CTCP-TUBE: Improving TCP-Friendliness Over Low-Buffered Network Links

Kun Tan Jingmin Song
Microsoft Research Asia
Beijing, China
{kuntan,jingmins}@microsoft.com

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

Cheng-Yuan Ho[1]
DCS, National Chiao Tung University
Taiwan
cyho@csie.nctu.edu.tw

***Abstract*—Compound TCP (CTCP) is a synergy of delay and loss-based congestion control, which achieves good efficiency, RTT fairness and TCP-friendliness. However, CTCP requires detecting incipient congestion effectively by estimating the backlogged packets and comparing it to a pre-defined parameter γ. Choosing the appropriate value for γ could be a problem because this parameter depends on both network configuration and the number of concurrent flows, which are generally unknown to the end-systems. As a consequence, when operating over under-buffered links, CTCP may demonstrate poor fairness to regular TCP flows that may be comparable to HSTCP.**

In this paper, we present a novel technique that automatically tunes CTCP parameters so that it greatly improves the TCP-friendliness of CTCP over under-buffered links. This new technique, called *Tuning-By-Emulation (TUBE)*, dynamically estimates the average queue size for a regular TCP flow, and based on which sets the parameter γ. This way CTCP can effectively lower γ on under-buffered links to keep good TCP-friendliness, and alternatively increases γ if the link buffer is sufficient to ensure high throughput. Our extensive packet-level simulations and test-bed experiments on a Windows implementation confirm the effectiveness of CTCP-TUBE.

Index Terms— high-speed network, congestion control, compound TCP

I. INTRODUCTION

Transmission Control Protocol (TCP) provides reliable data transmission with embedded congestion control algorithm, which effectively removes congestion collapses in the Internet by adjusting the sending rate according to the available bandwidth of the network. However, it has been reported that TCP substantially underutilizes network bandwidth over high-speed and long distance networks [2]. In last decade, researchers have been actively seeking new approaches to improve TCP performance over fast and long distance networks. However, new high-speed congestion control protocols must satisfy the following three requirements before they can be successfully 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 residual bandwidth, but not steal bandwidth from other flows.

In our previous work, we proposed *Compound TCP (CTCP)*, a promising approach that satisfies all aforementioned requirements [4]. CTCP is a synergy of both delay-based and loss-based congestion avoidance approaches, in which a scalable delay-based component is added to the standard TCP. This delay-based component can efficiently use the link capacity, and also can react early to congestion by sensing the changes in RTT. This way, CTCP achieves high link utilization, good RTT fairness and TCP friendliness.

To effectively detect early congestions, CTCP requires estimating the backlogged packets at bottleneck queue and compares this estimate to a *pre-defined threshold* γ. However, setting this threshold γ is particular difficult to CTCP (and to many other similar delay-based approaches), because γ largely depends on the network configuration and the number of concurrent flows that compete for the same bottleneck link, which are, unfortunately, unknown to end-systems. As a consequence, the original proposed CTCP with a *fixed* γ may still demonstrate poor TCP-friendliness over under-buffered network links. In the worst case, TCP-unfairness of CTCP may even be comparable to that of HSTCP [6]. One naïve solution to that problem is to configure γ to a very low value, but a very small γ may falsely detect congestion and adversely affect the throughput.

In this paper, we propose a novel technique that greatly improves the TCP-friendliness of CTCP over such under-buffered network links without degrading the protocol efficiency to utilize the link capacity. Instead of using a pre-defined threshold, our approach, *TUBE (Tuning-By-Emulation)* dynamically adjusts *threshold* γ based on the network setting in which the flow is operating. The basic idea of our proposal is to estimate the backlogged packets of a regular TCP along the same path by emulating the behavior of a regular TCP flow in runtime. Based on this, γ is set so as to ensure good TCP-friendliness. CTCP-TUBE can automatically adapt to different network configurations (i.e. buffer provisioning) and also concurrent competing flows. Our extensive simulations on NS2 simulator reveal the effectiveness of CTCP-TUBE. Although TUBE is proposed to improve the TCP-friendliness of CTCP, we believe it can shed the light on parameter tuning for general delay-based approaches as well.

