule when the network is sensed to be under-utilized. Secondly, it should also reduce sending rate accordingly when the network is sensed to be fully utilized. By reducing its sending rate, the delay-based component yields ways for competing TCP flows to ensure TCP fairness of CTCP. Lastly, it should also react to packet losses. It is because packet losses may still be an indicator of heavy congestion, and hence reducing sending rate upon packet loss is a necessary conservative behavior to avoid congestion collapse.

Our algorithm for delay-based component is derived from TCP Vegas. A state variable, called *baseRTT*, is maintained as an estimation of the transmission delay of a packet over the network path. When the connection is started, *baseRTT* is updated by the minimal RTT that has been observed so far. An exponentially smoothed current RTT, *sRTT*, is also maintained. Note that both *baseRTT* and *sRTT* should be of high resolution. Then, the number of backlogged packets of the connection can be estimated by following algorithm:

$$Expected = win / baseRTT$$
$$Actual = win / RTT$$
$$Diff = (Expected - Actual) \cdot baseRTT$$

The *Expected* gives the estimation of throughput we get if we do not overrun the network path. The *Actual* stands for the throughput we really get. Then, *(Expected – Actual)* is the difference between the expected throughput and the actual throughput. When multiplying by *baseRTT*, it stands for the amount of data that injected into the network in last round but does not pass through the network in this round, i.e. the amount of data backlogged in the bottleneck queue. An early congestion is detected if the number of packets in the queue is larger than a threshold γ. If $diff < \gamma$, the network path is determined as under-utilized; otherwise, the network path is considered as busy and delay-based component should gracefully reduce its window.

Note here, it requires the connection to have at least γ packets backlogged in the bottleneck queue to detect early congestion. Therefore, we want γ to be small, since it requires less buffer size on the bottleneck to ensure TCP fairness. On the other hand, we can not set γ to be too small, since it may cause false detection on early congestion and adversely affect the throughput. In this paper, we set γ to be 30 packets. As we will show in Section VI.B.4), we believe this γ value is a pretty good tradeoff between TCP fairness and throughput.

The increase law of the delay-based component should make CTCP more scalable in high-speed and long delay pipes. In this paper, we choose the CTCP window evolution to have the binomial behavior. More specifically, when there is no congestion occurs, neither increase in queue nor packet losses, the CTCP window increases as follows

$$win(t+1) = win(t) + \alpha \cdot win(t)^k; \quad (3)$$

while there is a lose, the window is multiplicatively decreased,

$$win(t+1) = win(t) \cdot (1-\beta). \quad (4)$$

Parameters of α, β and k are tunable to give out desirable scalability, smoothness and responsiveness. As we mentioned before, we tune CTCP to have comparable scalability to HSTCP when there is absence of congestion (the detailed derivation is presented in Section IV.A).

Considering there is already a loss-based component in CTCP, the delay-based component needs to be designed to only fill the gap, and the overall CTCP should follows the behavior defined in (3) and (4). We summarize the algorithm for the delay-based component as in (5)

$$dwnd(t+1) = \begin{cases} dwnd(t) + (\alpha \cdot win(t)^k - 1)^+, \text{if } diff < \gamma \\ (dwnd(t) - \zeta \cdot diff)^+, \text{if } diff \geq \gamma \\ (win(t) \cdot (1-\beta) - cwnd/2)^+, \text{if loss is detected} \end{cases}, \quad (5)$$

where $(.)^+$ is defined as max (., 0). The first line shows that in increase phase, *dwnd* only needs to increase $(\alpha \cdot win(t)^k - 1)^+$ packets, since the loss-based component (*cwnd*) will also increase by 1 packet. Similarly, when there is a loss, *dwnd* is set to the difference between the desired reduced window size and that can be provided by *cwnd*. The rule on the second line is important. It shows that *dwnd* does decrease when the queue is built, and this is the core for CTCP to preserve good RTT and TCP fairness. Here, ζ is a parameter that defines how rapidly the delay-based component should reduce this window when early congestion is detected. Note that *dwnd* will never be negative. Therefore, CTCP window is low-bounded by its loss-based component (i.e. a standard TCP).

In above control laws, we assume the loss is detected by three duplicate ACKs. If a retransmission timeout occurs, *dwnd* should be reset to zero and the delay-based component is disabled. It is because that after a timeout, the TCP sender is put into slow-start state. After the CTCP sender exits the slow-start recovery state, the delay-based component may be enabled once more.

IV. Analysis of CTCP

In this section, we develop analytical models of CTCP to study its characteristics. More specifically, we want to quantify how well CTCP satisfies the three requirements proposed earlier for high speed protocols, namely efficiency property, RTT fairness property, and TCP fairness property. In the following analysis, we use a synchronized loss model. There is much evidence showing that synchronized loss is common in high-speed networks [6]. We develop our analytic model based on a simple network topology which contains one bottleneck link, as shown in Figure 3, where *u* stands for the link capacity; *B* is the buffer size on the bottleneck link; and *Ti* is the transmission delay. Note that this transmission delay can be different for different connections.

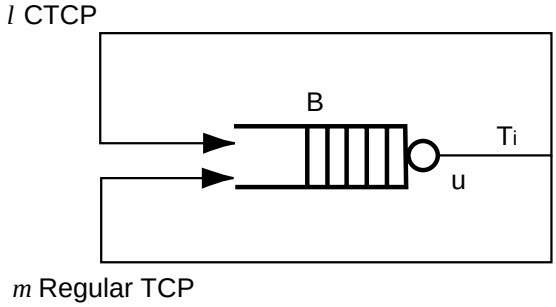

Figure 3. A simple network model.

A. Efficiency property

We first focus on the stable state throughput of CTCP. We ignore the slow-start phase and assume all loss events are detected by three duplicate ACKs. We follow the common assumptions as in previous work [12][19] that packet losses in different rounds are independent and RTT is independent to the window size of individual flow. The evolution of the CTCP window is illustrated in Figure 4. At time *D*, the connection just experiences a loss event and reduces the window size. At this point, the queue at the bottle-neck has been drained, and therefore, the window increases binomially. At time *E*, the queue begins to build, the CTCP senses this from the increase in the packet delay, and *dwnd* decreases to try to maintain a stable window. Note that *cwnd* still increases one MSS each round and when *dwnd* decreases to zero, CTCP window increases again at time *F*. CTCP continues to probe the network capacity until a packet loss occurs at time *G*, and window is reduced by βW. The dashed line in Figure 4 illustrates the evolution of *cwnd* in CTCP. From time *D* to time *G*, we define a Loss Free Period

(LFP).

