CONGESTION CONTROL SCHEMES FOR SINGLE AND PARALLEL TCP FLOWS IN HIGH BANDWIDTH-DELAY PRODUCT NETWORKS

A Dissertation

by

SOOHYUN CHO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2006

Major Subject: Computer Science

CONGESTION CONTROL SCHEMES FOR SINGLE AND PARALLEL TCP FLOWS IN HIGH BANDWIDTH-DELAY PRODUCT NETWORKS

A Dissertation

by

SOOHYUN CHO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Riccardo Bettati
Committee Members,	Wei Zhao
	A. L. Narasimha Reddy
	Dmitri Loguinov
Head of Department,	Valerie E. Taylor

May 2006

Major Subject: Computer Science

ABSTRACT

Congestion Control Schemes for Single and Parallel TCP Flows in High Bandwidth-Delay Product Networks. (May 2006)

Soohyun Cho, B.S., Korea University;
M.S., Korea University
Chair of Advisory Committee: Dr. Riccardo Bettati

In this work, we focus on congestion control mechanisms in Transmission Control Protocol (TCP) for emerging very-high bandwidth-delay product networks and suggest several congestion control schemes for parallel and single-flow TCP. Recently, several high-speed TCP proposals have been suggested to overcome the limited throughput achievable by single-flow TCP by modifying its congestion control mechanisms. In the meantime, users overcome the throughput limitations in high bandwidth-delay product networks by using multiple parallel TCP flows, without modifying TCP itself. However, the evident lack of fairness between the high-speed TCP proposals (or parallel TCP) and existing standard TCP has increasingly become an issue.

In many scenarios where flows require high throughput, such as grid computing or content distribution networks, often multiple connections go to the same or nearby destinations and tend to share long portions of paths (and bottlenecks). In such cases benefits can be gained by sharing congestion information. To take advantage of this additional information, we first propose a collaborative congestion control scheme for parallel TCP flows. Although the use of parallel TCP flows is an easy and effective way for reliable high-speed data transfer, parallel TCP flows are inherently unfair with respect to single TCP flows. In this thesis we propose, implement, and evaluate a natural extension for aggregated aggressiveness control in parallel TCP flows.

To improve the effectiveness of single TCP flows over high bandwidth-delay product networks without causing fairness problems, we suggest a new TCP congestion control scheme that effectively and fairly utilizes high bandwidth-delay product networks by adaptively controlling the flow's aggressiveness according to network situations using a competition detection mechanism. We argue that competition detection is more appropriate than congestion detection or bandwidth estimation. We further extend the adaptive aggressiveness control mechanism and the competition detection mechanism from single flows to parallel flows. In this way we achieve adaptive aggregated aggressiveness control. Our evaluations show that the resulting implementation is effective and fair.

As a result, we show that single or parallel TCP flows in end-hosts can achieve high performance over emerging high bandwidth-delay product networks without requiring special support from networks or modifications to receivers.

To my parents Hyungsik Cho, Hyunnam Son, and my wife Eunjoo Kim

ACKNOWLEDGMENTS

First, I would like to express my deep appreciation to my advisor, Dr. Riccardo Bettati. Without his guidance and invaluable comments on my research and patience with me, my research would not have been successful. I am greatly indebted to him for all of his support. I would also like to thank my Ph.D. committee members: Dr. Wei Zhao, Dr. A.L. Narasimha Reddy, and Dr. Dmitri Loguinov. Dr. Zhao allowed me to work with his students in the Real-Time Systems Group and used his valuable time to attend my presentations from NSF in Washington, DC. The classes with Dr. Reddy improved my understanding of networks. Further, his insight and advice on my research greatly improved this dissertation. From Dr. Loguinov I have learned not only trends in current network research, but also how to start research and the attitude of a researcher.

I also want to show my appreciation to current and former members of the Real-Time Systems Group. Especially, I deeply appreciate Shengquan Wang. He always tried to help me in everything, and without him I could not have even started the experiments in this dissertation. I also would like to thank Dr. Xinwen Fu, Dr. Sangig Rho, and Dr. Byung-Kyu Choi for their kindness during my study and valuable advice on my research. I would like to thank all of my family members, especially my wife Eunjoo Kim. She was my last resort whenever I felt lost and could not find an escape route. Without her dedication, endless love and care, my graduate studies could not have been continued.

Last, I pray to God for my father to stay alive until I return home. I believe he is waiting for my successful completion in study even though he is unconscious in a hospital bed. I hope my fulfillment may wake him from his deep sleep. I send my love to my mother and father who have always believed in their humble son's ability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

A. Introduction

The Internet has evolved from small research networks into the backbone infrastructure network for worldwide communications. It has become an almost indispensable tool in the daily lives of most people today. However, differently from the traditional telephone networks, the Internet is not a tightly managed or controlled communication network. The service provided by the current Internet is basically best-effort, which does not guarantee in-time delivery of data packets or even the delivery itself. The Internet allows users or applications the freedom in choosing the best way for their data delivery. Because of this freedom given to users and the lack of tight management of the network resources, there is always a possibility that the behavior of a set of selfish users may eventually lead to congestion collapse of the Internet, in which nobody can communicate with each other.

To avoid this, it is desirable that all the elements composing the Internet (end-hosts, servers, network equipment, etc.) work cooperatively. Significant amounts of research has gone into this area for decades, with the objective to provide a stable and efficient Internet. One of the main research directions to avoid the congestion collapse or fair bandwidth sharing among users of the Internet is to embed effective congestion control mechanisms in the end-hosts. Current Transmission Control Protocol (TCP) [1] represents the achievement of this effort. The success of the flourishing, worldwide Internet depends heavily on the protocol's ability to keep networks stable while using them effectively. The main topic studied in this work is TCP congestion control

The journal model is *IEEE Transactions on Automatic Control*.

schemes for the Internet. Because TCP is the main flow control mechanism and the most widely used protocol in the Internet, a small upgrade in the TCP congestion control mechanism can result in a large increase in the efficiency of the Internet.

Although the stability of the current Internet is mainly based on the end-to-end congestion control mechanism embedded in TCP, another research direction to improve the performance of the Internet is to embed mechanisms into the network equipment such as routers. There has been a significant amount of research dedicated to such mechanisms. RED [2] and XCP [3] are examples that represent efforts in this direction. Further, there have also been schemes proposed to support service differentiation in the Internet. IntServ [4], DiffServ [5], and Proportional Differentiation [6] are well-known examples of service-differentiation schemes. It is understood that IntServ is for strictly prioritized services, DiffServ and Proportional Differentiation are to provide relatively prioritized services to incoming flows. However, history has shown that it is unlikely in practice for operators of Internet routers/switches to adopt schemes that might burden their equipment with additional overhead. The issues of effectiveness and scalability (or deployability) that have plagued many such schemes still need to be addressed. We believe that the best place for improvement of the Internet is the end-host and, therefore, primarily focus on mechanisms that can achieve better performance without requiring special support from network infrastructures.

During the last decades, to overcome increasing heterogeneity and diversity in network elements that compose the Internet, TCP has continuously evolved from a simple error control mechanism not much different from “Selective Repeat ARQ” [7] to the current, state-of-art level congestion window control that defies easy understanding. The heterogeneity of network situations that TCP covers ranges from slow links such as dial-up modem lines to the thousands of miles of trans-Pacific optical links. Also, the wide use of wireless networking requires TCP to further evolve to become more intelligent to differentiate between packet losses caused by transmission error or noise from those caused by congestion. Furthermore, continuing evolution in transmission technology and ever-increasing demands for very-high bandwidth services pose a number of new challenges to TCP.

To deal with such challenges, there have been many proposals and suggestions with different approaches for TCP congestion control. However, the *de facto* standard for the TCP congestion control is still a loss-based approach, as exemplified by, in TCP-NewReno [8] and TCP-SACK [9]. By controlling the sender's congestion window size in response to packet loss in the network, loss-based congestion control adjusts the amount of outstanding data. However, a pure loss-based approach has the inevitable shortcoming that it cannot detect congestion until well after it happens. There have been proposals to compensate for this with delay variation information measured at end-hosts. TCP-Vegas [10] and FAST-TCP [11] are examples that represent achievements in this direction. However, it is known that pure delay-based approaches respond to congestion well before loss-based schemes, and thus cannot compete effectively. These schemes eventually lead to low-priority flows by giving up bandwidth to loss-based TCP.

The limited throughput achievable by single-flow TCP over high bandwidth-delay product networks has increasingly become an issue. TCP's flow control and error control mechanism has been shown to be a limiting factor for such networks: Given TCP's lack of ability to distinguish non-congestion packet loss from congestion related packet loss, its throughput is inversely proportional to the round-trip time. To overcome this ineffectiveness, recently, several proposals to modify the behavior of TCP for high bandwidth-delay product networks have been introduced. These proposals include HS-TCP [12], STCP [13], H-TCP [14], FAST TCP [11], BIC-TCP [15], and LTCP [16]. These have shown good performance in the utilization of high bandwidth-delay product networks. However, we will examine the fairness of these proposals and show that most of these proposals have serious fairness problems with respect to flows that use standard TCP, e.g., TCP-SACK. In Chapter IV, we propose a new TCP congestion control scheme called Adaptive TCP, which is effective in high bandwidth-delay product networks while maintaining fairness close to that of standard TCP when it competes against other TCP flows.

Differently from these approaches, high utilization over high bandwidth-delay product connections can be achieved without modifications to TCP itself, by opening multiple concurrent parallel TCP flows. If users at end-hosts establish multiple TCP connections simultaneously to the same destination, they can achieve higher throughput than users with single TCP flow. This is because, with standard TCP, the total congestion window size increase and recovery of parallel TCP flows are faster than for a single flow and each loss of a packet in a set of parallel TCP flows causes the congestion window of only a single TCP flow to half instead of halving all parallel TCP flows' windows [17]. As a result, the achievable throughput of parallel TCP flows, given the same packet loss probability and round-trip time, is bigger than that of a single TCP flow.

Opening multiple parallel TCP flows is unfair in terms of throughput with respect to hosts who open a single TCP flow when they compete for the same bottleneck link. There have been several efforts such as Fractional/Combined TCP flows [18] to achieve high performance with parallel TCP flows while constraining the unfairness of parallel TCP flows. However, we show that improving performance of parallel TCP flows while maintaining fairness (or TCP-friendliness) to single TCP flows is not easy to achieve. In Chapter III, we propose an aggregated aggressiveness control framework and its implementation called TCP-P that controls aggregated aggressiveness of parallel TCP flows to be comparable to that of a given number of parallel TCP flows according to the *strength* parameter of the group.

High-speed connections are often used by servers or very busy hosts, where multiple, *parallel* connections exist simultaneously. Similarly, in overlay networks, such as Content Distribution Networks (CDN) [19] and peer-to-peer networks [20], it is common to *aggregate* traffic at the edges of the network and forward it between overlay nodes as flows aggregate. Flows in such systems tend to share long portions of paths (and bottlenecks), and benefits can be gained by sharing congestion information. In Chapter II we leverage this correlation among parallel TCP flows and propose a congestion control scheme for parallel TCP flows. In the scheme, parallel TCP flows control their congestion windows using additional congestion related information from other parallel TCP flows in their group.

While the fairness (or TCP-friendliness) of the TCP proposals is considered as an important factor in determining the performance of the proposals, it is not usually mentioned what will happen if many such TCP flows compete against standard TCP flows. By simply opening multiple *fair* TCP flows users can easily achieve more than a fair share regardless of how fair the TCP was designed to be. Thus, we believe that parallel-level performance such as fairness to single TCP flow should be considered as one of the important criteria in evaluating TCP proposals. Finally, in Chapter V, by combining the aggregated aggressiveness control mechanism of TCP-P in Chapter III with the adaptive aggressiveness control mechanism of A-TCP in Chapter IV, we propose a new TCP scheme called Adaptive TCP-P (A-TCP-P) whose performance in terms of effectiveness and fairness are not affected by the number of parallel flows.