[1] This work is done when Cheng-Yuan Ho is an intern in Microsoft Research Asia.

II. BACKGROUND AND MOTIVATION

CTCP [4] is a synergy of both delay- and loss-based approaches. It contains two components that jointly control the sending rate of a TCP sender. A new state variable is introduced in current TCP Control Block, namely, *dwnd* (Delay Window), which controls this delay-based component in CTCP. And the conventional *cwnd* (congestion window) controls the loss-based component. Then, the TCP sending window (called *window* hereafter) is now calculated as follows:

$$win = \min(cwnd + dwnd, awnd) \ , \tag{1}$$

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

Cwnd is updated 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. Specifically, *cwnd* is updated as follows:

$$\text{cwnd}(t+1) = \begin{cases} \text{cwnd}(t) + \frac{1}{\text{win}(t)}, \text{on receiving an ACK} \\ \frac{\text{cwnd}(t)}{2}, \text{if loss detected} \end{cases} \tag{2}$$

Dwnd is updated based on the delay information. It uses an approach similar to TCP Vegas [7] to detect early congestion in the network path. More specifically, CTCP estimates the number of backlogged packets of the connection by following algorithm:

$$\begin{aligned} &Expected = win / baseRTT \\ &Actual = win / RTT \\ &Diff = (Expected - Actual) \cdot baseRTT \end{aligned} \ . \tag{3}$$

The *baseRTT* is an estimation of the transmission delay of a packet. 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, *Diff* 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, i.e. if $diff < \gamma$, the network path is determined as under-utilized; otherwise, the network path is considered as congested. CTCP updates its *dwnd* based on the following rules:

$$\text{dwnd}(t+1) = \begin{cases} \text{dwnd}(t) + \left(\alpha \cdot \text{win}(t)^k - 1\right)^+, \text{if diff} < \gamma \\ \left(\text{dwnd}(t) - \xi \cdot \text{diff}\right)^+, \text{if diff} \geq \gamma \\ \left(\text{win}(t) \cdot (1-\beta) - \frac{\text{cwnd}}{2}\right)^+, \text{if loss detected} \end{cases} . \tag{4}$$

Parameters of α, β and *k* are tuned to have comparable scalability to HSTCP when there is absence of congestion.

From the control laws stated in (4), it essential requires the connection to have at least γ packets backlogged in the bottleneck queue to detect early congestion. In [4], we use a fixed value $\gamma = 30$ packets, after a number of empirical experiments. Although this setting achieves pretty good tradeoff between TCP fairness and throughput in our testing environment, it fails to maintain good TCP-friendliness over links which are either poorly buffered, or have many competing flows [6]. To demonstrate this, we perform simulation using a dumb-bell topology as shown in Figure 1. The bottleneck buffer size is 110 packets which is less than 10% of BDP (or sustaining only 14ms transmission) of the network path.

We run one regular TCP flow against increasing number of CTCP and HSTCP flows and we draw the *bandwidth stolen* in Figure 2. The *bandwidth stolen* is a metric that quantifies the impact on throughput of new high-speed protocols on regular TCP flows [4]. It is defined as *the ratio between the throughput of regular TCP when they compete with high-speed flows and when they compete with same number of regular TCP flows*. For a high-speed protocol to be fair the value of bandwidth stolen should be low so as to not reduce the throughput for regular TCP flows.

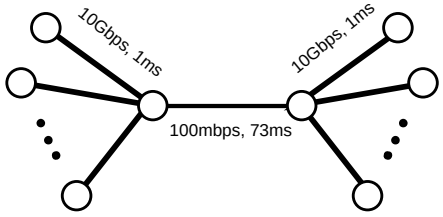

Figure 1. The dumb-bell topology for simulation.