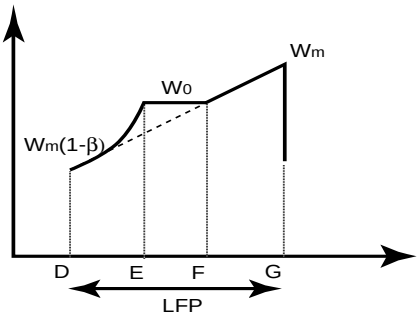

Figure 4. The evolution of CTCP window during a LFP.

We denote w_0 the window size at the stable state, i.e. from time *E* to time *F*, at which CTCP tries to maintain γ packets backlogged at the bottleneck queue. Following the previous analysis [19], the average *diff* in this stable state is approximately γ. Therefore, we have

$$w_0 = \gamma \cdot \frac{R}{R - baseRTT}, \tag{6}$$

where *R* is the round trip time.

Below, we develop the throughput model of CTCP. During time *D* to *E*, window increases according to

$$\frac{dW}{dt} = \frac{\alpha \cdot W^k}{R} \Rightarrow \frac{W^{(1-k)}}{1-k} = \frac{\alpha \cdot t}{R}. \tag{7}$$

Therefore, the interval between *D* and *E* can be calculated as

$$T_{DE} = t_E - t_D = \frac{R}{\alpha(1-k)}\left[W_0^{(1-k)} - (1-\beta)^{(1-k)} W_m^{(1-k)}\right]. \tag{8}$$

And the number of packets transmitted during time interval *D*-*E* is

$$\begin{aligned} N_{DE} &= \int_{t_D}^{t_E} \frac{W}{R} dt = \left(\frac{\alpha(1-k)}{R}\right)^{\left(\frac{1}{1-k}\right)} \cdot \frac{1}{R} \int_{t_D}^{t_E} t^{\frac{1}{1-k}} dt \\ &= \frac{1}{\alpha(2-k)}\left(W_0^{(2-k)} - (1-\beta)^{2-k} \; W_m^{(2-k)}\right) \end{aligned}. \tag{9}$$

During time *E* to *F*, while *cwnd* continues to grow, *dwnd* keeps dropping. As a consequence, the CTCP window remains stable until *dwnd* drops to zero and window starts again to increase at time *F*. Therefore, we have

$$T_{EF} = T_{FD} - T_{DE} = (W_0 - (1-\beta)W_m)R - \frac{R}{\alpha(1-k)}\left[W_0^{(1-k)} - (1-\beta)^{(1-k)} W_m^{(1-k)}\right], \tag{10}$$

and $N_{EF} = W_0 \cdot T_{EF}$. (11)

During time *F* to *G*, CTCP window increases linearly just as if it is a simple TCP Reno. Therefore, we simply have

$$T_{FG} = (W_m - W_0)R, \tag{12}$$

and $N_{FG} = \frac{1}{2}(W_0 + W_m)(W_m - W_0) = \frac{1}{2}(W_m^2 - W_0^2)$ (13)

The total number of packet sent in one LPF is

$$Y = N_{DE} + N_{EF} + N_{FG} \approx 1/p. \tag{14}$$

Note that W_m is the only variable in equation (14), so that it can be got by solving (14). Then, we plug W_m back into (8), (10), and (12) to get the total number of RTTs in each LPF. Finally, the throughput of CTCP is $\Lambda = \frac{1/p}{R \cdot (T_{DE} + T_{EF} + T_{FG})}$. Below, we give the close-form expression of CTCP throughput in a special case, from where we give the rational on how to choose CTCP parameters.

We assume the packet loss rate is rather high compared to the link speed. In this case, random packet loss limits the throughput of CTCP and the stable state mentioned above is never achieved. The response function in this case reflexes the ability of CTCP to use the free bandwidth. Since LFP ends before time *E*, from (9), we have

$$\frac{1}{\alpha(2-k)}\left[W_m^{(2-k)} - (1-\beta)^{(2-k)} W_m^{(2-k)}\right] = \frac{1}{p}. \tag{15}$$

So, now we get

$$W_m = \left[\frac{\alpha(2-k)}{\left(1-(1-\beta)^{(2-k)}\right)}\right]^{\frac{1}{2-k}} \cdot \frac{1}{p^{\frac{1}{2-k}}}. \tag{16}$$

Therefore, we have the throughput of CTCP as

$$\begin{aligned} \Lambda &= \frac{1/p \cdot \alpha(1-k)}{R \cdot W_m^{1-k} \cdot \left(1-(1-\beta)^{1-k}\right)} \\ &= \frac{1}{R \cdot \alpha^{\frac{1}{2-k}} \cdot \left(1-(1-\beta)^{1-k}\right)}\left[\frac{1-(1-\beta)^{2-k}}{2-k}\right]^{\frac{1-k}{2-k}} \cdot \frac{1}{p^{\frac{1}{2-k}}} \end{aligned}. \tag{17}$$

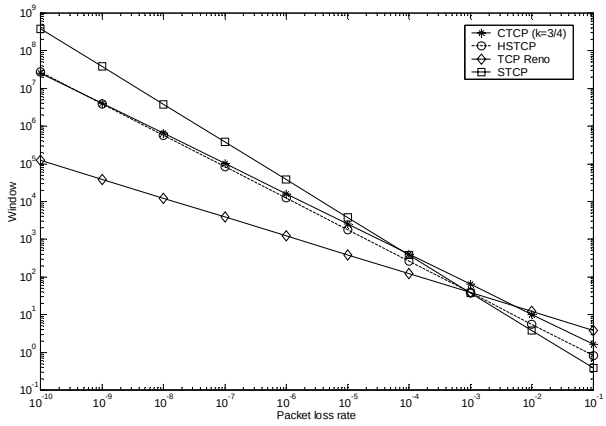

Figure 5. The response functions.