In Figure 1 we illustrate the relationship between existing TCP schemes and the extensions proposed in this dissertation. The relationship between the schemes is represented by the arrows. For example, TCP becomes parallel TCP when a node opens more than one active TCP to the same destination. The three gray circles

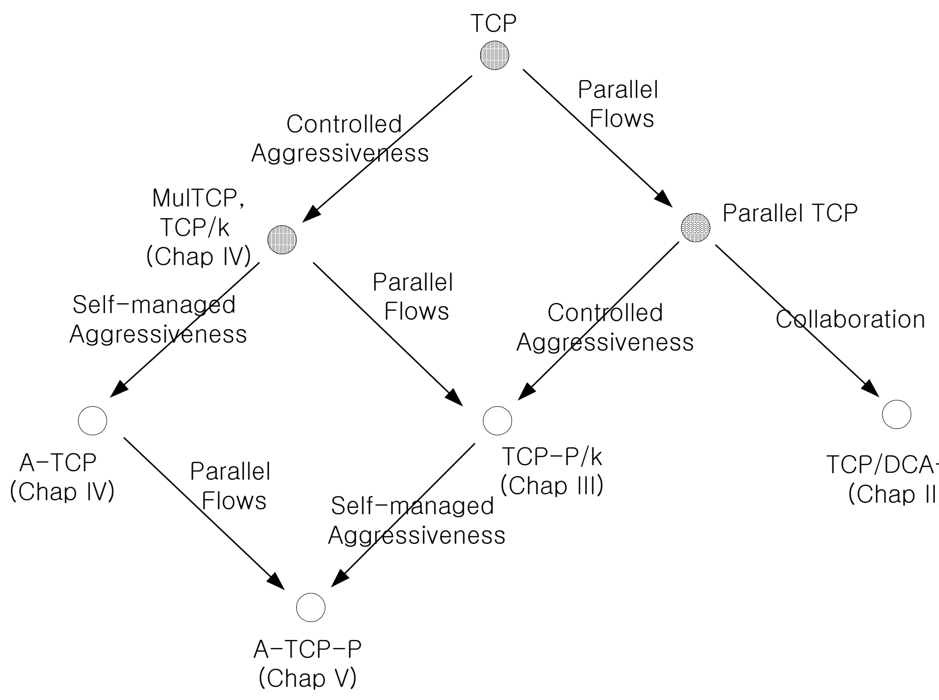


Fig. 1. Relationship Diagram

in the figure represent relationship between existing schemes: TCP, MulTCP [21], and parallel TCP. MulTCP and TCP/k in Chapter IV are similar to each other as both try to emulate the aggressiveness of parallel TCP with single-flow TCP while their implementations are slightly different. Transparent circles represent schemes proposed in this dissertation: *i*) TCP/DCA-C in Chapter II based on collaboration among parallel TCP flows, *ii*) TCP-P in Chapter III that controls aggregated aggressiveness of parallel TCP flows, *iii*) A-TCP in Chapter IV applies the adaptive aggressiveness control on single-flow TCP, *iv*) A-TCP-P in Chapter V employs the adaptive aggregated aggressiveness control on parallel TCP.

The remainder of this dissertation is structured as follows: In Chapter II, we suggest a collaborative congestion control scheme called TCP/DCA-C for dynamic congestion information sharing among parallel TCP flows. In Chapter III we propose aggregated aggressiveness control on groups of TCP flows. In Chapter IV we propose an adaptive aggressiveness control scheme for single TCP flows, which achieves effective utilization of high bandwidth-delay product networks without causing the fairness problem. In Chapter V we extend the scheme proposed in the previous section from single flows to parallel flows: We extend TCP-P by adopting an adaptive aggregated aggressiveness control mechanism based on group-wide information. Chapter VI concludes this dissertation.

CHAPTER II

COLLABORATIVE CONGESTION CONTROL IN PARALLEL TCP FLOWS

A. Introduction

A loss-based TCP congestion control scheme such as TCP-NewReno [8] adjusts the amount of outstanding data by controlling the sender's congestion window size in response to packet loss in the network. Although this loss-based approach has been the *de facto* standard for the TCP congestion control for decades, it has an inevitable shortcoming in that it can detect congestion only after it has happened. To solve this problem many researchers proposed alternative methods to replace loss-based congestion control with delay-based schemes, such as Delay-based Congestion Avoidance (DCA) [22] or TCP-Vegas [10], or to combine delay-based fine tuning (such as TCP/Dual [23] and TCP/DCA [24]) with loss-based (so called, coarse) congestion control. TCP/Dual decides that congestion is impending if a round-trip time is larger than the average of the maximum and minimum round-trip time. In TCP/DCA, sampled round-trip times are compared to long-term average and standard deviation of round-trip times to decide whether congestion is imminent. The rationale behind the various delay-based approaches is that the increase of round-trip time delay is due to queuing delay at switches and can be used as an indicator for impending packet losses.

Recent transmission technology improvements have resulted in much higher bandwidth available to Internet hosts, thus significantly increasing the delay-bandwidth product over end-to-end connections. In this chapter we use the following observations: First, high-speed connections are often used for servers or very busy hosts, where multiple connections exist simultaneously. Similarly, in overlay networks, such as Content Distribution Networks (CDN), it is common for much of the traffic to be forwarded between overlay nodes as flows aggregate. Finally, techniques such as TCP splicing [25] technique suggest to splice TCP flows through a proxy to improve forwarding performance of TCP by overcoming heterogeneity of networks. Flows in such systems tend to share long portions of paths (and bottlenecks), and improvements may be gained by sharing congestion information.

In order to do so, we group multiple TCP flows from a node according to their destinations, so that TCP flows in a group can share their congestion information with each other. We modify TCP flow control to adopt a simple delay-based congestion detection mechanism and make the flows exchange their congestion events within the group. By doing this, we expect that parallel TCP flows can (*i*) control their congestion windows (so, their sending rates) more accurately, so that they can (*ii*) reduce packet losses from flows in the group and (*iii*) maintain small and stable queue sizes in the network compared to those of unmodified parallel TCP flows.

The remainder of this chapter is organized as follows: Section B surveys related work on parallel TCP flows. Section C presents the proposed collaborative congestion control scheme in parallel TCP flows with the definition of groups and events. Section D shows that this scheme can be used to improve various performance measures of parallel TCP flows. Section E summarizes this chapter.

B. Related Work

There has been significant research done to improve the congestion control when multiple flows are sending traffic to similar destinations from a single node. T/TCP (TCP for Transactions) [26] proposes temporal information sharing among parallel TCP flows. A T/TCP host caches previous TCPs' congestion control information, e.g., Round Trip Time (RTT) and Maximum Segment Size (MSS). The cached information is used when a new TCP connection is established to the same remote hosts. Ensemble-TCP [27] suggests both temporal and ensemble information sharing using a common TCB (TCP Control Block) among concurrent connections. A new connection uses existing connection's TCB and congestion control of all TCP flows are done based on an aggregated congestion window.

Similarly, Congestion Management architecture (CM) [28] proposes an integrated congestion control and loss recovery scheme for parallel TCP connections. There is a single congestion window for the set of TCP connections between a sender and receiver. The integrated control block adjusts the total amount of unacknowledged data that the set of connections can put in the network. COCOON [29] has a similar approach to ours in that it uses an aggregate control of a group of TCP flows. Specifically, it changes the congestion window size of a TCP flow whenever other flows in the group experience packet losses.

Recently, attention has focused on prioritizing TCP flows. TCP-Nice [30] and TCP-LP [31] both try to control background TCP traffic not to interfere with more important foreground traffic. TCP-Nice is based on TCP-Vegas [10] and controls the amount of outstanding data to be less than that of foreground traffic. TCP-LP uses one-way delay increase as an early indicator of impending congestion and halves the congestion window of low-priority flows earlier than standard TCP. RCS [32] assumes that the network supports a priority scheme. It sends dummy (low priority) packets to probe available bandwidth before it determines its sending rates for multimedia traffic. SAReno [33] adopts the Shortest Remaining Processing Time First (SRPT) scheduling for TCP flows. To achieve better average bit transmission delay (i.e., delay/file size), this scheme changes priorities of TCP flows according to their residual size of data. In this scheme, the priority of each TCP flow is represented by its linear increase rate and multiplicative decrease rate.

The scheme proposed in this chapter is different from the above methods because each TCP flow in our scheme maintains its own congestion window and uses extra measurement information from *other* TCP flows in the group and responds to them independently. In addition, the particular realization described in this chapter is based on a delay-based congestion avoidance mechanism as opposed to packet loss.

C. Collaborative Congestion Control

In networks that do not provide congestion control mechanisms, such as Explicit Congestion Notification (ECN) [34], acknowledgement (ACK) packets are the only indicators for TCP traffic senders to estimate the current network state. In an increasing number of service deployment scenarios, however, TCP flows get aggregated, and so share both large portions of their paths and likely the congested links on them. As a result, parallel TCP flows would benefit by using the other flows' information in addition to their own to estimate the network status.

We use the term *group* to indicate a set of flows from a node that have the same destination node and that share the path between senders and destination nodes. This group concept can be extended to aggregated flows between clusters of nodes at the sender or the receiver side. For example, we can take advantage of IP's hierarchical addressing [35] or systems like GNP [36] to cluster end-hosts, so that connections between clients share most of their path, thus allowing for an effective grouping of flows that not necessarily share both source and destination nodes. However, in this chapter we only group flows with the same source and destination pairs.

Three types of congestion related indicators are available to the sending host: (a) round-trip time history, (b) duplicate acknowledgement (DUPACK) events, and

(c) time-out events. While all three indicators can be used in collaborative congestion control, we focus on (a), round-trip time history. The rationale for this is that DUPACK and time-out events, while necessary for flow and error control for the affected flows, happen *after* congestion, which is typically too late for use for preventing packet losses of *other* TCP flows in the group. We therefore extend traditional TCP by adding an inter-flow, delay-based, congestion control method in addition to the existing loss-based congestion control mechanism.

We call our scheme TCP/DCA-C because (i) parallel TCP flows in our scheme adopt a delay-based congestion avoidance scheme and (ii) flows share delay events detected by TCP flows in the group as indictors of imminent congestion collaboratively (i.e., C stands for *Collaborative*). To detect imminent congestion, a TCP/DCA-C flow monitors its round-trip time, and uses a threshold-based approach with the following threshold level T:

$$T = rtt_{\min} + \gamma * (rtt_{\max} - rtt_{\min}). \tag{2.1}$$

Here, $rtt_{\min}$ and $rtt_{\max}$ represent the maximum and minimum round-trip time of a TCP flow respectively. If a round-trip time is larger than the threshold, then a TCP flow in our scheme interprets this as an indicator for impending congestion. The value for parameter γ used in this chapter is 1/2.