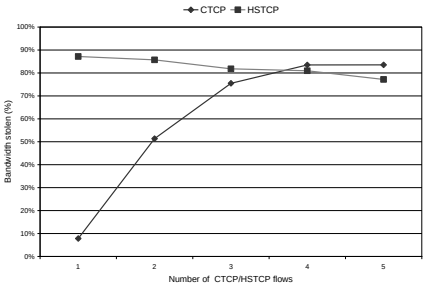

Figure 2. The bandwidth stolen when the number of high-speed flows increases.

Figure 2 clearly shows that when there is only one CTCP flow competing with one regular TCP flows, CTCP can retain pretty good TCP friendliness. However, with increase of the CTCP flows, CTCP becomes increasing unfair to regular TCP. When there are 5 CTCP flows compete with one regular TCP, the regular TCP flow loses over 80% of throughput compared to that if it is competing with 5 regular TCP flows. This is comparable to HSTCP.

The reason behind this phenomenon can be explained as follows. When there are only two flows in the network, the buffer is sufficient for each flow, i.e. each flow can get around 60 packets queued in the network buffer, and therefore, the delay-based component of CTCP can robustly detect congestion and retreat gracefully by decreasing *dwnd*. However, with the increase of the flow number, each flow gets fewer shares in the network buffer. As a consequence, the delay-based component in CTCP is less effective in detecting early congestion. When the flow number reaches four, the average buffer allocated for each flow is less than $\gamma = 30$, thus the delay-based component loses the ability to detect early congestion and it behaves as aggressively as HSTCP.

A naïve approach to fix this might choose a very small γ, e.g.

1 or 2 packet(s), which should be sufficiently small for most of practical network links. However, such small γ will make delay-based component too sensitive to delay jitter in the network path and generate a lot of false alarms, which in turn hurts the throughput.

In summary, a mechanism that can automatically adjust the parameter γ is critical for CTCP to work well in a general network setting: over under-buffered links, γ should be set to small to ensure TCP-friendliness; over sufficiently buffered links, γ should be adjusted to a high value to achieve better throughput.

III. CTCP-TUBE

Setting γ is very challenging in practice, because it is affected by the router buffer size and the number of concurrent competing flows. Our previous model on CTCP shows that γ should at least be less than $\frac{B}{m+l}$ to ensure the effectiveness of early congestion detection, where *m* and *l* present the flow number of concurrent regular TCP flows and CTCP flows that are competing for the same bottleneck link [4]. Generally, both *B* and *(m+l)* are unknown to end-systems. It is even very difficult to estimate them from end-systems in real-time, especially the number of flows, which can vary significantly over time. Fortunately there is a way to directly estimate the ratio $\frac{B}{m+l}$, even though the individual variables *B* or *(m+l) are hard to estimate*.

Let's first assume there are *(m+l)* regular TCP flows in the network. These *(m+l)* flows should be able to fairly share the bottleneck capacity in steady state. Therefore, they should also get roughly equal share of the buffers at the bottleneck, which should equal to $\frac{B}{m+l}$. For such a regular TCP flow, although it does not know either *B* or *(m+l)*, it can still infer $\frac{B}{m+l}$ easily by estimating its backlogged packets, which is a rather mature technique widely used in many delay-based protocols!

This brings us to the core idea of our approach, which we call *Tuning-by-Emulation*, or TUBE. We let the sender emulate the congestion window of a regular TCP. With this emulated regular TCP window, we can estimate the queue size of a regular TCP, Q_r, that competes with the high-speed flow in the same network path. Q_r can be regarded as a conservative estimate of $\frac{B}{m+l}$, assuming the high-speed flow is more aggressive than regular TCP. Therefore, if we choose CTCP $\gamma \le Q_r$, we can pretty well ensure its TCP-friendliness.

