Aggregated Aggressiveness Control on Groups of TCP Flows

Soohyun Cho and Riccardo Bettati

Computer Science Department,
Texas A&M University,
College Station, TX 77843 USA
{s0c6496, bettati}@cs.tamu.edu

Abstract. The use of multiple concurrent parallel TCP flows is an easy way to achieve higher speed reliable data transfers. However, parallel TCP flows are inherently unfair with respect to single TCP flows. We suggest a new scheme called TCP-P, which controls aggressiveness of a group of parallel TCP flows by regulating their total aggressiveness (or unfairness) to be comparable to a single TCP flow, or any multiple thereof. TCP-P makes a group of N parallel TCP flows appear to other flows like k separable TCP flows - i.e., have strength k - through appropriate manipulations of increase and decrease behavior of the congestion windows of the TCP flows in the group. We implemented our scheme as part of Linux and experimental results show that the proposed scheme effectively controls aggressiveness of parallel TCP flows.

1 Introduction

A widely used scheme to work around the limitations of TCP over high delay-bandwidth product connections is to use multiple parallel TCP connections. The use of parallel TCP flows has several benefits compared to a single TCP flow [1]. If an end-host opens N parallel TCP flows to the same destination, its congestion window recovery and increase are N times faster than a single TCP flow [2]. As a result, the achievable throughput of parallel TCP flows is significantly bigger than that of a single TCP flow given the same packet loss probability. Unfortunately, this increase comes at the expense of the throughput experienced by other, single TCP flows, as sender nodes who open multiple parallel TCP flows will consume unfairly more bandwidth when they compete for the same bottleneck links.

With the increased venues for bundling of TCP flows (e.g., overlay networks with TCP splicing [3], large servers with topological aggregation of service delivery, dedicated connections between supercomputers or campuses, etc.,) flexible schemes are needed for the *controllable* aggregation of large numbers of parallel TCP flows. Naively limiting the number of parallel connections that applications in a node can open concurrently is not appropriate in many situations, as it violates the separation of application design from network resource allocation: Making the number of available connections visible to the application unduly burdens the application design. On the other hand, hiding the varying numbers of connection endpoints from the application through tunneling or multiplexing schemes typically is costly. Also, statically limiting the maximum aggregate sending rate of parallel TCP flows from sender nodes may leave network resources under-utilized because it disables TCP's available bandwidth probing ability beyond the given sending rate. It is therefore preferable to allow for TCP flows to aggregate, but do so in a controlled way.

Methods to control aggregation of TCP flows must have the following capabilities:

- Transparency to applications (management of connections and expected dynamics of data transmission should maintain TCP characteristics,)
- Compatibility with existing TCP implementations ("TCP-friendliness",)
- Controllability and flexibility of the service (the "control knob" offered by the mechanism should be intuitive and have measurable effect on behavior,)
- Effective use of available bandwidth (the mechanism should not prevent TCP from quickly making use of available bandwidth,)
- Flexible deployability (the mechanism should be deployable in single-sender (server), or multi-sender (overlay) scenarios.)

In this paper we propose aggregate *strength* as means to control the fairness of parallel TCP flows: The aggressiveness or unfairness of parallel TCP flows is appropriately controlled to not exceed that of a configurable number of single TCP flows, regardless of the number of TCP connections. With the term fairness (unfairness) we mean how fairly (unfairly) a group of parallel TCP flows from a node share network resources such as bandwidth with TCP flows from other nodes. By setting the aggregate strength of a group to some value k, the group of parallel TCP flows in a node behaves as if there were a group of k parallel TCP flows regardless of the number of parallel flows applications in the node open. By doing so we provide a flexible aggregate control of parallel TCP flows while keeping the ability of parallel TCP flows to effectively utilize available bandwidth.