Algorithm 1 illustrates the details of the mechanics of TCP/DCA-C. As a basis, we use TCP-Reno. The TCP/DCA-C mechanism keeps track of RTTs to generate indicators of possible congestion. Flow grouping is done using a registration scheme: during connection establishment a new flow either creates and registers to the new group or register to an existing group. When a TCP flow is torn down it unregisters from the group. A TCP/DCA-C flow in a group behaves the same way as a normal loss-based TCP does, except that (i) it responds to delay-based congestion events as

Algorithm 1 TCP/DCA-C

```
At connection setup:
1: if a group for this flow exist then
2:    register to the group;
3: else
4:    create a new group and register to the group;
5: end if

During connection life time: same as TCP-Reno except for:

(1) Every $rtt$ updates:
1: update $rtt_{min}$ and $rtt_{max}$;
2: if $rtt > rtt_{min} + \gamma(rtt_{max} - rtt_{min})$ then
3:    Send a signal and $cwnd$ to TCP flows in the group;
4:    if $cwnd > ssthresh$ then
5:       $cwnd \leftarrow cwnd - \alpha \cdot cwnd$
6:    end if
7: end if

(2) When a signal is received from other flows in the group of size $N$:
1: if $cwnd >$ signal sender's $cwnd$ then
2:    $cwnd \leftarrow cwnd - \alpha \cdot cwnd/(N-1)$
3: end if

At connection tear down:
1: unregister from the group;
2: if no flow exists in the group then
3:    delete the group;
4: end if
```

well if it is in congestion avoidance phase (i.e., $cwnd > ssthresh$), (*ii*) it passes its delay events to other flows in its group, and (*iii*) it reacts to incoming indicators from other flows. The value for the parameter α, which decides the amount of reduction in congestion window of a TCP flow when it detects delay increase events from its own ACKs, is set to 0.125. This value is taken from the decrease ratio of congestion window suggested in DCA [22]. In DCA the sender decreases its congestion window if the normalized delay gradient is larger than 0.

When a TCP/DCA-C flow sends its delay event as a signal to other flows in the group, it also sends out its current congestion window size. TCP/DCA-C flows in the same group reduce their congestion windows, but do so only if the congestion window of the signal sender is smaller than their own. Furthermore, TCP/DCA-C adjusts the amount of congestion window reduction in response to indicator signals from other flows inversely proportional to the size of its group (i.e., the number of flows in the group). This compensates for group size effects, since more flows in the group would generate more indicator signals.

It is reasonable to assume that the expected size of congestion windows of all TCP/DCA-C flows in a group are similar because we define groups based on source-destination pairs and the path for a group is fixed. Furthermore, with a group of size N, we can reasonably assume that the number of flows sending congestion notifications with a smaller congestion window to a particular flow in the group is typically not more than $(N-1)/2$, as the remaining $(N-1)/2$ flows have a larger or equal congestion window than the particular flow. As a result, if all the other flows in the group simultaneously detect delay events then the expected amount of adjustment will not be more than $\alpha/(N-1) * (N-1)/2 = \alpha/2$ of the congestion window of a flow according to step (2) in Algorithm 1, regardless of the group size.

The threshold-based approach we use for the detection of imminent congestion is

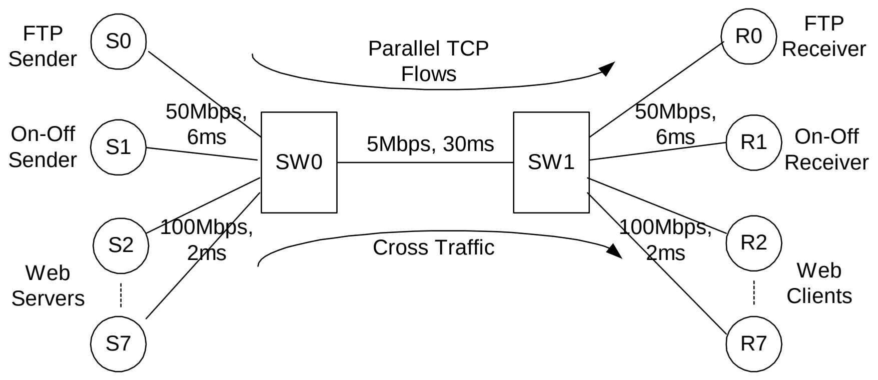


Fig. 2. Test Network

similar to that used in TCP-Nice [30], which counts the rate of these events and if the rate is larger than some threshold value it responds by reducing its congestion window by half. However, TCP-Nice is designed to minimize disturbances to foreground traffic. In our scheme, TCP/DCA-C flows in a group respond to each of the events by reducing their window size less than half of its current congestion window size (i.e., $\alpha < 0.5$) and sending the event as an indicator signal to other TCP flows in its group.

To compare the effectiveness TCP/DCA-C scheme with single-flow based schemes, we also define TCP/DCA-S (i.e., S stands for *Self*), which only uses its own delay events for the fine-tuning of its sending rate. This scheme is implemented by disabling the response of a TCP/DCA-C to signals from other TCP flows shown in Case (2) in Algorithm 1. In the next sections we will show using simulation that our TCP/DCA-C scheme in parallel TCP flows can improve the performance of parallel TCP flows in various network situations.

D. Simulation Results

We used ns-2 [37] Ver. 2.27 as a simulator for all the experiments. We implemented TCP/DCA-C by constructing a *group* management structure for parallel TCP flows to exchange information with each other. The TCP module used in our scheme was built by inheriting from the TCP-Reno module. All other TCP flows we use in the simulation are TCP-Reno. We use a fixed segment size of 1000 bytes for all connections including UDP. The simulated network we use is shown in Figure 2. The default buffer size at the output port of the bottleneck switch node SW0 is set to 36 packets, which is the bandwidth-delay product value of the bottleneck link between the two switch nodes SW0 and SW1. The advertised window sizes of TCP receivers were set to be large (e.g., 200 packets) enough for TCP senders to fill network resources.

FIFO Drop-Tail scheduling is used at the outport of Node SW0 when we test the TCP schemes that do not require network support. When we measure the performance of TCP/ECN flows, which require support from network, we configure an active queue management scheme at the bottleneck link. TCP/ECN responds to marked ACKs the same way it does to packet losses. For the ECN support in the network we run RED [2], one of the most popular active queue management (AQM) mechanisms, at the output buffer of Node SW0 and enable marking instead of dropping of packets. The parameters used for RED in all simulations are left to the default values of the ns-2 distribution except for a mean packet size of 1040 bytes. The default settings of RED give us $min_{th} = 5$, $max_{th} = 15$, $w_q = 0.001663$, and $max_p = 0.1$.

In the following, we compare the performance (in terms of goodput, packet loss rate, delay, and jitter) of TCP-Reno, TCP/DCA-S, TCP/DCA-C, and TCP/ECN for the case of no cross traffic, square-waved UDP cross traffic, and web cross traffic.

1. No Cross Traffic Case

In the first simulation we do not generate cross traffic in the test network shown in Figure 2. We establish two FTP flows (Flow 0 and Flow 1) from the sender node S0 to illustrate the difference of the various schemes. In this simulation we compare the results of our collaborative congestion control scheme TCP/DCA-C and its variants (TCP/DCA-S) with unmodified TCP flows and TCP flows with ECN. A sender node S0 opens two FTP flows at the same time to a receiver node R0 for 50 seconds.

Figure 3 shows the queue size changes in the bottleneck link from time 14 sec to 20 sec of the simulation run when we use different schemes for the two TCP flows. The graphs show that TCP/DCA-C maintains a small queue size and variation. With this result we expect that TCP/DCA-C will have low delay and jitter for packets arriving at the receiver. Unmodified TCP flows show large fluctuations of the queue size. The queue size of TCP/ECN is generally small as is to be expected, given the help from the network infrastructure.

In Figure 3 we also show the congestion windows of the two parallel TCP flows in each scheme for the same time period. From the graphs, we can see the TCP flows in the TCP/DCA-C scheme maintain stable congestion windows. TCP/DCA-S also shows stable congestion windows. However, we can see that there is significant difference in sizes of congestion windows between two TCP/DCA-S flows even though they are going to the same destination from the same source node. This shows that *lock-out* [38] is happening for TCP/DCA-S flows, which results in unfairness in throughput between the two flows. Lock-out occurs because TCP/DCA-S is a single-flow based scheme, where flows respond to their own congestion events only. This example illustrates the benefit of sharing congestion signals across flows.

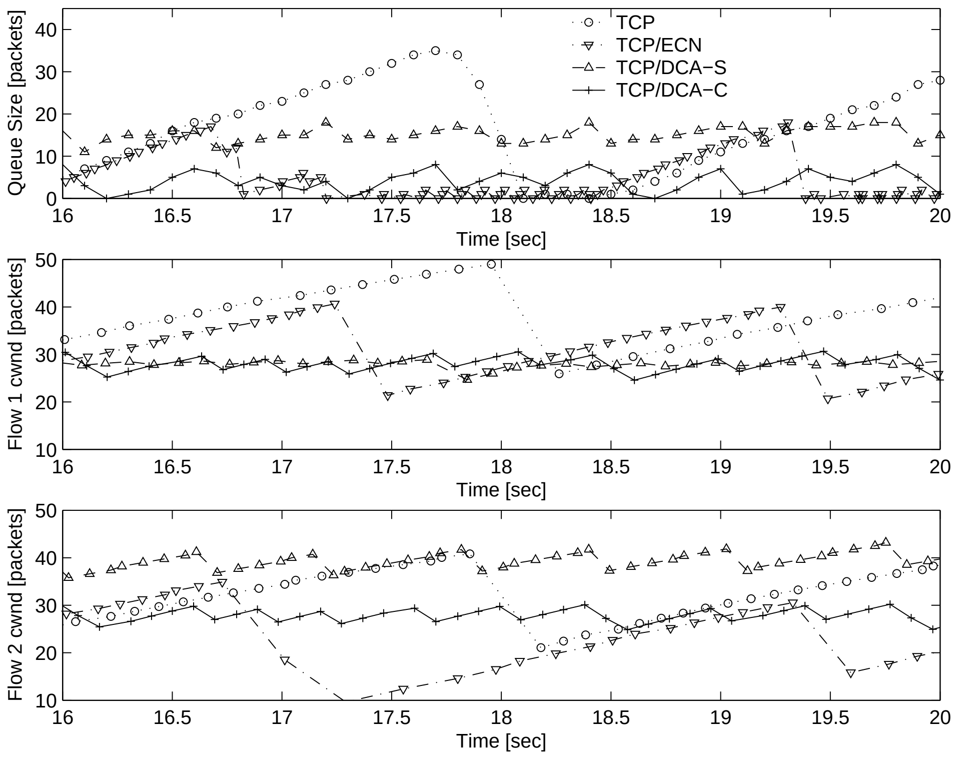


Fig. 3. Queue Size and Congestion Window Changes

In Table I we show performance measurement data of each scheme when two FTP flows start randomly from 0 to 0.1 sec and finish at 50 sec. All data shown in the table is obtained from averaging 10 simulation runs. The unit of aggregated goodput in the table is kbps and it is calculated by dividing the total number of packets arrived at the two FTP receivers by the simulation time. Except for TCP/ECN, the schemes do not show much difference in aggregated goodputs. The "Loss" column in the table shows percentage of lost packets of the two flows. Most of the losses of TCP/DCA-C flows happened during the initial slow-start phase. After they enter congestion avoidance phase they show little packet losses at all. Unmodified TCP and TCP/ECN showed significant packet losses during congestion avoidance phase as well.

The time unit in the "Delay" column is msec. The column shows average and

Table I. Aggregated Goodput, Loss, Delay, and Fairness of Two TCP Flows with Start Time [0, 0.1] sec

	Goodput (avg./std.)	Loss (avg.)	Delay (avg./std.)	Fairness (avg.)
TCP	4705.2/9.6	0.41	71.8/19.6	0.997
TCP/ECN	4497.7/30.8	0.31	53.5/9.29	0.996
TCP/DCA-S	4784.8/7.9	0.29	68.0/5.41	0.972
TCP/DCA-C	4740.3/37.9	0.26	51.7/5.38	0.999

standard deviation of one-way delays incurred by packets of the two TCP flows. In the column TCP/DCA-C flows show higher performance than TCP/ECN. To measure fairness in goodputs of the two flows we use the fairness index from Chiu and Jain [39] as shown below:

$$FI(x_1, x_2, .., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} {x_i}^2}. \tag{2.2}$$

The measured data in the last column of the table show that single-flow based TCP/DCA-S is less fair than the other schemes, just as we expected from Figure 3.