A. TUBE Algorithm

Although we flatter ourselves that the design of TUBE is subtle, the implementation is actually trivial. It is because in CTCP, there is already an *emulation* of regular TCP as the loss-based component. We can simply estimate the buffer occupancy of a competing regular TCP flow from state which CTCP already maintains. The details of TUBE algorithm is elaborated as follows.

We choose an initial γ. After every round, we calculate *diff* using Equation (3). At the same time, we estimate the backlogged packets of a regular TCP with

$$\begin{aligned} Expected\_reno &= cwin / baseRTT \\ Actual\_reno &= cwin / RTT \\ Diff\_reno &= (Expected\_reno - Actual\_reno) \cdot baseRTT \end{aligned} \quad . \quad (5)$$

However, since regular TCP reaches its maximum buffer occupancy just before a loss, we may only use the *diff_reno* calculated in the last round before a loss happens to update γ. We choose a $\gamma^* < diff\_reno$, and every time CTCP gets a loss, it updates γ with an exponentially moving average,

$$\gamma = (1-\lambda)\gamma + \lambda \cdot \gamma^*. \quad (6)$$

Figure 3 shows the pseudo-code of the TUBE algorithm. A new state variable, *diff_reno*, is added. Although *diff_reno* is updated every round, only the value before a packet loss is used to update γ. We further bound γ within a range $[\gamma_{low}, \gamma_{high}]$. Note that in line 17, *diff_reno* is set to invalid after updating. This is to prevent using stale *diff_reno* data when there are consecutive losses between which no *diff_reno* sample is taken.

```
1   Initialization:
2      Diff_reno = invalid;
3      Gamma = 30;
4
5   On-The-End-of-Round:
6      Expected_reno = cwnd / baseRTT;
7      Actual_reno   = cwnd / RTT;
8      Diff_reno=(Expected_reno-Actual_reno)
9                *baseRTT;
10
11  On-Packet-Loss:
12     If Diff_reno is valid then
13       g_star = 3/4*Diff_reno;
14       gamma=gamma*(1-lamda)+ lamda*g_star;
15       if (gamma < g_low) gamma=g_low;
16       elsif (gamma > g_high) gamma=g_high;
17       fi
18       Diff_reno = invalid;
19     fi
```

Figure 3. Pseudo-code for the TUBE algorithm.

We show the TUBE algorithm has following properties.

Property 1: CTCP-TUBE will not steal bandwidth from competing regular TCP flows.

Property 2: CTCP flows with TUBE will have same γ at the steady state, if they have same base RTT.

Due to the space limitation, we omit the proof of these two properties in this paper.

IV. PERFORMANCE EVALUATION

A. Methodology

We evaluate TUBE using NS2 simulations and lab experiments with our CTCP implementation on the Windows platform. Due to space limitation, we only present the NS2 simulations. All experiments are conducted under a dumbbell topology

as shown in Figure 1.

We mainly compare CTCP-TUBE with original CTCP with fixed gamma in very poorly buffered scenarios, in which our original CTCP fails to maintain good TCP-friendliness. Improving TCP-friendliness in such scenarios makes sense in practice, because during our tests on CTCP over production networks, we found a number of network links that are significantly under-provisioned [6]. In all TCP implementations, SACK is enabled by default.

Unless otherwise stated, the parameters of CTCP-TUBE in tests are $\gamma_{low} = 5$, $\gamma_{high} = 30$, $\lambda = 0.25$. For our original CTCP implementation, we keep $\gamma = 30$.