We implement strength control within TCP-P, which is an extension to TCP. TCP-P controls the aggressiveness of a group of N parallel TCP flows from a node against single TCP flows from other nodes by controlling the strength of the group of flows. The "strength" in this context is a scalar value k of the TCP group and describes how big (in terms of number of flows) the group is perceived by other TCP flows from other nodes sharing network resources with the group. We will show in the following how this parameter provides a simple and intuitive means to control aggressiveness of parallel TCP flows.

There have been several efforts to improve TCP performance using parallel flows while constraining the unfairness of parallel TCP flows comparable to that of a *single* TCP flow. The Congestion Management (CM) architecture [4], Fractional/Combined TCP flows [5] and COCOON [6] are some examples. In contrast to these schemes, MulTCP [7] was proposed to claim k times more bandwidth for a single TCP connection. A MulTCP flow with parameter k increases and decreases its congestion window size as if there were k multiple TCP flows. However, as far as we know, there has been no scheme to controllably constrain the aggressiveness of parallel TCP flows.

The remainder of this paper is organized as follows: Section 2 presents the methodology we used to control parallel TCP flows' total strength. Section 3 describes our implementation in the Linux kernel. Section 4 presents experimental results that demonstrate how this implementation effectively controls the aggressiveness of parallel TCP flows. Section 5 concludes this paper.

2 Aggregate Control

Aggregate control of parallel TCP flows is achieved through modifications to TCP's congestion window increase and decrease behavior. During the increase phase, a normal TCP has two modes: exponential increase during slow-start, linear increase during congestion avoidance. Within the decrease phase, normal TCP responds to congestion events, such as three duplicate acknowledgement packets (ACKs,) by halving its congestion window size. With a given strength parameter k, we want to match the total amount of increase and decrease of congestion windows of a group of N parallel TCP-P flows to those of k single TCP flows.

We denote the amount of *increase* of congestion window of a single TCP flow i in increase phase as Δ_i^+ and the amount of *decrease* in decrease phase as Δ_i^-, respectively. Let the amount of increase and decrease of a TCP-P flow j in a group of size N be Δ_j^{p+} and Δ_j^{p-} respectively. To make N parallel TCP-P flows become like k single TCP flows, we need to make sure that $\sum_{i=1}^{k} \Delta_i^+ = \sum_{j=1}^{N} \Delta_j^{p+}$ for the given number of non-duplicate ACKs, and $\sum_{i=1}^{k} \Delta_i^- = \sum_{j=1}^{N} \Delta_j^{p-}$ for a congestion event.

2.1 Controlling Increase

In slow-start mode, TCP increases its congestion window by one per non-duplicate ACK until it detects congestion events or the congestion window size reaches its slow-start threshold value. When a TCP is in slow-start mode, the congestion window size of a TCP, W, after a non-duplicate ACK arriving at time t is shown in the following equation:

$$W(t+) = W(t) + 1. \tag{1}$$

When all TCP flows in a group of N unmodified parallel TCP flows are in slow-start mode, the total congestion window increase will be N times faster that a single TCP flow. For the same group size of TCP-P flows to have strength k, the congestion window of each TCP-P flow, W_j, should increase by $\frac{k}{N}$ per non-duplicate ACK as shown in the following equation:

$$W_j(t+) = W_j(t) + \frac{k}{N}. \tag{2}$$

In congestion avoidance mode, we want to make the aggregate congestion window size increase of N parallel TCP-P flows with strength k be equal to that of k TCP flows for the same amount of non-duplicate ACKs. We describe the special case of $k = 1$ first, and generalize it later. Let $W(t)$ be the congestion window size of a single TCP at time t, and we assume the sum of congestion window size of each TCP-P flow in the group is equal to $W(t)$, i.e., $\sum_{j=1}^{N} W_j(t) = W(t)$. The amount of congestion window increase of a single TCP per non-duplicate ACK in this mode is $\frac{1}{W(t)}$ as shown in the following equation:

$$W(t+) = W(t) + \frac{1}{W(t)}. \tag{3}$$

If each TCP-P flow in a group size N increases its congestion window by one, the total increase will be N. For a single TCP flow to increase its congestion window size W by N, it needs $W + (W+1) + (W+2) + \cdots + (W+N-1) = \sum_{i=0}^{N-1}(W+i)$ non-duplicate ACKs. Hence, to match the congestion window increase speed of N parallel TCP-P flows to that of a single TCP flow, we should require the group of TCP-P flows with size N to receive $\sum_{i=0}^{N-1}(W+i)$ non-duplicate ACKs before each TCP-P flow in the group increases its congestion window by one.