2. On-Off CBR Cross Traffic Case

In this section we first apply On-Off (i.e., square-waved) Constant Bit Rate (CBR) traffic over UDP as cross traffic in the setup described in Figure 2. The CBR traffic comes from Node S1 and goes to Node R1 to disturb the TCP flows from Node S0 to Node R0 when it is On. The parallel TCP flows start randomly during the interval between 0 and 0.1 sec, and the On-Off CBR flow start in Off state at time 0 sec. The sending rate of the CBR flow when On is set to 60% of the bottleneck link, and durations of On and Off periods are both set to 10 sec. Intervals between CBR packets are randomly distributed to introduce jitters by enabling the *random_* parameter for CBR in ns-2.

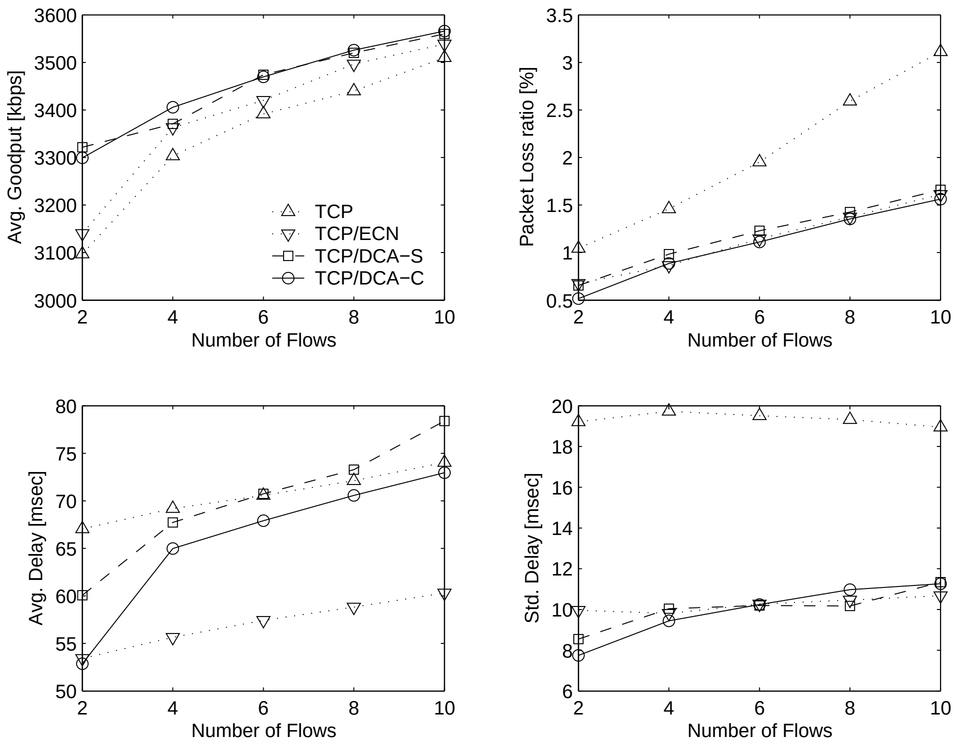


Fig. 4. On-Off Cross Traffic, Different Number of FTP Flows

Figure 4 shows the simulation results with various numbers of parallel TCP flows from Node S0 to Node R0. All the data shown in the figure are averages of 10 independent simulations of 50 second duration each. The figure shows that parallel TCP/DCA-C flows achieve high aggregate goodput and small packet loss ratios and jitters (we define jitter as standard deviation of one-way delays). One-way delays incurred by TCP/DCA-C packets are smaller than those of unmodified parallel TCP flows but larger than those of TCP/ECN flows. However, note that TCP/DCA-C scheme does not require support from the network nodes while TCP/ECN scheme does. In fact, TCP/DCA-C uses information acquired by TCP flows in its group at the sender-node only. In Figure 5 we also show simulation results obtained when we changed bottleneck buffer sizes from 30 to 90 with 6 parallel TCP flows. The figure shows that the relative performance of the TCP/DCA-C scheme compared to other

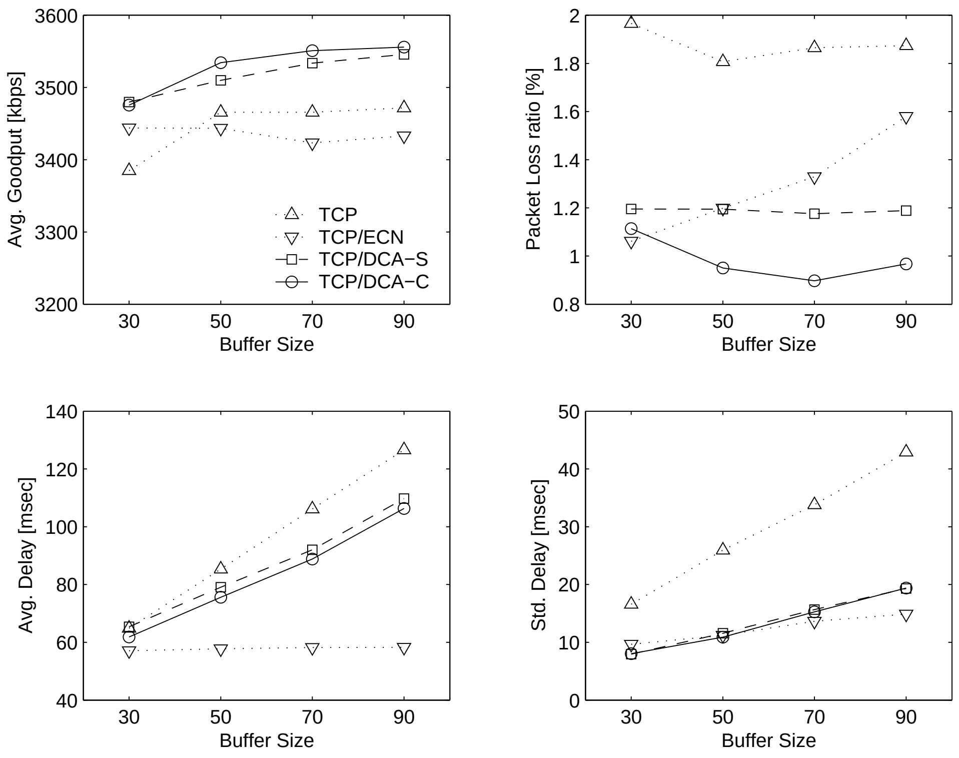


Fig. 5. On-Off Cross Traffic, Different Buffer Sizes

schemes is not affected by buffer size changes.

In Figure 6 we show experiment results when we change the periods of On-Off UDP cross traffic while the number of flows and the bottleneck buffer size are fixed: We change the periods of the On-Off UDP cross traffic from 0.05 to 12.8 second (i.e., 0.05, 0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4, 12.8 sec). The durations of On and Off states are both set to half of each period. The number of flows from Sender Node S0 is fixed at two, and the bottle-neck buffer size is set to 36 packets for all the experiments. The CBR UDP rate from Sender Node S1 is again 60% of the bottle-neck bandwidth (i.e., 3Mbps) when it is On, and zero cross traffic when it is Off. Therefore, the average available bandwidth that can be utilized by TCP flows is maximum 3.5Mbps. We use this maximum achievable bandwidth to normalize the goodput of TCP flows

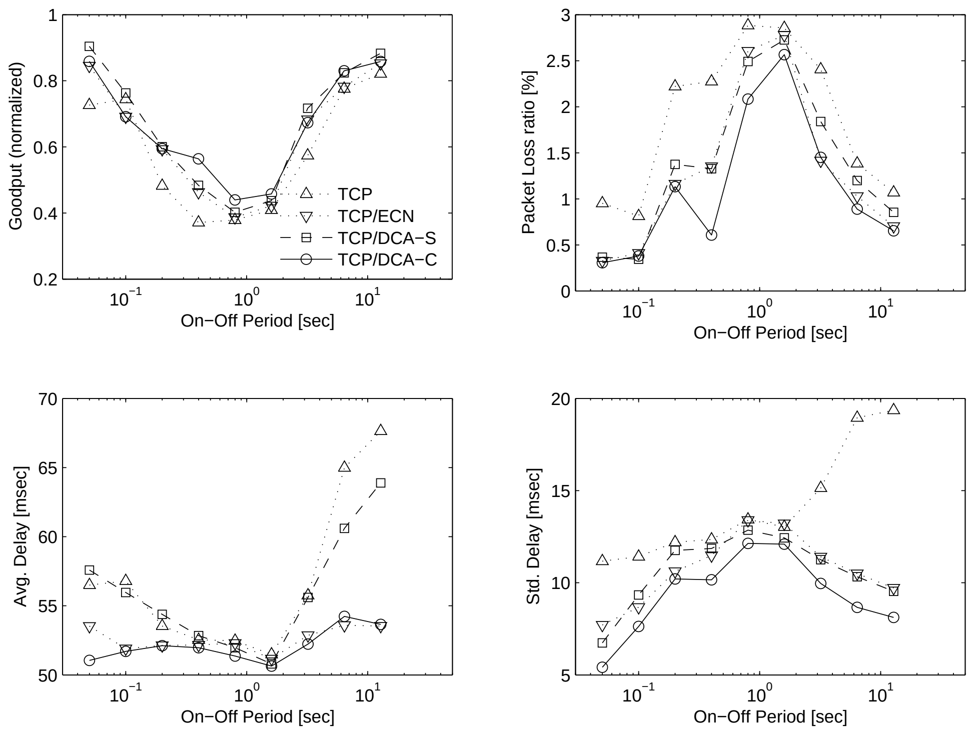


Fig. 6. On-Off Cross Traffic, Different Period

from experiment. This experiment have been suggested and used in [40] to measure how quickly TCP congestion control schemes utilize available bandwidth. From the experiment results in Figure 6 we can see that parallel TCP/DCA-C flows achieve small packet loss ratio and low delay and jitter while maintaining similar goodput compared to other schemes.

3. Web Cross Traffic Case

Figure 7 shows the experiment results in the presence of web cross traffic instead of On-Off traffic. The cross traffic is established between web server nodes (S2 to S6) attached to the switch node SW0 and web client nodes (R2 to R6) attached to the switch node SW1. All the data shown in the figure are averages of 10 simulation runs obtained of 50 second duration each. For the simulation of web traffic we adopted a

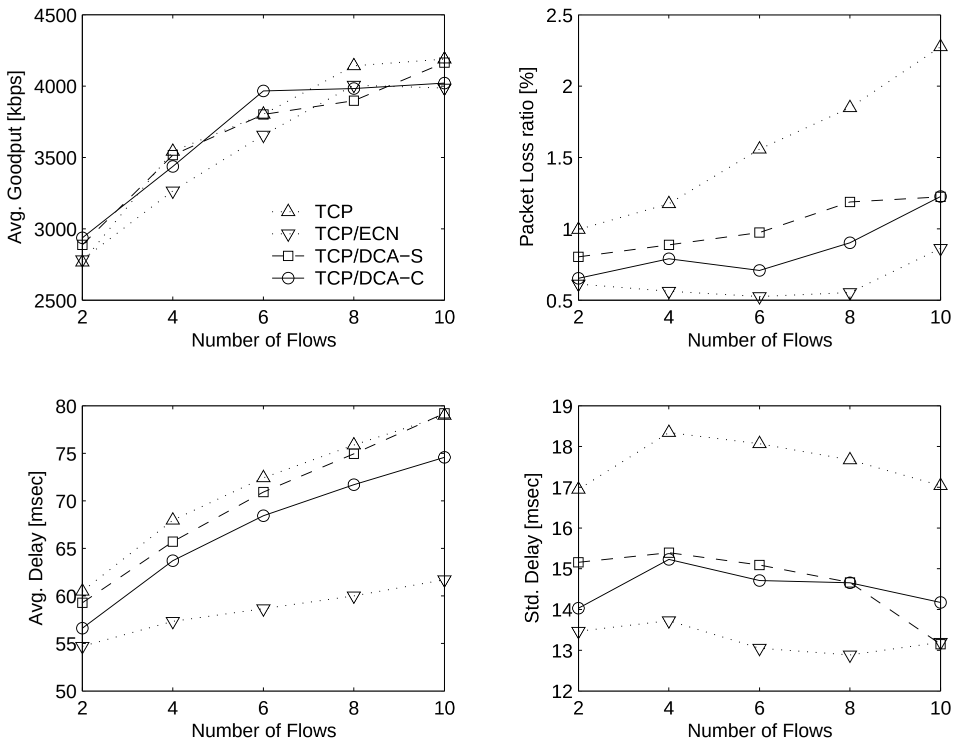


Fig. 7. Web Cross Traffic, Different Number of FTP Flows

model suggested in [41]. In this model clients randomly initiate sessions to download files from server nodes. The parameters and distributions for the web traffic model used in simulations are: number of sessions: 100, inter-session time: exponential with mean 5 sec, pages per session: 10, inter-page time: exponential with mean one sec, number of object per page: 10, inter-object time: exponential with mean 10 msec, and object size: Pareto II with mean 10 packets and shape 1.2. These parameters resulted in quite significant amount and variation of cross traffic in the test network. However, as we can see from Figure 7, TCP/DCA-C shows similar performance to the previous On-Off cross traffic case in most aspects.