Note that *k* determines the slope of response function (aggressiveness). As we mentioned before, we intend to let CTCP have the similar ability to HSTCP in a congestion-free network. By compared (17) to the response function of HSTCP, whose window is $W_{HSTCP} \propto \frac{1}{p^{0.833}}$. Therefore, we can get *k* by solving the following equation, $\frac{1}{2-k} = 0.833 \Rightarrow k \approx 0.8$. However, it is rather difficult to implement an arbitrary power calculation using integer algorithm. Therefore, we choose *k* equal to 0.75, which can be implemented with fast integer algorithm of square root. α and β is a trade-off between smoothness and the responsiveness. In this paper, we choose α =1/8, β =1/2. We plot the CTCP response function in log-log scale, in Figure 5, as well as the response functions of the standard TCP, STCP and HSTCP. From the figure, we can see that CTCP is slightly more aggres-

sive than HSTCP in moderate and light packet loss rate and is approaching to HSTCP when window is large.

B. Convergence and RTT fairness

We first demonstrate that two CTCP flows with same RTT converge to their fair shares and then study its RTT fairness.

Theorem 1: Two CTCP flows converge to fair share under the network model as shown in Figure 3 with same round trip delay.

Proof. We use Jain's fairness index [23], which is defined as follows,

$$F(x) = \frac{(\sum x_i)^2}{n(\sum x_i^2)}.$$

Let's consider two CTCP flows, whose window size are *x1* and *x2*, respectively. We assume *x1* < *x2*. We see $\Delta F \geq 0$, if and only if $\Delta x_1 / x_1 \geq \Delta x_2 / x_2$. In increasing phase, $\Delta x = \alpha \cdot x^k$ where *k<1*, it is easy to see that $1/x_1^{(1-k)} > 1/x_2^{(1-k)}$, and therefore, $\Delta F > 0$. When the queuing delay is detected, *dwnd* is decreasing while *cwnd* still increases by 1MSS. Therefore,

$$\Delta x = 1 - \zeta diff = 1 - \zeta x(1 - \frac{baseRTT}{RTT})baseRTT.$$

We see that when the two streams share the same bottleneck and the propagation delay is same, the *baseRTT* observed by two streams should also converge to the same value. Therefore, $\frac{\Delta x_1}{x_1} = \frac{1}{x_1} - \frac{\zeta \cdot diff_1}{x_1} = \frac{1}{x_1} - \zeta \cdot (1 - \frac{baseRTT}{RTT})baseRTT > \frac{1}{x_2} - \frac{\zeta \cdot diff_2}{x_2} = \frac{\Delta x_2}{x_2}$, the fairness index also increases. When the packet loss is detected, both rates are decreased by β times. So the fairness index keeps the same value. In summary, in each phase of the control law, the fairness index is strictly none decreasing. Therefore, eventually, the two CTCP flows converge to their fair share. □

Then, we consider the case where different CTCP flows may have different delays. We show by *Theorem 2* that the RTT unfairness of CTCP is bounded by that of the standard TCP Reno.

Theorem 2: Let *Th1* and *Th2* present the throughput of two CTCP flows with round trip time as *R1* and *R2*, respectively. Then, the following inequality satisfied

$$\frac{Th_1}{Th_2} < \left(\frac{R_2}{R_1}\right)^2.$$

Proof. One CTCP flow contains two components, delay-based component (*dwnd*) and the loss-based component (*cwnd*). The delay-based component reacts to the increase of round trip time and tries to maintain certain number of packets backlogged in the bottleneck queue. Since the flow rate of each connection is approximately proportional to its queue size, the delay-based component of each connection give roughly same throughput. However, we know that the throughput ratio of TCP Reno is inversely proportional to the square of their RTT ratio.

Let Λ and Λ' stand for the throughput of the delay-based and loss-based component of a CTCP flow. We have,

$$\frac{\Lambda_1}{\Lambda_2} \approx 1 < \frac{\Lambda'_1}{\Lambda'_2} = \left(\frac{R_2}{R_1}\right)^2 \Rightarrow \frac{Th_1}{Th_2} = \frac{\Lambda_1 + \Lambda'_1}{\Lambda_2 + \Lambda'_2} \leq \left(\frac{R_2}{R_1}\right)^2. \qquad \square$$

Note that when including a delay-based component into the standard TCP would improve the RTT fairness of TCP's congestion avoidance algorithm.

C. TCP Fairness

In this section, we study the impact of CTCP on other standard TCP flows when they are competing for the same bottleneck. More specifically, we want to quantify the throughput reduction of the regular TCP flows because of the introduction of CTCP flows. We define a new metric here to measure the TCP fairness, named *bandwidth stolen.*

Definition 1: *bandwidth stolen*. Let *P* be the aggregated throughput of *m* regular TCP flows when they compete with another *l* regular TCP flows. Let *Q* be the aggregated throughput of *m* regular TCP flows when they compete with another *l* high-speed protocol flows in the same network topology. Then, $B_{stolen} = \frac{P - Q}{P}$ is the *bandwidth stolen* by high-speed protocol flows from regular TCP flows.

Theorem 3: With the system model shown in Figure 3, when $\frac{B}{m+l} > \gamma$, CTCP will not steal bandwidth from competing regular TCP flows.

Proof. We assume all packet losses are caused by buffer overflow and synchronized. We use a graph argument. Figure 6 shows a LFP when CTCP flows compete with regular TCP flows. As discussed before, CTCP flows will aggressively increase *dwnd* until there are γ packets are backlogged at the bottleneck queue at time *E*. Assume the window size at this point is w_0. After that, *dwnd* declines while *cwnd* continues to increase by one packet every RTT. Since $\frac{B}{m+l} > \gamma$, *cwnd* will eventually reach w_0 before the next packet loss event. However, at this point (time *F*), *dwnd* is approaching zero. From then, CTCP is just controlled by its loss-based component. And at time *G*, buffer overflows and all flows sense packet loss. Since all *dwnd* drop to zero when packet loss occurs, each regular TCP flow will get a maximal window size as if there were (*m*+*l*) regular TCP flows. The average window of a TCP flow equals to 3/8 of its maximal window size. Therefore, the *m* regular TCP flows will receive the same throughput no matter they compete with *l* other TCP flows or *l* CTCP flows. □

Theorem 3 shows that CTCP is fairness to TCP flows in term of not reducing TCP's throughput when the network is sufficiently buffered. However, CTCP does have higher throughput than regular TCP. It is because CTCP can make better use of free bandwidth that is currently not utilized.