To ensure fairness among TCP-P flows within the group, we evenly distribute the total amount of non-duplicate ACKs required for a group to each TCP-P flow in the group, so that each TCP-P flow in a group needs to receive $\frac{\sum_{i=0}^{N-1}(W+i)}{N}$ non-duplicate ACKs before it can increase its window size by one. In this way, with the same amount of non-duplicate ACKs, i.e., $\sum_{i=0}^{N-1}(W+i)$, the N parallel TCP-P flows will increase their total congestion window size by the same amount, N, just as the single TCP flow.

For the case of $k > 1$, we generalize the previous case: parallel TCP-P flows need to increase their total window size of the group by k after the group received $\sum_{i=0}^{N-1}(k\overline{W} + i)$ non-duplicate ACKs. Here, $\overline{W}$ is the average of the congestion window sizes of k TCP flows, and we assume that $\sum_{i=1}^{k} W_i = k\overline{W} = \sum_{j=1}^{N} W_j$, i.e., the sum of congestion window W_i of k single TCP flows is equal to the sum of congestion window W_j of N parallel TCP-P flows at time t. We use the fact that the increase of the total congestion window size of k parallel TCP flows with a given number of non-duplicate ACKs is k times larger than the increase of the congestion window of a single TCP flow with the same window size (i.e., $k\overline{W}$). For this, each TCP-P flow in a parallel TCP-P group of size N should increase its congestion window by one after receiving the following amount of non-duplicate ACKs:

$$\frac{\sum_{i=0}^{N-1}(\sum_{j=1}^{N} W_j + i)}{k * N}. \tag{4}$$

As a result, the increase behavior of a group of N TCP flows can be made to closely reflect that of k TCP flows.

2.2 Controlling Decrease

A single TCP flow reduces its congestion window size W by half when it detects a congestion event, such as three duplicate ACKs at time t:

$$W(t+) = \frac{W(t)}{2}. \tag{5}$$

In unmodified parallel TCP flows, only single TCP flow in the group halves the congestion window size for each congestion event to the group. This behavior primarily contributes to the observed throughput advantage (and the unfairness) of parallel TCP flows over a single flow. In contrast, we let each TCP-P flow in a group responds to its own congestion event by reducing its own congestion window. In addition, TCP-P adjusts congestion windows of *other* TCP-P flows in the group as well based on the group size N and strength parameter k.

For $k = 1$, we halve *all* parallel TCP-P flows' congestion window sizes whenever *any* member flow detects a congestion event. For $k > 1$, we let the total congestion window size after a congestion event be $\frac{2k-1}{2k}$ of the previous total congestion window size of N parallel TCP-P flows. Hence, for a group of N TCP-P flows with strength k, the total amount of congestion window decreases according to the following equation:

$$\sum_{j=1}^{N} W_j(t+) = \sum_{j=1}^{N} W_j(t) * (\frac{2k-1}{2k}). \tag{6}$$

For $k = N$, N parallel TCP-P flows becomes unmodified N parallel TCP, and the total decrease amount of N TCP-P flows become $\frac{2N-1}{2N}$ of the previous total window size. This is the same as that of unmodified parallel TCP flows' [2].

2.3 Avoiding Unnecessary Decreases

Since packet drops in the network are typically bursty [8], multiple TCP flows in a parallel TCP group may simultaneously experience packet losses. If bursty packet drops occur, they may result in too much congestion window reduction to parallel TCP-P flows: Every TCP-P flow that experiences a congestion event might in turn trigger a congestion window reduction in other flows, which already may have responded to the congestion event. This results in a congestion response cascade. In traditional TCP a flow responds to congestion events only once within a congestion window, regardless of the number of lost packets. In comparison, TCP-P flows may end up with a lower throughput than that of a single TCP flow.