E. Summary

In this chapter we proposed a new, measurement based, collaborative congestion control scheme called TCP/DCA-C for parallel, or quasi-parallel, TCP flows, which exchanges indicator signals about imminent congestion within the group in order to improve performances of all the flows in the group. In TCP/DCA-C, flows in a group can manage their data sending rates more accurately to achieve better performance by taking advantage of information that comes from other TCP flows, which experience congestion earlier, and by treating their congestion signals as indicators of imminent congestion in network.

Simulation results show that the TCP/DCA-C scheme for parallel TCP flows achieves better performance than unmodified parallel TCP flows in terms of packet loss ratio and delay and jitter without compromising aggregated goodput. The scheme also shows improved fairness among parallel TCP flows and comparable performances to TCP/ECN scheme which requires support from networks.

CHAPTER III

AGGREGATED AGGRESSIVENESS CONTROL ON GROUPS OF TCP FLOWS

A. Introduction

Parallel TCP has been widely used to work around the limitations of single-flow TCP over high bandwidth-delay product connections. GridFtp [42] and XFTP [43] are examples of the use of parallel flows at application level to achieve high throughput. PSockets [44] is a library that supports applications to use parallel connections. The use of parallel TCP flows has several benefits compared to a single TCP flow [45]. 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 [17]. As a result, the achievable aggregate throughput of parallel TCP flows is significantly bigger than the achievable throughput of a single TCP flow given the same packet loss probability and round-trip time. Unfortunately, this increase comes at the expense of the throughput experienced by other, single TCP flows, as sender nodes who open parallel TCP flows will unfairly consume more bandwidth when they compete with single TCP flows from other nodes for the same bottleneck links.

With the increased venues for bundling of TCP flows (e.g., overlay networks, TCP splicing [25], large servers with topological aggregation of service delivery, dedicated connections between supercomputers or campuses) 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 chapter we propose aggregate *strength* as a mean 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

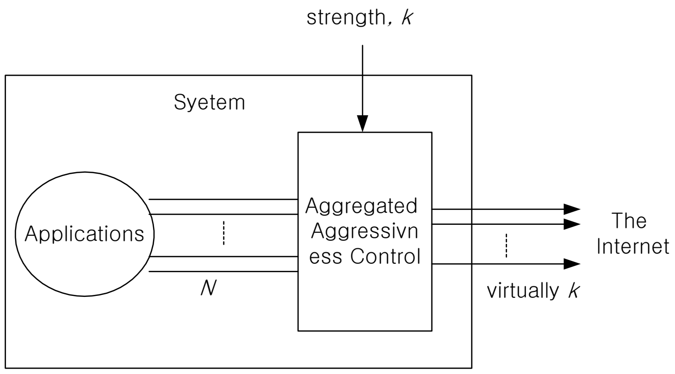


Fig. 8. Aggregated Aggressiveness Control

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 aggregated aggressiveness (or 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. In Figure 8 we show the conceptual diagram of aggregated aggressiveness control working in an Internet node. With the figure we illustrate that the system achieves the effectiveness of k TCP flows, and competing flows in the Internet experience the aggressiveness of k TCP flows even though applications in the system open N 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 [28], Fractional/Combined TCP flows [18] and COCOON [29] are some examples. In contrast to these schemes, MulTCP [21] 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 chapter is organized as follows: Section B presents the methodology we used to control parallel TCP flows' total strength. Section C describes our implementation in the Linux kernel. Section D presents experimental results that demonstrate how this implementation effectively controls the aggressiveness of parallel TCP flows. In Section E we analyze and evaluate the throughput ratios between parallel TCP flows and competing single TCP flows. Section F summarizes this chapter.

B. Aggregate Aggressiveness Control

Aggregate aggressiveness control on 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, 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.

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:

$$W(t+) = W(t) + 1. \tag{3.1}$$

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 k/N per non-duplicate ACK as shown in the following equation:

$$W_j(t+) = W_j(t) + \frac{k}{N}. \tag{3.2}$$

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 $1/W(t)$ as shown in the following equation:

$$W(t+) = W(t) + \frac{1}{W(t)}. \tag{3.3}$$

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 $\sum_{i=0}^{N-1}(W+i)/N$ 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:

$$\frac{\sum_{i=0}^{N-1}(\sum_{j=1}^{N} W_j + i)}{k * N}. \tag{3.4}$$

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. 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:

$$W(t+) = \frac{W(t)}{2}. \tag{3.5}$$

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 $(2k-1)/(2k)$ 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:

$$\sum_{j=1}^{N} W_j(t+) = \sum_{j=1}^{N} W_j(t) * (\frac{2k-1}{2k}). \tag{3.6}$$

For $k = N$, N parallel TCP-P flows becomes unmodified N parallel TCP, and the total decrease amount of N TCP-P flows become $(2N-1)/(2N)$ of the previous total window size. This is the same as that of unmodified parallel TCP flows' [17][1].

However, when a TCP-P flow detects a congestion event, its slow-start threshold (*ssthresh*) value is set to half of its congestion window size *before* the congestion window reduction regardless of the amount of the reduction. Furthermore, the TCP-P flow that detects a congestion event reduces only the congestion windows of other member TCP-P flows not their slow-start threshold values. The rationale behind this is that if TCP-P flows reset *ssthresh* values to the half of the new congestion window sizes, it may result in too aggressive behavior of TCP-P flows by putting them in multiplicative increase mode (i.e., $cwnd < ssthresh$) when next congestion events are detected. By doing this we drive the congestion window increase behavior of parallel TCP-P flows in linear increase mode (i.e., congestion avoidance mode) in most time of their operation.

3. Avoiding Unnecessary Decreases

Since packet drops in the network are typically bursty [46], 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

[1]When $N = k$, in TCP-P implementation, we disable all the mechanisms for TCP-P, so that N TCP-P flows with strength N becomes purely unmodified N parallel TCP flows.

trigger a congestion window reduction in other flows, which already have responded to the congestion event. This may result in a cascade of congestion window reductions. 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 in this situation.

To avoid unnecessary reduction of congestion windows, when the strength k of a group of TCP-P flows is less than their group size N, the TCP-P flows skip adjusting congestion windows 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[2] (i.e., minimum *srtt*.) In doing so, we assume that the length of packet drop bursts does not typically last longer than the minimum of the moving average of round-trip times.

C. 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 [47]. Also, Linux uses the packet as the unit of congestion window size, unlike BSD, which uses bytes, and adopts rate-halving [48] for congestion window reduction mechanism.

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

[2]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 $w = \frac{1}{8}$.

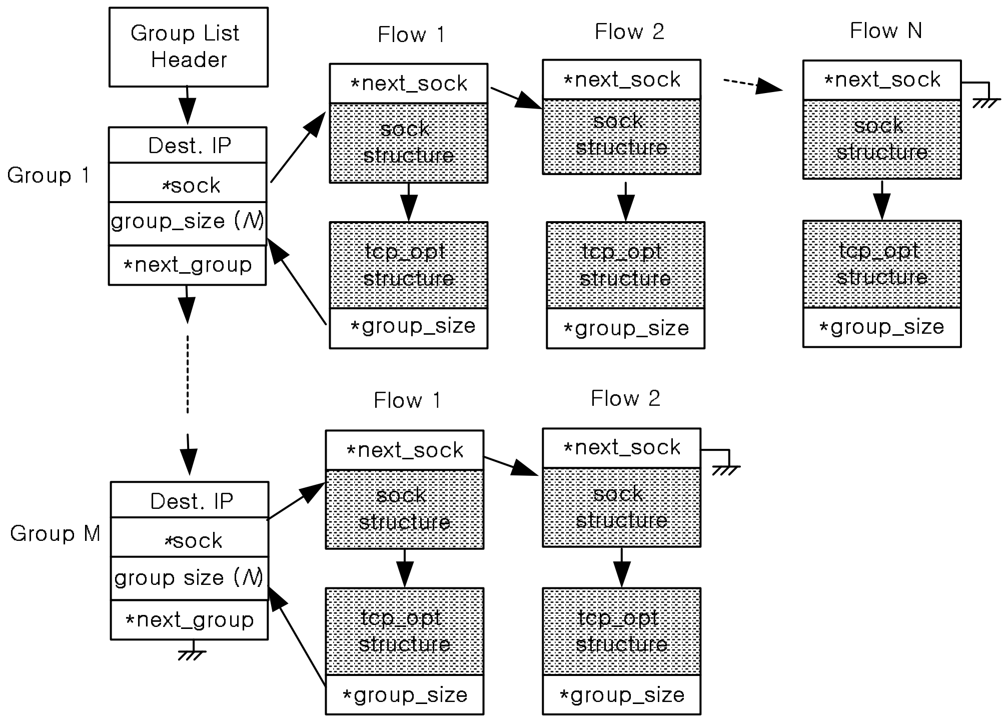


Fig. 9. Manage Groups and Flows

the same destination[3]. 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.

Figure 9 illustrates how the Linux kernel 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 these structures for TCP-P: a pointer that points `group_size` variable of its group is added to `tcp_opt` to get the number of flows of

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

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 has an unsigned integer variable `group_size` 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.

2. Implementing Increase

In slow-start mode, the congestion window of each TCP-P flow in a group is increased by k/N per non-duplicate ACK, as described in Equation (3.2). The amount of increase, 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 $\sum_{i=0}^{N-1}(\sum_{j=1}^{N} W_j + i)/(k * N)$ non-duplicate ACKs as Equation (3.4). To implement this for each TCP-P flow independently, we use the following equation:

$$\begin{aligned} & \frac{\sum_{i=0}^{N-1}(\sum_{j=1}^{N} W_j + i)}{k * N} \\ = & \frac{\sum_{i=0}^{N-1}\sum_{j=1}^{N} W_j + \sum_{i=0}^{N-1} i}{k * N} \\ = & \frac{N\sum_{j=1}^{N} W_j + \sum_{i=0}^{N-1} i}{k * N}. \end{aligned} \tag{3.7}$$

Therefore, each TCP-P flow should increase its congestion window size W_j by

1 after $(N\sum_{j=1}^{N} W_j + \sum_{i=0}^{N-1} i)/(k * N)$ non-duplicate ACKs. Alternatively, it can increase the congestion window by $(k * N)/(N\sum_{j=1}^{N} W_j + \sum_{i=0}^{N-1} i)$ 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 `group_size` as shown in Figure 9.

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

$$\begin{aligned} & \frac{N^2 * W_j + \sum_{i=0}^{N-1} i}{N} \\ = \; & NW_j + \frac{(N-1)N}{2N} \\ = \; & NW_j + \frac{N-1}{2}. \end{aligned} \tag{3.8}$$

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

3. Implementing Decrease

TCP-P controls the decrease amount of total congestion window sizes of parallel TCP-P flows according to Equation (3.6) 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. Another way is that 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 (3.6).