Note that when the network is significantly under buffered (which we argue should not be a normal setup), CTCP may still steal bandwidth from regular TCP flows. It is because when buffer size is small, the early congestion detection may not perform well. We will discuss this worse case behavior in Section VI.B.5).

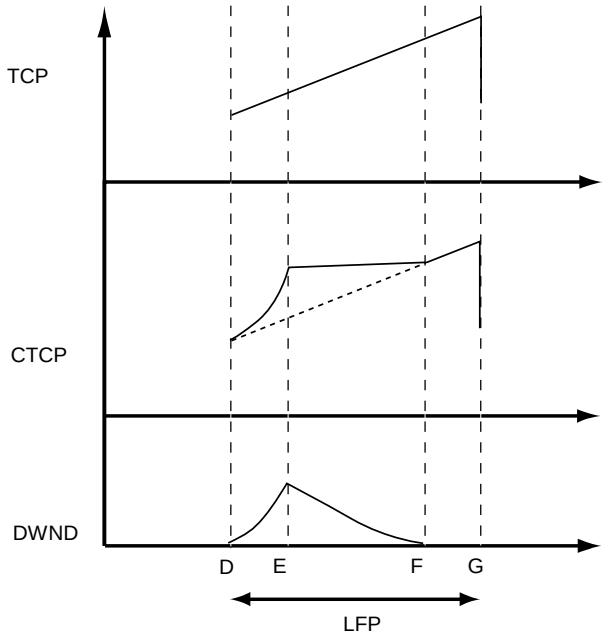

Figure 6. Window evolution of TCP and CTCP.

D. Summary of CTCP characteristics

Our analysis above shows that the scalable delay-based component in CTCP can aggressively obtain free bandwidth when the network is light-loaded, while gracefully reduce its sending rate when queue is built and avoid adding more self-induced packet losses. In this way, CTCP can efficiently utilize the bandwidth while at the same time maintaining good RTT fairness and TCP fairness.

V. IMPLEMENTATION

We have implemented CTCP on the Microsoft Windows Platform by modifying the TCP/IP stack.

The first challenge is to design a mechanism that can precisely track the changes in round trip time with minimal overhead, and can scale well to support many concurrent TCP connections. Naively taking RTT samples for every packet will obviously over-kill both CPU and system memory, especially for high-speed and long distance networks where a whole window worth of data may have tens of thousands packets. Therefore, we need to limit the number of samples taken, but without lose of much accuracy. In our implementation, we only take up to *M* sample per window of data. *M* scales with the round trip delay. More specifically, $M \propto RTT/\delta$, where δ is the minimal RTT value on the Internet. We believe $\delta = 1ms$ is a reasonable value, since most of operating systems have a scheduling accuracy larger than that. Since TCP flows can not change their sending rate faster than their RTT, letting $M \approx RTT/\delta$ can pretty well track the changes of queueing delay on the network path. In order to further improve the efficiency in memory usage, we develop a dynamic memory allocation mechanism to allocate sample buffers from a kernel fix-size per-processor pool to each connection in an on-demand manner. The smallest unit (block) is 256 bytes which can hold 32 samples. As the window increases, more packets are sent. If current sample buffer is not enough, more blocks are allocated and linked to the existing sample buffer until up to *M* samples are taken. Note that sampled packets are uniformly distributed among the whole window. If a sample block is empty due to a reduced window or lack of application data, the unused blocks are returned to the memory pool. This dynamic buffer management ensures the scalability of our implementation, so that it can work well even in a busy server which could host tens of thousands of TCP connections simultaneously. Note that it may also require high-resolution timer to time RTT sample. On Win32 platform, we can get a micro-second timer by using *KeQueryPerformance-Counter*. After WinXP SP2, *KeQueryPerformanceCounter* has been optimized to directly read CPU's Time Stamp Counter if available, and therefore introduces very less overhead.

The rest of implementation is rather straightforward. We add two new state variables into the standard TCP Control Block, namely *dwnd* and *baseRTT*. The *baseRTT* is a value that tracks the minimal RTT sample measured so far and it is used as an estimation of the transmission delay of a single packet. Following the common practice of high-speed protocols, CTCP also revert to standard TCP behavior when the window is small. Delay-based component only kicks in when *cwnd* is larger than some threshold, *lowwnd*. When the delay-based component kicks in, we let it at least increase one MSS per RTT. Therefore, from the increase law in equation (5), CTCP window should be at least 41 packets. So, we select *lowwnd* to be 41 MSS.

Dwnd is updated at the end of each round. If more than *N* (currently set to 5) RTT samples are taken, an average RTT is calculated and used to update *dwnd* according to equation (5). Note that RTT sampling and *dwnd* update are frozen during the loss recovery phase. It is because the retransmission during the loss recovery phase may result in inaccurate RTT samples and can adversely affect the delay-based control.

VI. PERFORMANCE EVALUATION

A. Methodology

We constructed a test-bed to conduct experiments for CTCP in our lab. The test-bed contains several DELL Desktop GX280 desktops equipped with Intel Pro/1000 XT Giga Ethernet cards. We use a DELL WS450 workstation as a router that connects two DLink DGS-1008T gigabit switches. The router is running FreeBSD 5.3 and DummyNet [25]. The DELL desktops are running Microsoft Windows and connected to the DLink switches. The testing environment is illustrated in Figure 7. We have extended the Windows TCP/IP stack to simultaneously support multiple TCP variants. Applications can dynamically select a congestion control scheme by using a Socket option. We modified Iperf [21] to support the new Socket option, so that it can test the different TCP variants.