To avoid this unnecessary reduction of congestion windows, a TCP-P flow skips adjusting congestion windows of all member TCP-P flows if the elapsed time since its last adjustment by other flows is less than the minimum of the moving average of its round-trip times[1] (i.e., minimum *srtt*.) In doing so, we

[1] This is also called *smoothed* round-trip time, $srtt(t+) = (1-w) * srtt(t) + w * rtt(t)$ where $rtt(t)$ is round-trip time at time t and $w = \frac{1}{8}$.

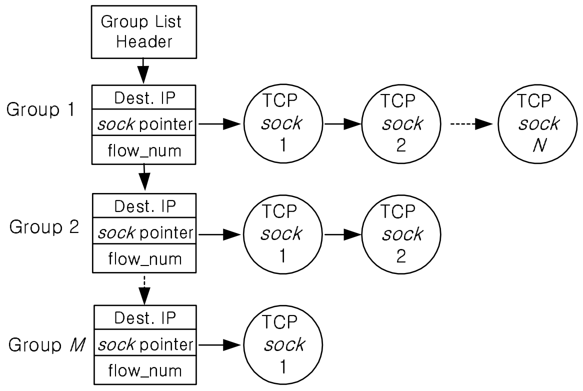


Fig. 1. Manage groups and flows

assume that the length of packet drop bursts does not typically last longer than the minimum of the moving average of round-trip times.

3 Implementation Issues

We implemented TCP-P scheme on Redhat Linux 9.0 kernel 2.4.20-8. The default behavior of Linux TCP implementation is based on TCP-Sack [9], time-stamping on each packet, and Quick-ACK [10]. Also, Linux uses the packet as the unit of congestion window size, unlike BSD, which uses bytes.

3.1 Structure

Whenever a TCP connection is established, the system kernel looks up the *group list* using the destination IP address as a key to know whether other flows already exist to the same destination[2]. If no such group exists, a new group entry is added in the list and the connection is registered as a member of the group using its `sock` structure pointer. Otherwise, the connection is added as a member to the group. When a connection closes, the connection is removed from the list of members of the group. If the group has no more members it is also deleted.

Fig. 1 illustrates how the Linux data structures are extended to manage TCP flow groups. The Linux TCP implementation has a structure named `sock` to manage socket information for each connection and `tcp_opt` for TCP specific information. We added new variables to those structures for TCP-P: a pointer that points `flow_num` variable of its group is added to `tcp_opt` to get the number of flows of its group without searching the group list, and a pointer to the next `sock` structure in the same group is added in the `sock` structure. Each *group* structure has a pointer to its first member's structure `sock` and

[2] In this implementation, we use destination address as group classifier. Other classifications could be used just as well.

has a integer variable flow_num to count the number of member flows of the group. Whenever a new member TCP is added or deleted in a group, this count variable updates the number of member TCP flows. For a system-wide control of *strength* we added a new system control parameter sysctl_tcp_strength in net/ipv4/sysctl_net_ipv4.c. This parameter can be easily changed in the run-time using the sysctl system call.

3.2 Implementing Increase

In slow-start mode, the congestion window of each TCP-P flow in a group is increased by $\frac{k}{N}$ per non-duplicate ACK, as described in Equation (2). The amount of increase, $\frac{k}{N}$, is less than or equal to 1 when k is not bigger than N. Since floating point arithmetic is not supported in the Linux kernel, we let each TCP-P flow in our scheme increase its congestion window size by k after receiving N non-duplicate ACKs.