B. Results

1) TCP-friendliness

In the first experiment, we try to see how CTCP–TUBE behaves under the situation mentioned in Section II, in which one regular TCP flow is competing with varying number of CTCP flows. Figure 4 shows the bandwidth stolen of CTCP-TUBE. It is clearly evident that CTCP-TUBE greatly improves the TCP-friendliness. In our original CTCP design, when there are four flows (1 regular TCP and 3 CTCP), the average buffer allocated for each flow is less than 30 packets, so that the delay-based component of CTCP fails to detect congestion and it behaves as aggressively as HSTCP. However, TUBE effectively tracks the size of buffer that is occupied by the regular TCP and adjusts γ to a lower value when there are more flows. For example, when there are four flows in the network, TUBE effectively sets γ to be 21 packets. As a consequence, CTCP-TUBE maintains very good TCP-friendliness.

In the second experiment, we evaluate CTCP-TUBE with different buffer sizes. We set the bottleneck link speed to be 1Gbps and the round trip delay to be 100ms. We run 15 regular TCP flows and 15 CTCP flows simultaneously. We vary the buffer from 50 packets (0.6% of BDP) to 2000 packets (24% of BDP). Figure 5 shows the bandwidth stolen of original CTCP as well as CTCP-TUBE. We can see that CTCP steals much bandwidth from regular TCP flows until the buffer size is large enough (1000 packets), when the average buffer allocated to each flow is more than 30 packets. However, TUBE improves the TCP-friendliness of CTCP greatly in most cases due to its ability to adjust γ dynamically. Certainly, when the buffer is really tiny, i.e. 50 packets, even with $\gamma = \gamma_{low} = 5$, the delay-based component still cannot reliably detect congestion. However, we believe such tiny buffered links are rare in practice.

In the final experiment in this section, we test TCP-TUBE with varying number of CTCP flows under a high-speed network link. The bottleneck speed is 1Gbps and the round trip delay is 100ms. The buffer is set to 250 packets, which equals to 3% of BDP. We find that many links are actually configured with similar buffer sizes [6]. This could be because many routers choose this as the default value and administrators just keep it when deploying the network. We run 5 regular TCP flows and we vary the number of CTCP flows.

Figure 6 shows the bandwidth stolen. When there is only one CTCP flow, it can perfectly maintain the TCP friendliness. However, with increasing number of CTCP flows, the original CTCP cannot get the sufficient buffer for detecting congestion. As a consequence, it just greedily increases its sending rate thus adversely impacting regular TCP flows. However, TUBE can maintain TCP-friendliness well with more flows again due to its ability to adjust γ dynamically.

In summary, we show that TUBE is really able to adapt not only to changes of buffer size, but also to the number of concurrent flows. Although both of these values are unknown, TUBE is still able to choose a proper γ by emulating a regular TCP and estimating the buffer occupied by a regular TCP at runtime. This way, CTCP-TUBE remains TCP-friendly in a wide-range of scenarios including many under-buffered cases, in which original CTCP exhibited its worst case behavior.

1) Throughput

In this subsection, we evaluate the impacts of TUBE on CTCP throughput. Although TUBE may dynamically adjust γ to a small value when the link is less provisioned, we expect CTCP-TUBE is still able to utilize the link capacity efficiently when the buffer is sufficient.

In the first experiment, we set the link speed to be 1Gbps, and the round trip delay is 100ms. We set the buffer size is 1500 packets. We varied the loss rate of the link from 10^{-2} to 10^{-6}. We run 4 flows of the same type simultaneously and the aggregated throughput of the four flows are presented in Figure 7. It clearly shows that CTCP-TUBE has the similar ability to efficiently use link capacity compared to the original CTCP.

2) Intra-flow fairness and convergence of γ

Although each CTCP-TUBE flow individually adjusts its γ value, these values are converged if the flows have similar round trip delay. In this section, we verify this property. In this experiment, we set the bottleneck link speed to be 1Gbps and the round trip delay is 30ms. The bottleneck buffer is 200 packets. We start 3 CTCP-TUBE flows at time zero. Then after 300s, we start another 3 CTCP-TUBE flows. After that, we add 3 CTCP-TUBE flows every 900s, until there are 12 flows in the network.