We configure DummyNet to emulate network conditions with different packet loss rate and round trip delay. In each experiment, we set DummyNet to limit the bottleneck link speed to be 700Mbps. It is the highest speed we can get before the router's CPU becomes a bottleneck. We configure the router to use DropTail queue management. Unless otherwise pointed, the link delay is 100ms and the buffer size at the router is set to 1500

packets.

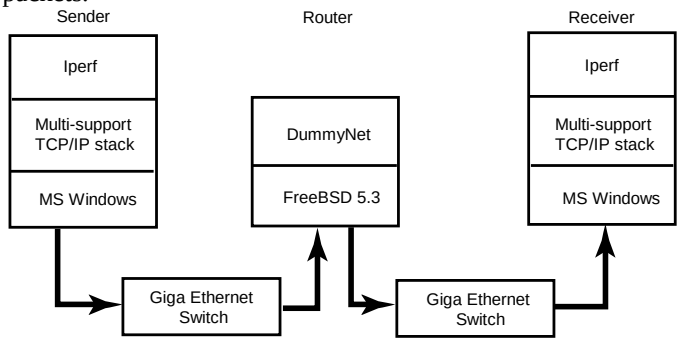

Figure 7. Testing Environment.

We test three TCP implementations on our test-bed: CTCP, HSTCP and the default Windows TCP implementation (regular TCP). We make our own implementation of HSTCP according to RFC 3645 and the reference implementation in NS2 [22]. In all three TCP implementations, New Reno, SACK, D-SACK and TCP high performance extensions are implemented and enabled by default. Each experiment lasts for 300 seconds, and the results presented are averaged over 5 runs of each test.

B. Results

1) Utilization

We first want to verify whether or not CTCP can effectively use the available bandwidth in high-speed and long delay environment. We configured DummyNet to generate random packet losses. We varied the loss rate from 10^{-2} to 10^{-6}. We ran 4 regular TCP, HSTCP and CTCP flows, respectively. The aggregated throughput of each TCP variant is plotted in Figure 8. Note that when packet loss is high ($>10^{-3}$), all three protocols behave exactly the same. However, with the decrease of packet loss rate, HSTCP and CTCP can use the bandwidth more efficiently. CTCP has slightly higher throughput compared to HSTCP. The reasons are two-fold: 1) the CTCP's response function is slightly more aggressive than HSTCP in moderate window range; and 2) CTCP introduces much less self-induced loss due to the delay-based nature.

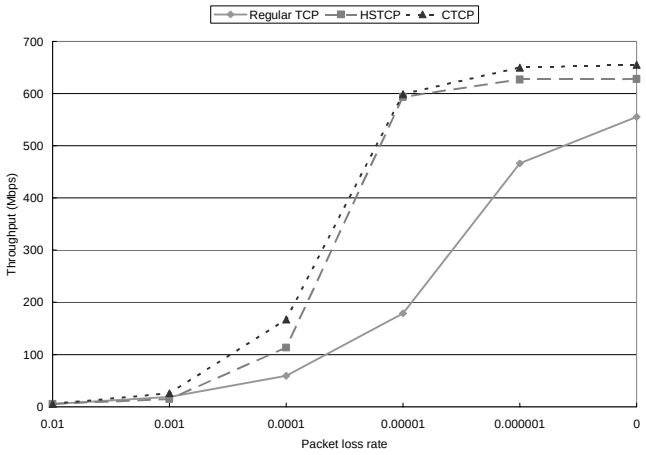

Figure 8. Throughputs under different packet loss rates.

We then conducted another set of experiments under burst background traffic. We generated On/Off UDP traffic with different peak data rate. The on-period and off-period of the UDP traffic were both 10s. Table 1 summarizes the aggregated throughputs and link utilizations (shown in brackets, normalized by the theoretical available bandwidth left over by UDP traffic) of 4 testing flows. It shows that CTCP and HSTCP can efficiently recover from the packet losses caused by burst background traffic and remain high link utilization. However, regular TCP can not efficiently use the link bandwidth. When the peak rate of UDP traffic goes from 50Mbps to 200Mbps, the utilization of the bottleneck link drops to 66%. This verifies that TCP's congestion control algorithm is too conservative under high-speed and long delay networks.

Table 1. Throughputs under burst UDP traffic.

BG traffic peak rate	50Mbps	100Mbps	150Mbps	200Mbps
Regular TCP	583.47 (86%)	558.44 (85%)	415.01 (66%)	404.19 (66%)
HSTCP	613.77 (91%)	595.87 (91%)	566.85 (90%)	543.35 (90%)
CTCP	625.01 (93%)	600 (93%)	581.5 (93%)	544.83 (91%)

2) TCP fairness

After showing that CTCP is effective in utilizing link bandwidth, we evaluate the TCP fairness property of CTCP. There are many methods that improve the efficiency at the cost of fairness to the regular TCP flows. However, our goal of CTCP is to improve the efficiency and maintain TCP fairness at the same time.

To qualify the TCP fairness, we first ran 8 regular TCP flows as baseline. Then, we replaced 4 flows to be the high speed protocols and repeated the experiments under the same condition. We compared the throughputs got by regular TCP flows with and without the present of high speed protocols. We used the *bandwidth stolen* defined in IV.C as an index in our comparisons.

The first experiment investigated the TCP fairness under different link packet loss rates. Figure 9 and Figure 10 present the results of HSTCP and CTCP, respectively. The *Regular TCP(baseline)* presents the throughput of 4 regular TCP flows in the baseline test (total 8 regular TCP flows are in test). The *Regular TCP* line shows the throughput got of 4 regular TCP flows when they were competing with 4 high-speed protocols. The gap between these two lines demonstrates the throughput reduction of regular TCP flows when there are high-speed protocols. In Figure 9, we can see when the packet loss is high (>0.0001), HSTCP will not degrade the throughput of regular TCP. This is because when the packet loss rate is high, the link is under utilized. Fairness issue only rises when the link is fully utilized. When packet loss is light, HSTCP begins to occupy the buffer more quickly than regular TCP and induces more packet loss events. As a consequence, the regular TCP obtains much less throughput compared to the baseline case. On the contrary, in Figure 10, when the bottleneck link is fully utilized, the delay-based component begins to retreat early without causing additional self-induced packet losses, and therefore, the competing regular TCP flows will receive similar throughput as in the baseline case. Figure 11 shows the *bandwidth stolen*. HSTCP

can steal up to 70% of bandwidth from regular TCP flows, while throughput reduction of regular TCP when competing with CTCP is less than 10%.