In congestion avoidance mode, each TCP-P flow in a group of size N with strength parameter k should increase its congestion window by 1 after receiving $\frac{\sum_{i=0}^{N-1}(\sum_{j=1}^{N} W_j+i)}{k*N}$ non-duplicate ACKs as Equation (4). To implement this for each TCP-P flow independently, we use the following equation:

$$\begin{aligned}&\frac{\sum_{i=0}^{N-1}\sum_{j=1}^{N} W_j + \sum_{i=0}^{N-1} i}{k*N}\\ &= \frac{N\sum_{j=1}^{N} W_j + \sum_{i=0}^{N-1} i}{k*N}.\end{aligned} \tag{7}$$

Therefore, each TCP-P flow should increase its congestion window size W_j by 1 after $\frac{N\sum_{j=1}^{N} W_j+\sum_{i=0}^{N-1} i}{k*N}$ non-duplicate ACKs. Alternatively, it can increase the congestion window by $\frac{k*N}{N\sum_{j=1}^{N} W_j+\sum_{i=0}^{N-1} i}$ per non-duplicate ACK.

Looking up other TCP-P flows' congestion window size at every non-duplicate ACK arrival may result in serious overhead. To reduce operation cost we assume that all TCP-P flows in a group have the same window size W_0, so that $\sum_{j=1}^{N} W_j = NW_0$. With this assumption, each flow does not need to know other TCP-P flows' congestion window sizes. Instead, it can use its own congestion window W_j to estimate the total window size for the group. Each TCP-P can find the size of its group, N, easily because each TCP-P structure has a pointer to its group's member count variable flow_num as shown in Fig. 1.

Since floating point arithmetic is not supported in Linux kernel, we increase the congestion window size of each flow by k after it received $\frac{N^2*W_j+\sum_{i=0}^{N-1} i}{N}$ non-duplicate ACKs. This number can be further simplified as follows:

$$\begin{aligned}&\frac{N^2 * W_j + \sum_{i=0}^{N-1} i}{N}\\ &= NW_j + \frac{(N-1)N}{2N}\\ &= NW_j + \frac{N-1}{2}.\end{aligned} \tag{8}$$

Therefore, each TCP-P flow in a group of size N and strength k should increase its congestion window by k after receiving $NW_j + \frac{N-1}{2}$ non-duplicate ACKs.

3.3 Implementing Decrease

TCP-P controls the decrease amount of total congestion window sizes of parallel TCP-P flows according to Equation (6) to match with that of k unmodified parallel TCP flows. One possible method to implement this is to decrease every TCP flow's congestion window by the same proportion. However, in this paper, when a TCP-P flow i detects a congestion event at time t and the elapsed time is not less than its minimum $srtt$, it responds like a normal TCP: It enters recovery mode and halves its own congestion window regardless of other parallel flows. Therefore, other TCP-P flows in the group can reduce their congestion window sizes less than the proportion shown in Equation (6) when the strength is $k > 1$.

Hence, the amount of decreases of congestion window sizes of other member TCP-P flows' become as follows:

$$W_j(t+) = W_j(t) * (\frac{1}{2} + \frac{N(k-1)}{2(N-1)k}), \ \forall j \neq i. \tag{9}$$

This equation is derived from the following equation to distribute the remaining amount of congestion window decrease among the other member TCP-P flows:

$$\begin{aligned} &\sum_{j=1}^{N} W_j(\frac{2k-1}{2k}) \\ &= W_0 N(\frac{1}{2} + \frac{k-1}{2k}) \\ &= W_0(1+N-1)(\frac{1}{2} + \frac{k-1}{2k}) \\ &= \frac{1}{2}W_0 + W_0(N-1)(\frac{1}{2} + \frac{N(k-1)}{2(N-1)k}). \end{aligned} \tag{10}$$

4 Evaluation

For the evaluation of TCP-P, we used the topology shown in Fig. 2. To emulate delays and packet losses in the Internet, we use NIST Net Emulator [12]. The NIST Net Emulator is implemented on a Linux machine and emulates the Internet by appropriately delaying and dropping packets. Because the network links in our experiments are fairly high-bandwidth (default 100Mbps) and the NIST Net delay parameters are large (50msec round-trip propagation delay,) we set the TCP parameters of the Linux end systems - such as `tcp_wmem` and `tcp_rmem` sizes - appropriately, rather than using system defaults. We also disabled the TCP time-stamping and TCP-Sack options to see the effects of our modification more clearly. By disabling TCP-Sack, Linux TCP works based on TCP-NewReno.