Figure 8 shows the instantaneous throughput of each CTCP-TUBE flows. We see they converge to fair share quickly every time new flows are added. Figure 9 shows the instantaneous γ value of each CTCP-TUBE flow. At the beginning, there are only 3 flows in the network, so each of them can occupy enough buffer and their γ values are set to the maximum, i.e. 30 packets. However, when another 3 flows come, the average buffer size allocated for each flow is around 33 packets, so TUBE adjusts γ to a value equal to 3/4 of the average buffer allocation, i.e. 24 packets. And more flows are added, TUBE reduces γcorrespondingly. Each time, we can see that the γ value of each flow converges.

3) RTT fairness

As aforementioned, CTCP-TUBE with different round trip delays may adjust γ to different values so that RTT fairness may be slightly affected. To evaluate this, we conduct the following experiment. We use four flows competing for the bot-

tleneck links with different round trip delay. Two of them have shorter delay with 40ms. The other two flows have longer delay which varied between 40ms, 80ms, 120ms and 240ms. The bottleneck link speed is 1Gbps and the buffer size is set to 1500 packets.

Table 1 summarizes the throughput ratio between the two sets of flows with different round trip delay. We can see TUBE has little impact on the RTT fairness and it still keeps the similar RTT fairness property as original CTCP.

Table 1. Throughput ratio with different round trip delay.

Inverse RTT ratio	1	2	3	6
Regular TCP	1.01	3.38	8.51	21.7
HSTCP	1.04	9.65	85.76	198.7
CTCP	1.1	1.67	3.1	5.5
CTCP-TUBE	1.01	1.55	2.03	5.42

A. Lab test-bed experiments

In this section, we present the results of CTCP-TUBE in a lab test-bed. The implementation of TUBE is based on our original implementation of CTCP on Windows platform.

1) TCP friendliness

We repeat a similar experiment to the one described in Section IV.B.1) to verify the TCP-friendliness property of CTCP-TUBE. We configure DummyNet to set the link speed to be 300Mbps and buffer is set to 500 packets. The round trip delay is 100ms. We run one regular TCP against varying number of CTCP flows. We plot the results in Figure 10. Similarly, when there are 17 flows, the average buffer allocated for each flow is roughly 500/17 < 30. CTCP with a fixed value of γ will steal bandwidth from regular TCP. However, CTCP-TUBE maintains the good TCP friendliness in all cases.

2) Throughput

We also conducted experiments in the test-bed to verify the efficiency of CTCP-TUBE in utilizing high-speed link capacity. We set the bottleneck link speed is 700Mbps and buffer size is 1500 packets. The round trip delay is 100ms. We generate on/off UDP traffic with different peak data rate. The on-period and off-period are both 10s. In each experiment, we start 4 flows of same type simultaneously. We plot the throughput of each type of flow in Figure 11. We also plot the throughput of 4 CTCP flows with γ set to a fixed value 5. We can see TUBE has only modestly impact on CTCP's throughput (reducing 3%), although it significantly improves the TCP-fairness over under-provisioned links. However, as we can see, simply setting a low γ value may hurt the TCP throughput. In this case, it has nearly 10% throughput degradation compared to the original CTCP.