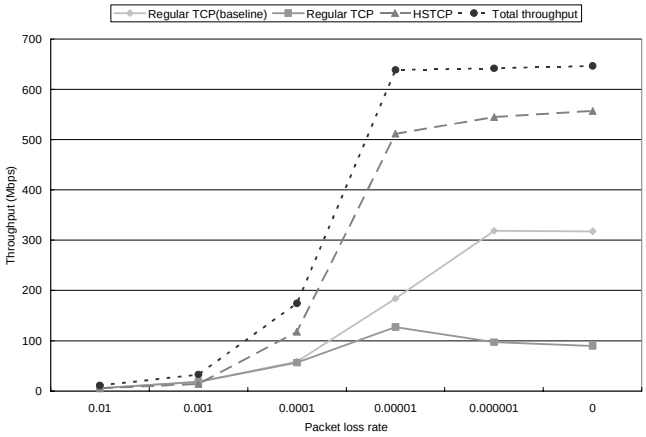

Figure 9. Throughput of HSTCP and Regular TCP flows when competing for same bottleneck.

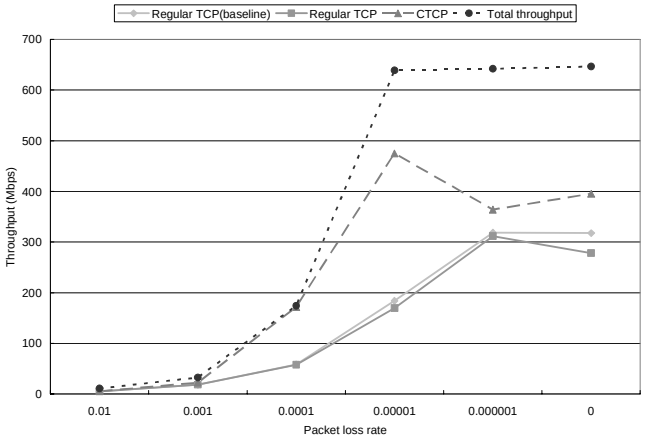

Figure 10. Throughput of CTCP and Regular TCP flows when competing for same bottleneck.

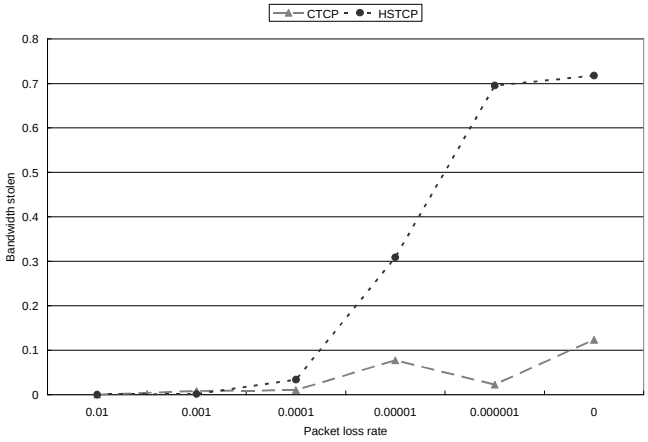

Figure 11. Bandwidth Stolen under various packet loss rate.

We then repeated the experiment under burst traffic setup. We used the same On/off UDP background traffic pattern as in last section. Figure 12 shows the *bandwidth stolen* of CTCP and HSTCP in this experiment. We get similar results that HSTCP can cause around 60% throughput reduction of regular TCP, while the throughput reduction cause by CTCP is around 10%. Therefore, CTCP is much fairer to regular TCP then HSTCP.

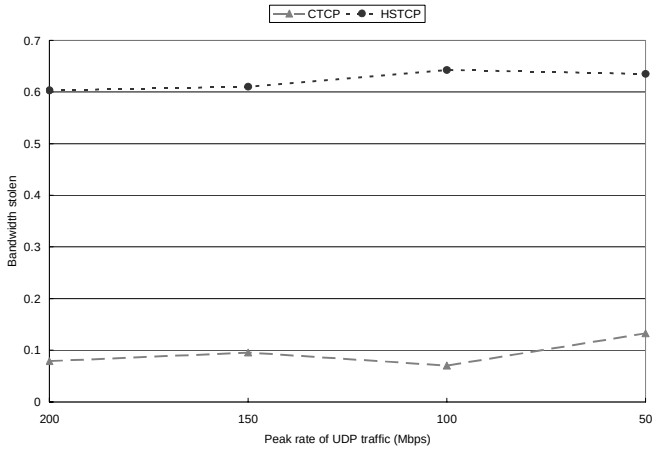

Figure 12. Bandwidth Stolen under burst background traffic.

3) RTT fairness

In this experiment, four high-speed flows were competing for the bottleneck link with different round trip delay. Two of them had shorter delay with 40ms. Two others had longer delay which varied among 40ms, 80ms, 120ms, and 240ms. The bottleneck link delay is 20ms, and we set the buffer size to be 1000 packets.

Table 2 summarizes the throughput ratio between flows with different round trip delay. It is not surprising to see that HSTCP has very severe RTT unfairness because in this experiment, most of packet losses are synchronized. The interesting thing is that CTCP has much improved RTT fairness compared to regular TCP. It is because the delay-based component included in CTCP gives favor to long-delay flows.

Table 2. Throughput ratio with different round trip delay.

Inverse RTT ratio	1	2	3	6
Regular TCP	0.9	3.6	6.2	31.6
HSTCP	1	28.9	90.5	233.8
CTCP	1	2.2	4.1	9.5

4) Impact of gamma

Note that in previous experiments, we evaluate CTCP with γ=30. Recall from in Section III.B, that γ is a tradeoff between throughput and the buffer requirement for TCP fairness. In this section, we explore the impact of γ on the performance of CTCP, and this gives us the insight of how to choose γ value.