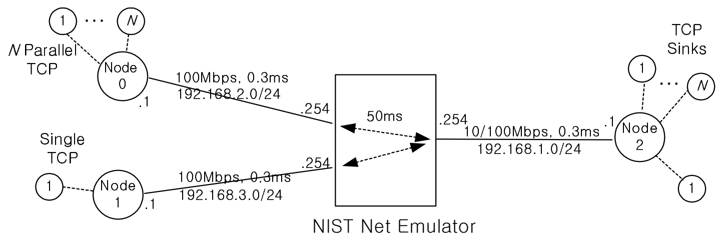


Fig. 2. Experiment Network

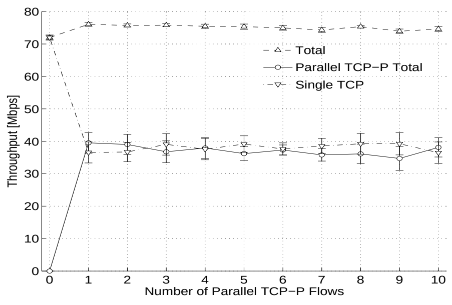
All the end-host nodes and the NIST Net Emulator are running on Linux PCs. The PCs we use for experiments are Pentium 4 or 3 machines with 10/100 Mbps Fast Ethernet network interface cards. Each Fast Ethernet card has an output queue of length 100 packets by default, and can be controlled if needed. Two TCP sender nodes, Node 0 and Node 1, run both on Redhat 9.0 with kernel 2.4.20-8. In machine Node 0 we installed a modified Linux kernel that supports. NIST Net Emulator and the TCP sink, Node 2, are running on Redhat Linux 7.2 with kernel 2.4.7-10.

For traffic generation and throughput measurements we use `iperf` [13], which supports parallel TCP flows and offers great flexibility for measurements. In all experiments in this section, every experiment was done for 100 sec to get an average value and repeated 10 times with 5 sec waiting time after each experiment unless told otherwise. Error bars in figures of this section represent 95% Confidence Interval of the data.

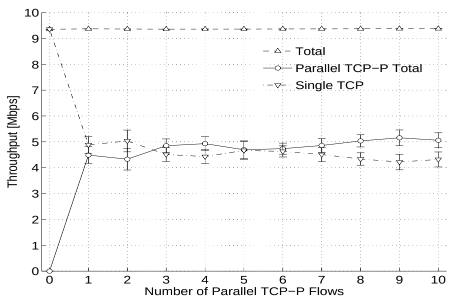
4.1 TCP Flow Groups with Strength $k = 1$

We first show the performance of TCP-P with $k = 1$. We open a group of parallel TCP-P flows from modified Linux kernel at Node 0 to a TCP sink Node 2 for 100 seconds, and a single unmodified TCP flow from another sender Node 1 to Node 2 for the same time. Fig. 3 (a) shows the experimental results with a varying number of parallel TCP-P flows from Node 0 with $k = 1$. This figure shows that the average of aggregated throughput of a group of parallel TCP-P flows of $k = 1$ with group size from 1 to 10 remains comparable to the average throughput of the single TCP flow from Node 1. In order to investigate the robustness of the TCP-P approach we repeated the experiments with a reduced bottleneck link speed by limiting the link speed from NIST Net Emulator to TCP sink node Node 2 from 100Mbps to 10Mbps. Fig. 3 (b) shows the experiment results with 10Mbps link. These results also show that TCP-P can regulate the aggressiveness of parallel TCP-P flows not to steal bandwidth from the single TCP flow, so that the throughput of the single TCP flow from Node 1 is comparable to that of total parallel TCP-P flows from Node 0 regardless of the group size N.