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

$$W_j(t+) = W_j(t) * (\frac{1}{2} + \frac{N(k-1)}{2(N-1)k}), \ \forall j \neq i. \tag{3.9}$$

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

$$\begin{aligned} & \sum_{j=1}^{N} W_j(\frac{2k-1}{2k}) \\ = \; & W_0 N(\frac{1}{2} + \frac{k-1}{2k}) \\ = \; & W_0(1+N-1)(\frac{1}{2} + \frac{k-1}{2k}) \\ = \; & \frac{1}{2}W_0 + W_0(N-1)(\frac{1}{2} + \frac{N(k-1)}{2(N-1)k}). \end{aligned} \tag{3.10}$$

However, note that when $k > N$, the latter method is not applicable because it may result in more than $(2k-1)/(2k)$ reduction of aggregate congestion window size.

D. Evaluation

For the evaluation of TCP-P, we use the topology shown in Figure 10. To emulate delays and packet losses in the Internet, we use NIST Net Emulator [49]. The NIST Net Emulator is implemented on a Linux machine and emulates the Internet by appropri-

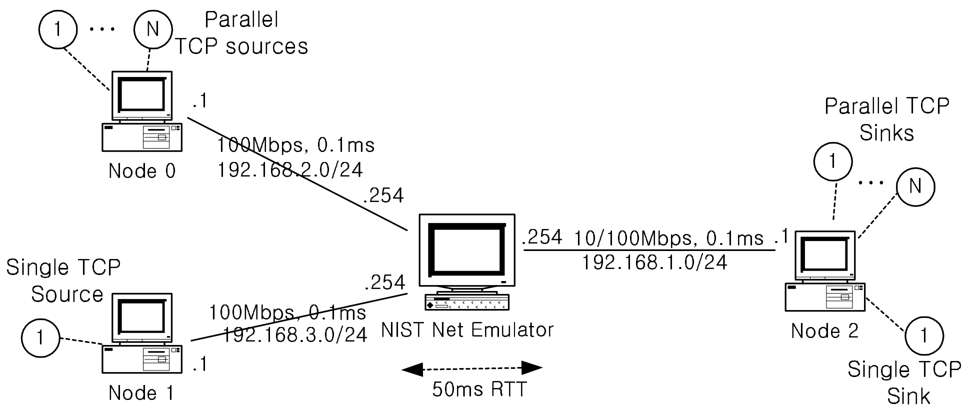
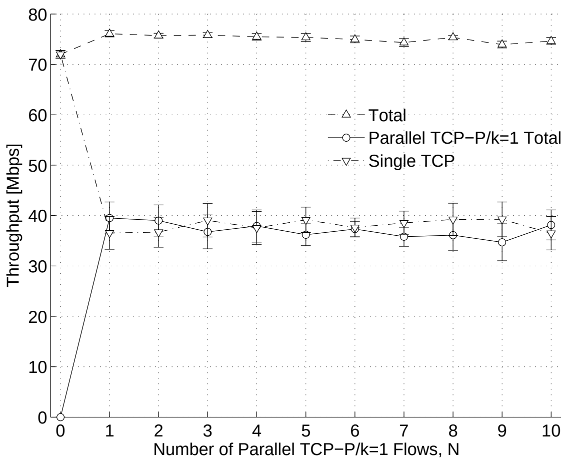


Fig. 10. Experiment Network

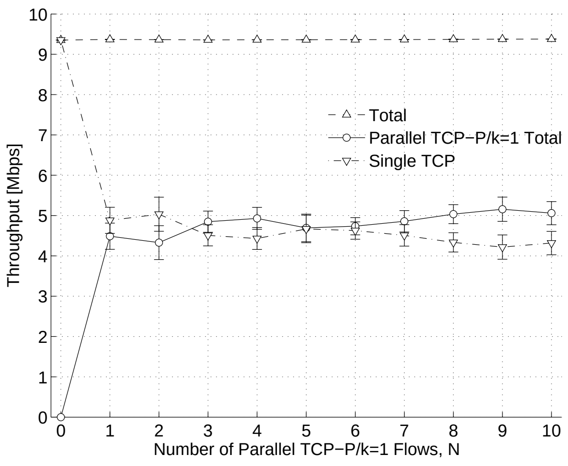
ately delaying and dropping packets. Because the network links in our experiments are fairly high-bandwidth (default 100Mbps) and the NIST Net delay parameters are large (50msec round-trip propagation delay,) we set the TCP parameters of the Linux end systems - such as `tcp_wmem` and `tcp_rmem` sizes - 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.

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 Node 0 we installed a modified Linux kernel that supports TCP-P. 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` [50], which supports parallel TCP flows and offers flexibility for measurement. In all experiments in this section, every experiment was done for 100 sec to get an average value and



(a) 100Mbps Link



(b) 10Mbps Link

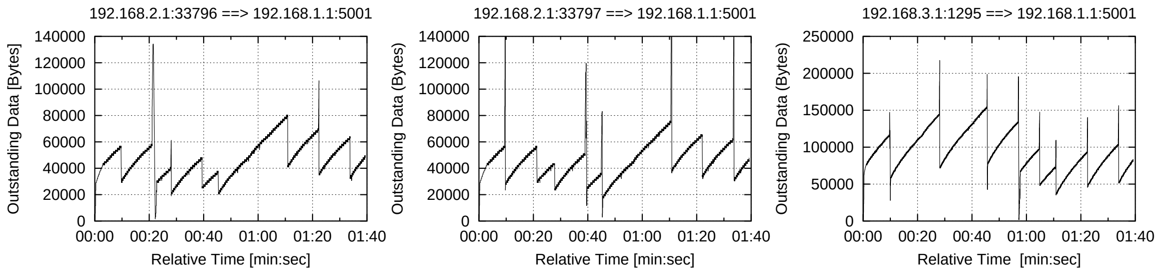
Fig. 11. Effect of N TCP-P Flows with $k = 1$ on a Single TCP Flow

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.

1. TCP-P Flow Groups with Strength $k = 1$

We first show the performance of TCP-P with $k = 1$ (We use TCP-P/k to call parallel TCP-P flows with strength k.) 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. Figure 11 (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 repeat the experiments with a reduced bottleneck link speed by limiting the link speed from the NIST Net emulator to the TCP sink Node 2 from 100Mbps to 10Mbps. Figure 11 (b)



(a) Flow 0 (b) Flow 1 (c) Flow 2

Fig. 12. Observed Outstanding Packets of Flows

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.

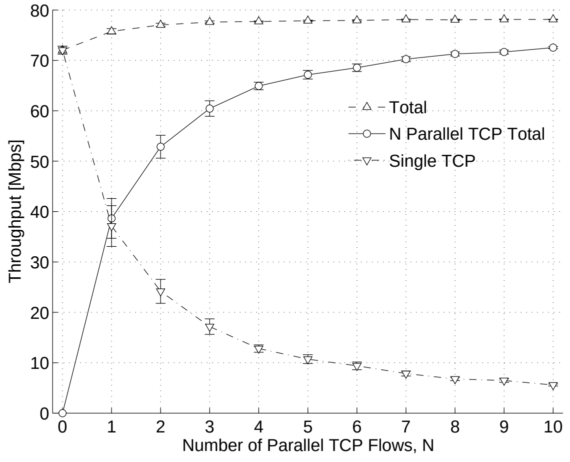
The experimental results in Figure 12 illustrate some of the details of the operation of TCP-P. These figures are generated by `tcptrace` [51] using `tcpdump` [52] data on the sender Node 0 and Node 1. For the simplicity of comparison of the time-dependent behaviors we open two parallel TCP-P flows with $k = 1$, Flow 0 and Flow 1, from Node 0 to Node 2, and one TCP flow, Flow 2, from Node 1 to Node 2. All other conditions are the same as 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 Retransmit/Fast Recovery [1] behaviors of TCP flows. Figure 12 (a) and Figure 12 (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 Figure 12 (c) always have a spike before a reduction.

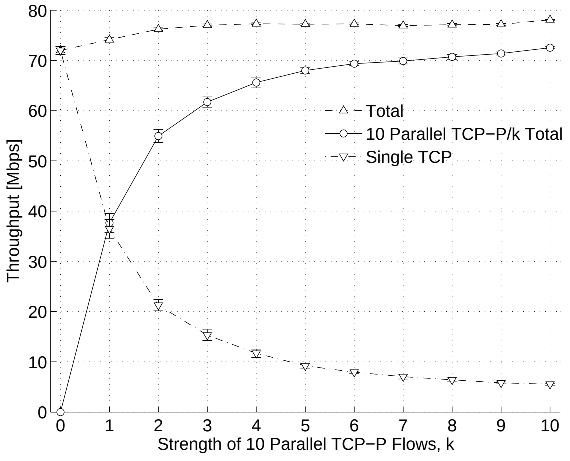
2. TCP-P Flow Groups with Strength $k > 1$

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 experimental results with unmodified parallel TCP flows in Figure 13(a). Node 0 opens N 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. Figure 13(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 the increasing numbers of the unmodified parallel TCP flows from Node 0. It illustrates the unfairness of parallel TCP flows mentioned in Section A of this chapter.

In comparison, Figure 13(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. Figure 13(b) shows average aggregate 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 TCP flow from Node 1 consumes all the bandwidth. By comparing (a) and (b) in Figure 13 we can see that the 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.



(a) N Parallel TCP



(b) 10 Parallel TCP-P/k

Fig. 13. Effect of Parallel N TCP and 10 TCP-P/k Flows on Single TCP in 100Mbps, 50msec Link

We also test parallel TCP-P with different strength k in 10Mbps bottleneck link by limiting the link speed from NIST Net Emulator to TCP sink Node 2 from 100Mbps to 10Mbps. Figure 14 shows the experimental results with (a) N parallel TCP flows and (b) 10 parallel TCP-P flows with different strength k in 10Mbps, 50msec RTT network. By comparing Figure 14(a) with 13(a) from experiments with 100Mbps link, we can see that the effect of parallel TCP flows on a single TCP flow remains similar in both cases. Further, by comparing Figure 14(a) with Figure 14(b) we can see that 10 parallel TCP-P flows with different strength k and k parallel TCP flows have similar effect on the competing single TCP in 10Mbps bottleneck link. These results also show that TCP-P can regulate the aggressiveness of parallel TCP-P/k flows comparable to that of k parallel TCP flows.

In the next section we analyze the steady-state throughput of N parallel TCP and TCP-P flows with strength k (TCP-P/k), and evaluate it with our experimental results.