In theory, we expect to choose small γ value. But we show that too small γ will adversely impact the throughput. It is because a slightly disturbance on the RTT will be sensed as early congestion in a dynamic network. To measure the impact of different γ value, we ran a CTCP flow with mixed traffic of UDP and TCP background traffic. We added a CBR UDP traffic which occupied 20% of capacity, and then we randomly added TCP flows. Each TCP flow would transmit 50Mbyte data. We measured the throughput of the CTCP flow over 5 minutes. Figure 13 shows the throughput of the CTCP flow with different γ value. The grayed column presents the median value of 10

runs. The two short bars present the maximal and minimal value. It is expected that with higher γ, CTCP will have higher throughput. It is because that CTCP connection will accumulate more packets in the bottleneck queue and reduce the chance of buffer underflow, which causes throughput reduction. However, when $\gamma > 30$, the throughput increase is almost saturated (from γ=30 to γ=50, the increase is less than 5%). Therefore, we set γ to be 30 in this paper.

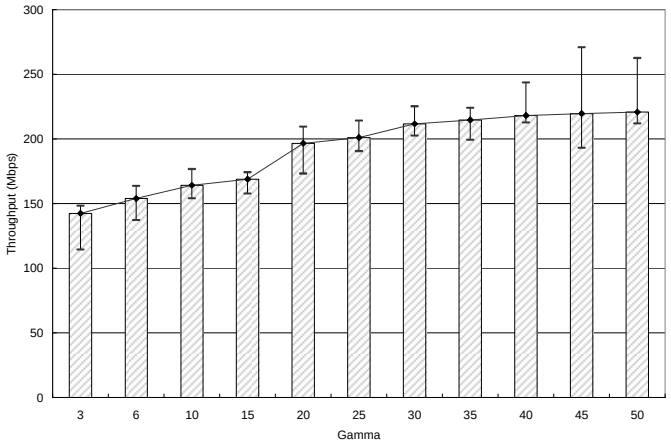

Figure 13. Throughput of CTCP with different gamma.

5) Impact on bottleneck buffer size

As we mentioned before, delay-based component requires certain amount of buffer space at the bottleneck link, i.e. γ packet per flow. If the buffer size is too small, the delay may not significantly increase during the congestion period, so that the early congestion detection will not be effective. If this happens, CTCP would degrade to a pure loss-based congestion avoidance algorithm. We conducted experiments under different buffer sizes. Our tests contained 4 high-speed flows with 4 regular TCP flows. Similar as before, we firstly ran 8 regular TCP flows and measured their throughput as a baseline. We plot the *bandwidth stolen* of HSTCP and CTCP in Figure 14.

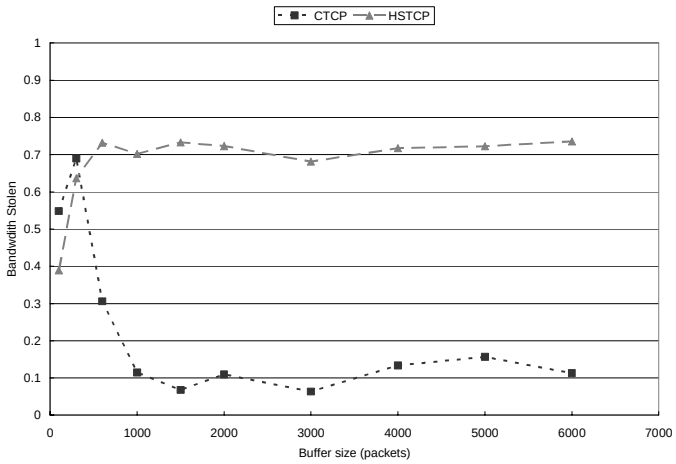

Figure 14. Bandwidth Stolen under different buffer size.

We can see from Figure 14, that when the buffer size is very small (<=300packets), the delay-based control law can not work well. As a consequence, CTCP stole bandwidth from regular TCP flows similar as HSTCP, as we intentionally set CTCP to have similar aggressiveness as HSTCP (actually, CTCP is a bit more aggressive than HSTCP with low window size). However, when the buffer size increases, the delay-based control law became more effective. Therefore, CTCP demonstrated good TCP fairness (the bandwidth stolen drops to around 10%). HSTCP, on the other hand, stole more bandwidth from competing regular TCP flows with the increase of the buffer size (the *bandwidth stolen* became saturated after some point). The reason is that when buffer size is large, the window of HSTCP is also larger and it is more aggressive compared to regular TCP.

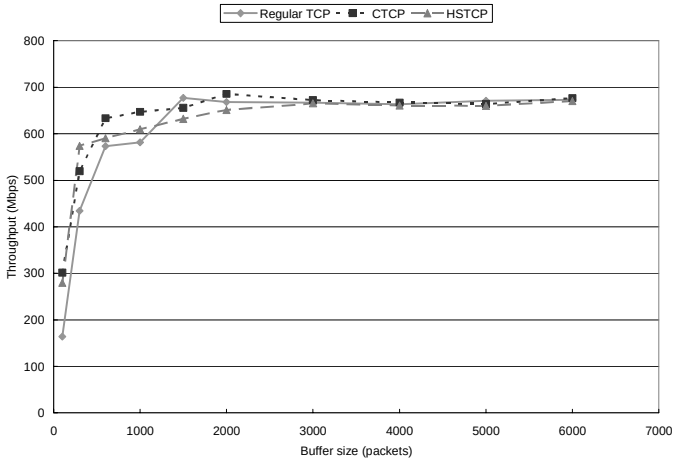

Figure 15. Throughput under different buffer size.

We argue that for high-speed and long delay network, it is essential to provide sufficient buffer space in order to fully utilize the link capacity. To show that, we plot in Figure 15 the aggregated throughput of the eight flows under different buffer sizes in above experiment. It shows that only with enough buffer size (e.g. 1000 packets), the 700Mbps link with 100ms delay can be fully utilized. Therefore, we expect reasonable large buffer should be deployed on the high-speed and long delay Internet that we target at.