Figures in Fig. 4 illustrates some of the details of the operation of TCP-P. These figures are generated by `tcptrace` using `tcpdump` data on the sender Node 0 and Node 1. For the simplicity of comparison of the time-dependent behaviors

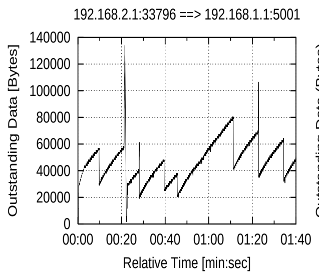


(a) 100Mbps Link

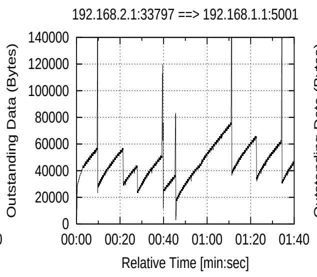


(b) 10Mbps Link

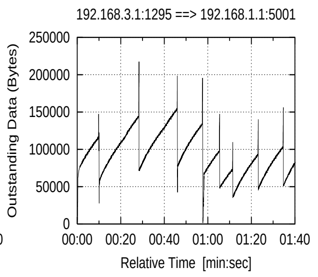
Fig. 3. Effect of N TCP-P Flows with $k = 1$ on a Single TCP Flow



(a) Flow 0



(b) Flow 1

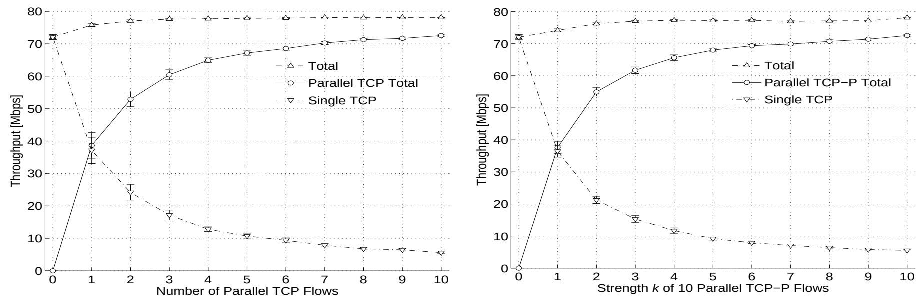


(c) Flow 2

Fig. 4. Observed Outstanding Packets of Flows

we open two parallel TCP-P flows with $k = 1$, Flow 0 and Flow 1, from Node 0 to Node 2, and one TCP flow, Flow 2, from Node 1 to Node 2. Other conditions are the same to the previous experiment, and all connections start at the same time and finish after 100 seconds. Average throughput achieved by the two TCP-P flows from Node 0 and the single TCP flow from Node 1 were 4.73Mbps and 4.78Mbps, respectively.

The figures show the amount of outstanding data of each TCP flow, from which we can infer the changes of congestion windows of TCP flows. The spikes in the figures represent Fast-Retransmission behaviors of TCP flows. Fig. 4 (a) and Fig. 4 (b) are for two TCP-P flows from Node 0. We can see in these figures that there are congestion window decreases *without* spikes, which indicates adjustments of the congestion window by the other TCP-P in the group. Compared to these two figures, the change in the congestion window of Flow 2 in Fig. 4 (c) always have a spike before a reduction.



(a) Unmodified k Parallel TCP (b) 10 Parallel TCP-P for different k

Fig. 5. Effects of Unmodified k TCP Flows and 10 TCP-P Flows with varying Strength k

4.2 TCP Flow Groups with strength $k > 1$

In the following, we illustrate how TCP-P effectively controls the magnitude of aggressiveness of parallel TCP flows according to the strength parameter k. We first present experiment results with unmodified parallel TCP flows in Fig. 5 (a). Node 0 opens k unmodified parallel TCP flows to Node 2 for 100 seconds, while Node 1 opens a single TCP flow to Node 2 for the same time. Fig. 5 (a) shows the average throughput of TCP flows from Node 0 and Node 1 for a varying number of unmodified TCP flows from Node 0. The single TCP flow from Node 1 achieves increasingly smaller throughput with increasing numbers of unmodified parallel TCP flows from Node 0. It illustrates the unfairness of parallel TCP flows mentioned in Sec. 1.