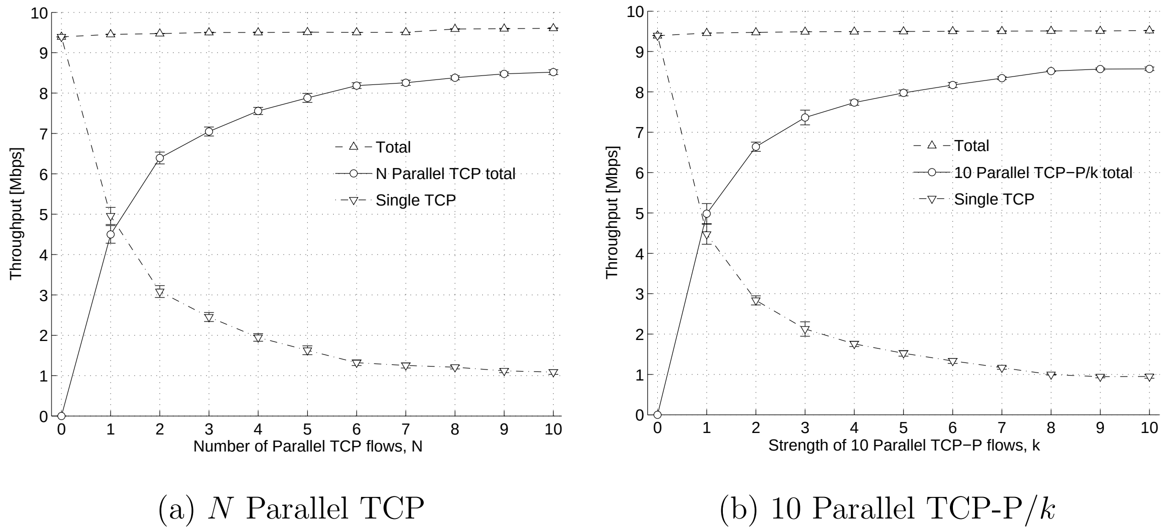


Fig. 14. Effect of Parallel N TCP and 10 TCP-P/k Flows on Single TCP in 10Mbps, 50msec Link

E. Throughput of Parallel TCP and TCP-P Flows

Previously the throughput of N parallel TCP flows was known as proportional to the number of parallel flows, i.e., N times larger than a single TCP flow [18]. Also, the throughput of MulTCP [21] had been modeled with similar results. However, we analyze steady-state throughput of parallel TCP and TCP-P/k flows and evaluate the results using throughput ratios between parallel TCP (and TCP-P/k) flows and the competing single TCP flow based on the experimental results in the test-bed network shown of Figure 10.

To analyze the aggregate throughput of N parallel TCP flows we closely follows the method used in [53]. We first assume that all the parallel TCP flow in a group has the same congestion window size W_0 at time $t = 0$, so that the sum of all the congestion windows of the parallel TCP flows is NW_0 as shown in Figure 15. We also assume that the packet loss probability is very small, so that only one packet loss happens per congestion epoch (we call a period that starts from window size with NW_0

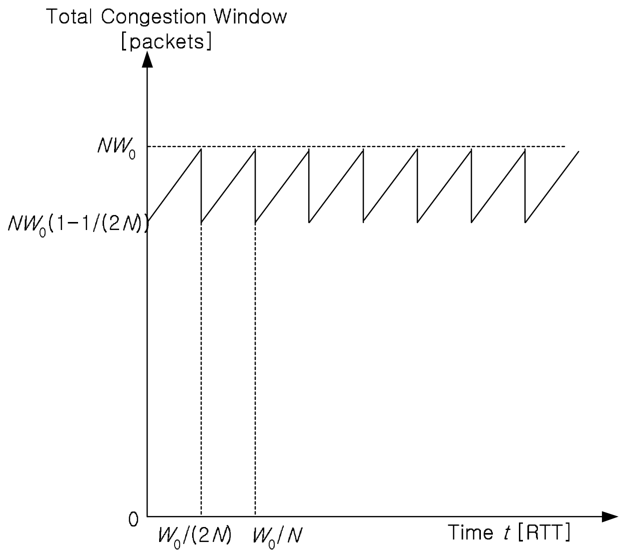


Fig. 15. Aggregate Congestion Window of N Parallel TCP Flows

and ends at the next NW_0 as a *congestion epoch.*), and delayed acknowledgement [1] is not used, i.e., one ACK per one data packet is generated by the receiver.

Because a TCP flow in unmodified parallel TCP flows behaves the same way as a normal TCP flow does, the total increase amount of congestion window of N parallel TCP flows is N per round-trip time instead of one of a single TCP flow if there is no packet loss. Also, the reduction of total congestion window is only $1/(2N)$ of total congestion window (NW_0) [17]. Hence, in parallel TCP flows the time needed to recover the congestion window (period of the congestion epoch) is $W_0/(2N)$ round-trip times because the amount of windows size needed to recover is only $W_0/2$ as shown in Figure 15.

Let S be the total amount of packets sent in each congestion epoch. Then, from Figure 15 we can see that $S = NW_0(1-\frac{1}{2N})\frac{W_0}{2N}+\frac{1}{2}\frac{W_0}{2}\frac{W_0}{2N}$. If we let p be the packet loss probability then, by the assumption, p equals to $1/S$. Hence, $1/p$ can be expressed using the following equation:

$$\begin{aligned}\frac{1}{p} &= S \\ &= \frac{W_0}{2N}(NW_0\frac{(2N-1)}{2N}+\frac{W_0}{4}) \\ &= \frac{W_0^2}{2N}(\frac{2N-1}{2}+\frac{1}{4}) \\ &= W_0^2(\frac{4N-1}{8N}). \end{aligned} \tag{3.11}$$

As a result, $W_0 = (\frac{1}{p}\frac{8N}{4N-1})^{1/2}$. Therefore, the average aggregate throughput $T(N)$ [packets/sec] of N parallel TCP flows can be modeled as:

$$\begin{aligned}T(N) &= \frac{\text{data sent per epoch}}{\text{time per epoch}} \\ &= \frac{W_0^2(\frac{4N-1}{8N})}{\frac{W_0}{2N}RTT} \\ &= \frac{W_0(4N-1)}{4RTT} \\ &= \frac{(\frac{8N}{4N-1})^{1/2}(4N-1)}{4RTT\sqrt{p}} \\ &= \frac{\sqrt{\frac{N(4N-1)}{2}}}{RTT\sqrt{p}} \text{ [packets per sec].}\end{aligned} \tag{3.12}$$

This equation can also be found by setting the AIMD parameter to be $a = N$ and $b = \frac{1}{2N}$ in the AIMD(a,b) equation of $T = \frac{\sqrt{2-b}\sqrt{a}}{\sqrt{2b}*RTT*\sqrt{p}}$ given in [54]. We can also compare the result when $N = 1$ with the following throughput equation of a single TCP flow given in [53].

$$T(1) = \frac{\sqrt{3/2}}{RTT\sqrt{p}} \text{ [packets per sec].} \tag{3.13}$$

From Equation (3.12) we can see that with the same round-trip time and packet loss probability, N parallel TCP flows can achieve $\sqrt{N(4N-1)/3}$ more throughput

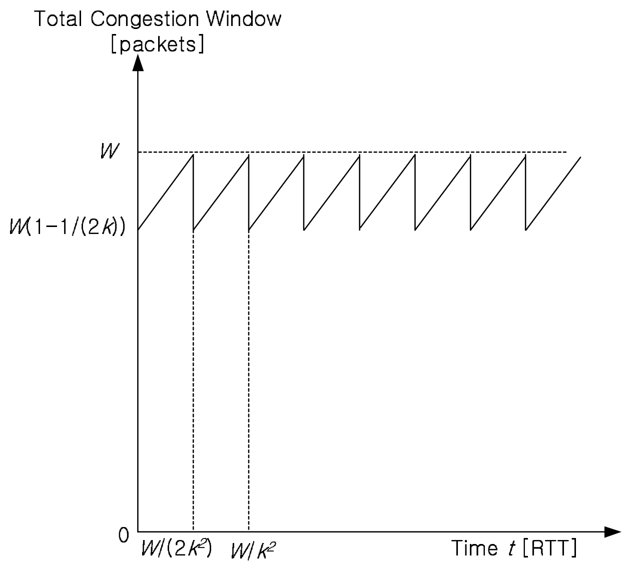


Fig. 16. Aggregate Congestion Window of Parallel TCP-P/k Flows

than a single TCP flow:

$$\frac{T(N)}{T(1)} = \sqrt{N(4N-1)/3}. \tag{3.14}$$

In contrast to the unmodified TCP flows, parallel TCP-P flows with strength $k = 1$ controls the total congestion window increase and decrease of parallel TCP-P flows to match that of a single TCP flow, so that it results in effective AIMD parameters for parallel TCP-P flows with $a = 1$ and $b = 1/2$, which are equal to those of a standard TCP flow.

Next, based on the behavior of parallel TCP-P flows described by Equation (3.4) and (3.6), we analyze steady-state throughput of parallel TCP-P/k flows. Figure 16 illustrates the changes of the aggregate congestion window size of parallel TCP-P/k flows. In the figure, W is the total congestion windows size of parallel TCP-P flows when packet loss happens. As the figure shows, the time needed to recover the congestion window in TCP-k is reduced to $\frac{W}{2k^2}$ round-trip times (RTTs) from the $\frac{W}{2}$ RTTs in standard TCP. This reduction stems from the smaller congestion window size needed to recover ($\frac{W}{2k}$), and the higher rate at which the window size increases (increases speed is k packets per RTT in congestion avoidance mode.) We use Figure 16 to model the steady-state throughput of TCP-P/k, and again let S be the total amount of packets sent in each congestion epoch by TCP-P/k flows. From Figure 16, we can see that $S = W(1-\frac{1}{2k})\frac{W}{2k^2} + \frac{1}{2}\frac{W}{2k}\frac{W}{2k^2}$. If we let p be the packet loss probability, then p equals to $\frac{1}{S}$. Hence, $\frac{1}{p}$ can be expressed using the following equation.

$$\frac{1}{p} = \frac{W^2}{2k^2}\frac{4k-1}{4k} = W^2\frac{4k-1}{8k^3}. \tag{3.15}$$

As a result, $W = (\frac{1}{p}\frac{8k^3}{4k-1})^{1/2}$. Thus, the average throughput $T(k)$ [packets/sec] of a TCP-k flow can be modeled as:

$$\begin{aligned} T(k) &= \frac{\text{data sent per congestion epoch}}{\text{time per congestion epoch}} \\ &= \frac{W^2\frac{4k-1}{8k^3}}{\frac{W}{2k^2}RTT} = \frac{W(4k-1)}{4kRTT} \\ &= \frac{(\frac{8k^3}{4k-1})^{1/2}(4k-1)}{4kRTT\sqrt{p}} \\ &= \frac{\sqrt{\frac{k(4k-1)}{2}}}{RTT\sqrt{p}} \text{ [packets per sec]}. \end{aligned} \tag{3.16}$$

By comparing Equation (3.16) with Equation (3.13) we can see that, given the same round-trip time and packet loss probability, TCP-P/k flows achieve $\sqrt{k(4k-1)/3}$ times the throughput of a single TCP flow:

$$\frac{T(k)}{T(1)} = \sqrt{k(4k-1)/3}. \tag{3.17}$$

This is consistent with the results for parallel TCP flows in Equation (3.14) if we replace k with N.

We evaluate the ratio models by comparing aggregate throughput of N parallel TCP and parallel TCP-P/k flows with the throughput of the competing single TCP

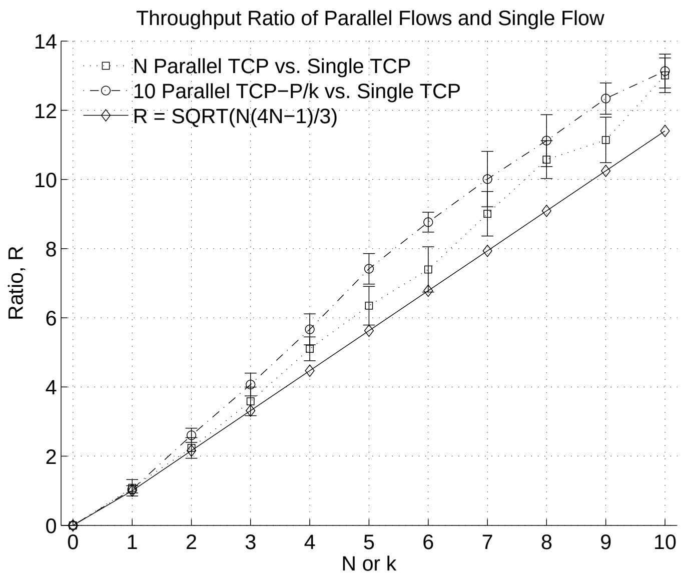


Fig. 17. Throughput Ratios of Parallel TCP-NewReno and TCP-P against Single TCP-NewReno

flow. We first use the data from the previous experiments in Figure 13. Figure 17 shows the ratios of achieved aggregated throughput by parallel TCP and TCP-P flows against a single TCP flow in Figure 13(a) and Figure 13(b). The dashed line in the figure presents the ratio between aggregate throughput of 10 TCP-P flows with different strength k and the throughput of a single TCP flow. The dotted line in the figure shows the ratio between the aggregate throughput of N unmodified parallel TCP flows and the throughput of a single TCP flow. Both graphs show the ratios closely follow Equation (3.14) shown as the solid line in the figure.