6) Impact of window reduction rule according to delay increase

In this section, we evaluate the impact on the TCP fairness of the window reduction control law with the increasing of delay. We modified the CTCP implementation and removed the window reduction rule. We refer this modified CTCP implementation as CTCP-NWR. Note that CTCP-NWR is very similar to TCP Africa [27]. We conducted the following experiment. We set the bottleneck link speed to be 500Mbps with round trip delay 60ms. Accordingly, we set the buffer size to be 750 packets. We tested 4 regular TCP flows first as a baseline and we had averaged throughput of two flows as 229.1Mbps. Then, we ran 2 regular TCP flows against 2 HSTCP, CTCP-NRR, and CTCP flows, respectively. Table 3 summarizes the results. We can see that without the window reduction rule, CTCP-NRR still behaved more aggressive that regular TCP and caused 50% throughput reduction. Indeed, this is better than HSTCP, which resulted in 81% bandwidth stolen, but much worse than CTCP,

which only had 6% bandwidth stolen. This confirms that reducing window based on delay information is essential to ensure TCP fairness in a mixed network environment.

Table 3. Impact on TCP fairness of window reduction rule in CTCP (unit Mbps)

	Regular TCP	High-speed protocol	Sum	Bandwidth stolen
HSTCP	42.9	435.	478.	81%
CTCP-NWR	113.6	362.	476.	50%
CTCP	215.2	264.9	480.1	6%

VII. CONCLUSIONS

In this paper, we present a novel congestion control algorithm for high-speed and long delay networks. Our Compound TCP approach combines a scalable delay-based component with a standard TCP loss-based component. The delay-based component can efficiently use free bandwidth with its scalable increasing law. When the network is congested, the delay-based component will gracefully reduce the sending rate, but the loss-based component keeps the throughput of CTCP lower bounded by TCP Reno. This way, CTCP will not be timid, nor induce more self-induced packet losses than a single TCP Reno flow, and therefore achieves good TCP fairness. Further, delay-based schemes allocate network resource without RTT bias. Therefore, adding a delay-based component in CTCP greatly improves the RTT fairness even compared to TCP Reno.

We have implemented CTCP on Windows Platform by modifying Win32 TCP/IP stack. We conducted excessive lab experiments with our implementation and convinced ourselves that our implementation is stable and robust. The experimental results verify that CTCP can effectively utilize the link capacity, while at the same time maintaining excellent RTT and TCP fairness.

Finally, we note that CTCP may still be able to improve in many ways. For example, one going-on effort is to adaptively set γ value. Our goal is to detect early congestion with constant buffer requirement independent of the number of CTCP flows. We are currently investigating several ways to achieve this goal.

VIII. ACKNOWLEDGEMENT

The authors are very grateful to Sanjay Kaniyar, Deepak Bansal and Arvind Murching for their insightful comments and suggestions.

REFERENCES

[1] M. Allman, V. Paxson and W. Stevens, "TCP Congestion Control", *RFC 2581*, April 1999.

[2] V. Jacobson and M. J. Karels, "Congestion Avoidance and Control", *SigCOMM 88*, 1988.

[3] S. Floyd, "HighSpeed TCP for Large Congestion Windows", *RFC 3649*, December 2003.

[4] Tom Kelly, "Scalable TCP: Improving Performance in HighSpeed Wide Area Networks", *in First International Workshop on Protocols for Fast Long Distance Networks*, Geneva, February 2003.

[5] C. Jin, D. Wei and S. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance", *In Proc IEEE Infocom 2004*.

[6] L. Xu, K. Harfoush and I. Rhee, "Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks", *In Proc. IEEE InfoCOM 2004*.

[7] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in high performance computational grid environments", *Parallel Computing*, May 2002.

[8] S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet", *IEEE/ACM Trans. on Networking*, August 1999.

[9] J. Mo, R.J. La, V. Anantharam, and J.Walrand, "Analysis and Comparison of TCP Reno and Vegas", *in Proceedings of INFOCOM '99*, March 1999.

[10] J. Martin, A. Nilsson, I. Rhee, "Delay-based Congestion Avoidance for TCP", *IEEE/ACM Transactions on Networking*, June 2003.

[11] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance", *in Proceedings of the SIGCOMM '94 Symposium*, Aug. 1994.

[12] J. Padhya, V. Firoiu, D. Towsley and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", *in Proc ACM SigCOMM* 1998.

[13] E. de Souza and D. Agaral, "A Highspeed TCP Study: Characteristics and Deployment issues", *LBL Technique report*.

[14] D. Bansal and H. Balakrishnan, "Binomial Congestion Control Algorithms", i*n Proc Infocom 2001*.

[15] R. Wang, G. Pau, K. Yamada, M. Y. Sanadidi and M. Gerla, "TCP Start up Performance in Large Bandwidth Delay Networks", *in Proc Infocom 2004*.

[16] J.S. Ahn, P. B. Danzig, Z. Liu and L. Yan, "Evaluation of TCP Vegas: Emulation and experiment", *ACM SigCOMM*, 1995.

[17] S. Ravot, "TCP transfers over high latency/bandwidth networks & Grid DT", *presented at First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003)*, February 3-4, 2003, CERN, Geneva, Switzerland

[18] S. Floyd, "Limited Slow-Start for TCP with Large Congestion Windows", *Internet Draft, draft-floyd-tcp-slowstart-01.txt*, Aug, 2002.

[19] C. Samios and M. K. Vernon, "Modeling the Throughput of TCP Vages", *ACM SigMetrics 2003.*

[20] Network Emulator for Windows (NEW). Available at http://blogs.msdn.com/eec/archive/2005/01/20/357532.aspx

[21] Iperf. Available at http://dast.nlanr.net/Projects/Iperf/

[22] The Network Simulation - NS2. Available at http://www.isi.edu/nsnam/ns/

[23] D. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks", Computer Networks, 1989.

[24] G. Hasegawa, M. Murata and H. Miyahara, "Fairness and Stability of Congestion Control Mechanisms of TCP", *In Proc Infocom* 1999.

[25] L. Rizzo, DummyNet, available at http://info.iet.unipi.it/~luigi/ip_dummynet.

[26] S. Low, Question about FAST TCP. *End-to-end mailing list*, Nov. 2003.

[27] R. King, R. Baraniuk and R. riedi, "TCP-Africa: An Adaptive and Fair Rapid Increase Rule for Scalable TCP", *In Proc Infocom 2005*.