In comparison, Fig. 5 (b) shows the results of TCP-P in the same environment, except that we let Node 0 open a group of 10 parallel TCP-P flows to Node 2 with varying strength k. Fig. 5 (b) shows average throughput of parallel TCP-P flows and a single TCP flow when we control the aggressiveness of the group of parallel TCP-P flows. In the figure, with $k = 0$ we describe the case of no parallel TCP-P flows sending any traffic to the destination, so that only the single flow from Node 1 consumes all bandwidth. By comparing (a) and (b) in Fig. 5 we see that TCP-P scheme accurately controls the overall aggressiveness of a group of 10 parallel TCP-P flows according to k. 10 TCP-P flows with strength k show almost the same effect to a single TCP as k unmodified parallel TCP flows.

The steady-state throughput models for N parallel TCP and TCP-P flows with strength k have been derived and interested readers can refer to them in [14].

5 Conclusion

In this paper, we proposed TCP-P for aggregate control of parallel TCP flows. TCP-P scheme uses *strength* as a single - easily tunable - parameter to accurately control the aggressiveness of a group of TCP flows with respect to a single flow sharing the same bottleneck link. We showed that by employing TCP-P we can control the total aggressiveness or unfairness of parallel TCP flows against TCP flows from other nodes in a easily parameterizable and controllable way without requiring application modification. For future work, we are considering an adaptive control of the strength of parallel TCP flows.

References

1. Tom Dunigan: Net 100 Project (2004) URL: `http://www.csm.ornl.gov/~dunigan/netperf/parallel.html`.
2. S. Floyd and K. Fall: Promoting the Use of End-to-End Congestion Control in the Internet. IEEE/ACM Transactions on Networking **7** (1999) 458–472
3. D. Maltz and P. Bhagwat: TCP Splicing for Application Layer Proxy Performance. IBM Research Report RC 21139 (1998)
4. H. Balakrishnan, H. Rahul, and S. Seshan: An Integrated Congestion Management Architecture for Internet Hosts. In: ACM SIGCOMM. (1999)
5. Thomas Hacker, Brian Noble, and Brian Athey: Improving Throughput and Maintaining Fairness using Parallel TCP. In: Infocom. (2004)
6. Y. Gao, G. He, C. Hou, and S. Paul: COCOON: an alternate approach to end-host congestion management. submitted to IEEE Trans. on Computers (2002) URL: `stat.bell-labs.com/who/yuangao/papers/cocoon.pdf`.
7. J. Crowcroft and P. Oechslin: Differentiated end-to-end Internet Services using a Weighted Proportionally Fair Sharing TCP. ACM CCR **28** (1998) 53–69
8. V. Paxson: End-to-End Internet packet dynamics. In: ACM SIGCOMM. (1997)
9. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow: TCP Selective Acknowledgement Options. RFC-2018 (1996)
10. P. Sarolahti and A. Kuznetsov: Congestion Control in Linux TCP. In: Proceedings of Usenix 2002/Freenix Track. (2002)
11. Matt Mathis, Jeff Semke, Jamshid Mahdavi, and Kevin Lahey: The Rate Halving Algorithm for TCP Congestion Control (1999) URL: `http://www.psc.edu/networking/rate_halving.html`.
12. M. Carson and D. Santay: Tools: NIST Net: a Linux-based network emulation tool. ACM CCR **33** (2003) 111–126
13. Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs: Iperf Version 1.7.0 (2003) URL:`http://www.noc.ucf.edu/Tools/Iperf/`.
14. Soohyun Cho and Riccardo Bettati: Aggregate Control of Parallel TCP flows. Technical Report TAMU-CS-2004-11-1 (2004)