V. RELATED WORK

The research community is well aware of the fact that the conservative TCP congestion control algorithm becomes inefficient in high-speed and long delay networks. In the last decade, researchers have proposed numerous enhancements for TCP protocol. Some of them directly modify the increase/decrease parameters of TCP so as to improve the throughput in high-speed environments. Examples of this sort of enhancements include STCP [1], HSTCP [2], BIC-TCP [3]. However, these proposals suffer fundamental tradeoffs between throughput, RTT fairness and TCP-friendliness. Some other proposals use delay as congestion information, e.g. FAST [5], and the delay-based approach demonstrates nice properties in a network in which all flows are delay-based. But delay-based approaches generally are not competitive to loss-based flows as they try to remain only a small queue on the bottleneck link. Moreover, choosing the target queue size is an open question for delay-based approaches. CTCP [4] is designed to combine the advantages of both loss- and delay-based approaches. It incorporates a scalable delay-based component into the standard TCP congestion avoidance algorithm. The scalable delay-based component has a rapid window increase rule when it senses the network to be under-utilized and gracefully reduces the sending rate once the bottleneck queue is built. Using the delay-based component as an auto-tuning knob, CTCP achieves good efficiency, pretty RTT fairness and TCP-friendliness. However, as CTCP uses delay-based approach, it also inherits the fundamental question of how to set parameter for early congestion detection. In [4], the authors propose to use a fixed number. And therefore, in some excessively under-buffered link, CTCP may fail to maintain its TCP-friendliness and behaves in the worst case similar to HSTCP.

Tuning parameters for delay-based approaches in the literature is largely based on TCP Vegas [7], and tries to enhance TCP Vegas to be competitive to TCP Reno. In [10] and [11], the authors have developed mathematical models as well as used simulations to identify that the target queue-size for delay-based approaches is a function of both bottleneck buffer as well as the flow number of each type. Unfortunately, none of the above parameters is easily known to the end-systems. Hasegawa, et. al. [8] proposed a heuristic to dynamically switch TCP Vegas between a *moderate* and an *aggressive* mode. However, the switching decision is based on another parameter $Count_{max}$. This leaves another open question on how to set $Count_{max}$. Srijith et. al. [9] proposed TCP Vegas-A that can dynamically tune α and β. They used the throughput changes in adjacent rounds as heuristics. The basic idea is if the throughput increases in the new round, α and β should both increase. Otherwise, α and β should be both decreased. Although the proposed algorithm reduced "unfairness" of TCP Vegas to Reno, it is yet still far from true fairness. Similar to [10] and [11], CTCP-TUBE derives a relation between the parameter γ and the network parameters of buffer size and the number of concurrent flows. However, TUBE does not estimate either network buffer size, or the flow number. Instead, TUBE exploits an emulator at the sender side that emulates the behavior of a regular TCP. This way, a sender can guess a fair share of queue size of a regular TCP, which actually is the target queue-size for a delay-based flow to be TCP-friendly. We believe TUBE is not only applicable to CTCP, but it sheds light on tuning other delay-based approaches to be TCP-friendly.

VI. CONCLUSIONS

In this paper, we present a technique, called *Tuning-by-Emulation (TUBE)*, which effectively adjusts the targeting queue size for delay-based approaches. TUBE greatly improves the TCP-friendliness of *Compound TCP* over severely under-buffered network links. Moreover, TUBE sheds light on tuning parameters of other delay-based approaches to improve TCP-friendliness. Our extensive simulations confirm the effectiveness of TUBE.

REFERENCES

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

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

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

[4] K. Tan, J. Song, Q. Zhang and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. IEEE Infocom 2006.

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

[6] Y. T. Li. Evaluation of TCP Congestion Control Algorithms on the Windows Vista Platform. SLAC-TN-06-005, 2006.

[7] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance", *in Proc. ACM SIGCOMM*, 1994.

[8] G. Hasegawa, K. Kurata, and M. Murata, "Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet," ICNP 2000, pp. 177~186, Nov. 2000.

[9] K. N. Srijith, L. Jacob, and A. L. Ananda, "TCP Vegas-A: solving the fairness and rerouting issues of TCP Vegas," IPCCC 2003, pp. 309~316, Apr. 2003.

[10] E. Weigle and W. Feng, "A Case for TCP Vegas in High-Performance Computational Grids", HPDC 2001.

[11] W. Feng and S. Vanichpun, "Enabling compatibility between TCP Reno and TCP Vegas," SAINT 2003, pp. 301~308, Jan. 2003.

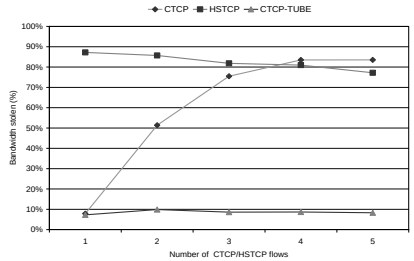

Figure 4. Bandwidth stolen of CTCP-TUBE. One regular TCP flow competes with varying number of high-speed flows.

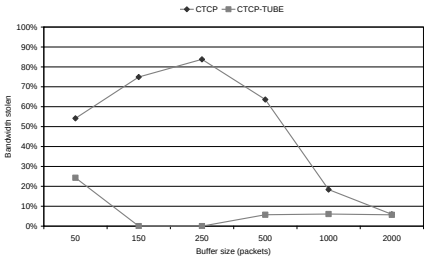

Figure 5. Bandwidth stolen under different buffer setting.

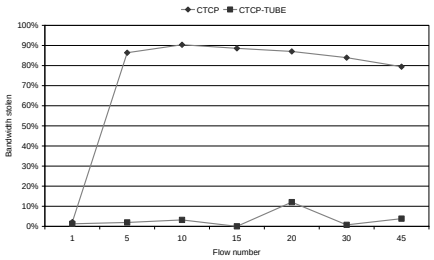

Figure 6. Bandwidth stolen under various CTCP flows.

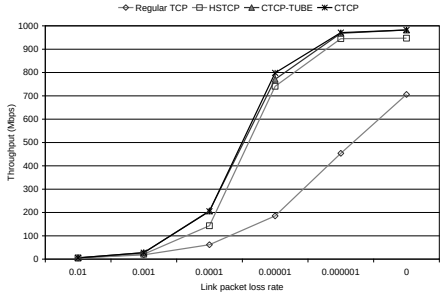

Figure 7. Throughput under various random link loss rate.

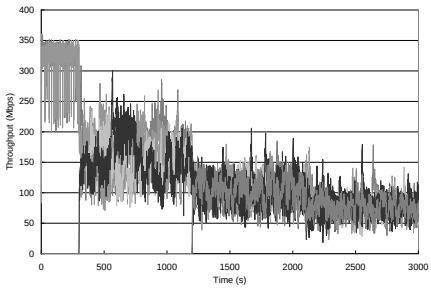

Figure 8. The instantaneous throughput of each CTCP-TUBE flow.

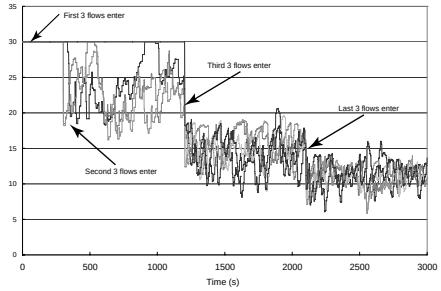

Figure 9. The instantaneous γ value of each CTCP-TUBE flow.

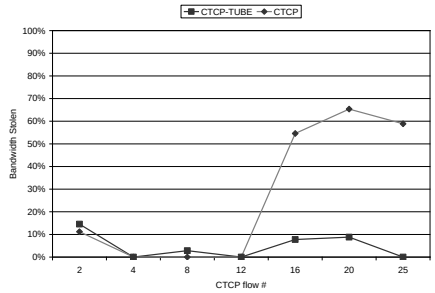

Figure 10. Bandwidth stolen. One regular TCP flow competes with different number of CTCP flows.

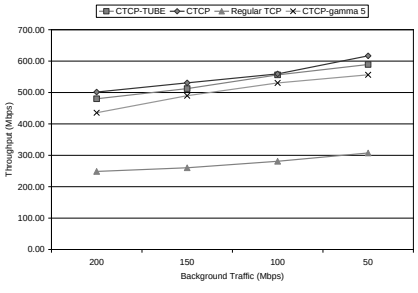

Figure 11. Throughput under on/off background traffic.