Next, we configure the test bed shown in Figure 10 to have random packet drops by NIST Net Emulator. The achieved throughput by TCP flows is significantly reduced when NIST Net invoked random packet drops to emulate non-congestion related packet losses such as transmission channel errors. When we configure NIST Net to drop 0.01% of packets, the achievable bandwidth for a single TCP flow drops to about 20Mbps. This value is close to the theoretical average throughput of a TCP

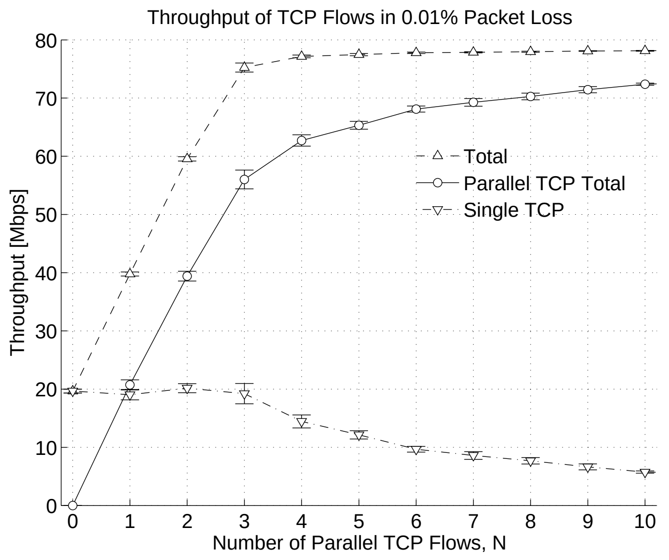


Fig. 18. Throughput of TCP-NewReno Flows in 0.01% Packet Loss

flow [53] if we consider delayed-ACK of Linux TCP and a maximum segment size of 1500 bytes.

Again, Node 0 opens a varying number of parallel TCP flows to Node 2 while Node 1 opens only a single TCP flow to Node 2. Figure 18 shows the experiment results for unmodified parallel TCP flows against a single TCP flow. From the experimental results in the figure show the average throughput of unmodified parallel TCP flows when it increases the number of parallel TCP flows. We can see from the results that by increasing the number of unmodified parallel TCP flows, Node 0 can achieve high throughput even if there is high rates of non-congestion related packet loss. However, note that Node 1 again *steals* bandwidth from the competing single TCP flow from Node 1 when Node 1 opens more than four parallel TCP flows.

Figure 19 shows the ratio of the total achieved throughput of unmodified parallel TCP flows from Node 0 to the throughput of the single TCP flow from Node 1 in the previous experiments. The figure shows that the throughput ratio of the experiment

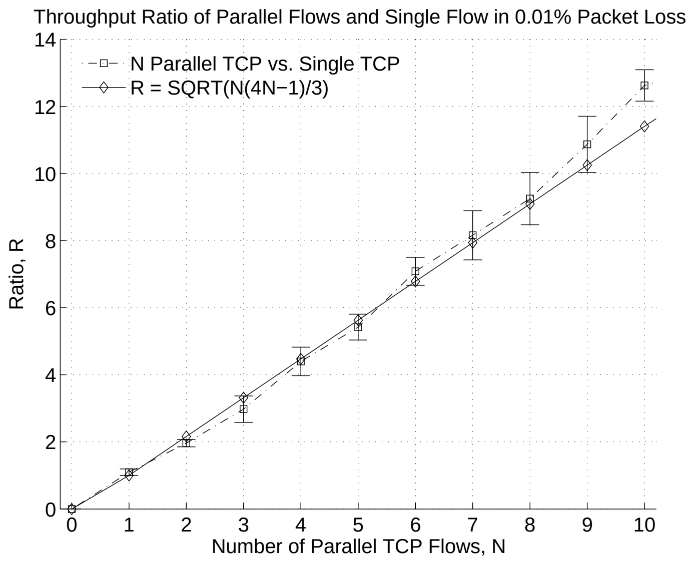


Fig. 19. Throughput Ratios of Parallel TCP (TCP-P) against Single TCP in 0.01% Packet Loss

results is indeed close to the ratio model given in Equation (3.14).

F. Summary

In this chapter, we have proposed TCP-P for aggregated aggressiveness control on parallel TCP flows. The 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 have showed that by employing TCP-P we can control the total aggressiveness or unfairness of parallel TCP flows against single TCP flows from other nodes in an easily parameterizable and controllable way without requiring application modification.

CHAPTER IV

ADAPTIVE TCP FOR EFFECTIVE AND FAIR UTILIZATION OF HIGH BANDWIDTH-DELAY PRODUCT NETWORKS

A. Introduction

TCP [55] has proven remarkably effective in providing stable end-to-end reliable services for large-scale distributed applications, despite the fact that the Internet has increased by several orders of magnitude. The evolution of transmission technology and the increasing demand for very-high bandwidth services, however, pose a number of new challenges to TCP. Over emerging high bandwidth-delay product networks, TCP's flow control and error control mechanisms have shown to be limiting factors for the effective utilization of such networks: Traditional TCP increases its congestion window (and sending rates) too slowly with non-duplicate acknowledgement packets (ACKs) and decreases the congestion window size too much with congestion events such as three duplicate ACKs. Further, at this level, even very small non-congestion related packet loss can further reduce utilization of the links. This becomes problematic especially over high-speed long-distance paths because of the long time required to recover congestion windows and, hence, the sending rate.

In order to fully utilize the available link bandwidth in high bandwidth-delay product networks, it is important to adapt the system parameters of the TCP sender and receiver hosts, such as maximum TCP sender and receiver window sizes, to reflect link characteristics [56]. Because of the heterogeneity and variety in connection characteristics, such as bandwidth or delay in today's networks, adapting system parameters for TCP itself is not easy. The experiments in projects such as web100 [57] clearly show how poor configuration of TCP parameters leads to a low utilization of available bandwidth even if the systems are connected over high-bandwidth links. Furthermore, even if system parameters are appropriately configured, TCP's congestion control mechanism itself needs to be more effective: TCP should increase its sending rate much faster than current traditional TCP when it detects available (or unclaimed) bandwidth in its path.

Recently, several proposals to improve the performance of TCP over high bandwidth-delay product networks have been introduced. These proposals include HS-TCP [12], STCP [13], H-TCP [14], BIC-TCP [15], FAST-TCP [11], and LTCP [16]. An experimental performance comparison of some of these new TCP proposals was reported in 2004 and 2005 PFLDnet workshops [58, 59]. Given the design objective of these protocols, it is not surprising that they tend to not inter-operate well with traditional TCP flows. For example, all protocols listed above - except for FAST-TCP - are seriously unfair against standard TCP, such as TCP-SACK [9], as we will show in the following. FAST-TCP, on the other hand, becomes weaker than loss-based standard TCP when the two compete for the same network resources. This is due to the congestion avoidance mechanism in FAST-TCP reduces the sending rate to avoid packet loss when the measured delay increases, and it can not distinguish whether the delay increase is self-introduced or caused by other flows.

In order to support "cohabitation" of traditional applications and emerging applications with very high bandwidth demands, protocols must be developed to support both effective *and* fair use of network resources. For the effective and fair utilization of network resources, a number of proposed schemes, such as eXplicit Congestion Protocol (XCP) [3], require participation of the underlying network infrastructure, such as switches or routers. History has shown that, in practice, it is unlikely for Internet service providers to adopt these schemes given the additional operational overheads they cause on routers/switches. In the following, we focus on end-host-based solutions. In other words, we look for congestion control mechanisms purely based on information that can be gathered at the sender side.

In this chapter, we first evaluate the performance of a number of well-known high-speed TCP proposals using a test-bed setting. These experiments will illustrate that all evaluated schemes have problems - one way of the other - with fairness to standard TCP. We believe that the fairness problem of the high-speed TCP proposals stems from their ineffectiveness in detecting competing TCP flows or differentiating self-induced congestion from the congestion caused by other flows. Based on these results we propose a new, adaptive, congestion control scheme, which relies on competition detection to identify whether impending congestion is self-inflicted or non-self inflicted. Based on this scheme we implement Adaptive TCP (A-TCP), an extension of TCP/k (or MulTCP [21]) that uses a competition detector to adaptively tune its TCP friendliness. We show that A-TCP can achieve high performance in high bandwidth-delay product network without requiring any special support from network infrastructure or from receivers. Also, we show that A-TCP exhibits good TCP-friendliness toward standard TCP when both compete for the same network resources.

The remainder of this chapter is organized as follows: Section B surveys related work on high-speed TCP proposals and evaluates their effectiveness and fairness. In Section C, we discuss the benefits and problems emerging from the use of parallel TCP flows in high-bandwidth settings. We also discuss how some of these benefits are harnessed in single-flow emulation schemes of parallel TCP, such as MulTCP [21] and our TCP/k. Section D describes basic concept for our competition detection mechanism. In Section E, we propose Adaptive TCP and describe how it combines TCP/k with the competition detection mechanisms to automatically adapt its aggressiveness. Section F presents experimental results that demonstrate the effectiveness

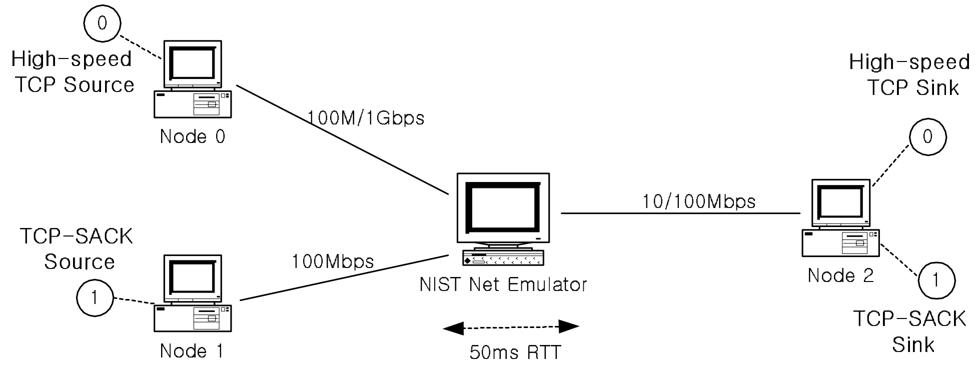


Fig. 20. Test-Bed for Experiments

of the proposed TCP in high bandwidth-delay product networks. In Section G we evaluate the fairness property of Adaptive TCP through experiments. In Section H we show ns2 simulation results of A-TCP and other high-speed TCP proposals. Section I summarizes this chapter.

B. Related Work and Fairness

In this section, we survey and evaluate the performance of several popular high-speed TCP protocols over high bandwidth-delay product networks in terms of effectiveness and fairness (or TCP-friendliness[1]). By *effectiveness* we mean how well a high-speed TCP flow utilizes high bandwidth-delay product networks. By *fairness* we mean how fairly (or unfairly) a high-speed TCP flow shares network resources - such as bandwidth - with standard TCP flows, e.g., TCP-SACK, from other nodes. Our experiments will show how effectively existing high-speed TCP proposals utilize available bandwidth and how fairly (or unfairly) they share bandwidth with standard TCP.

For this series of experiments, we use the test-bed shown in Figure 20. In this test-bed, we measure the bandwidth achieved by a high-speed TCP flow and compare

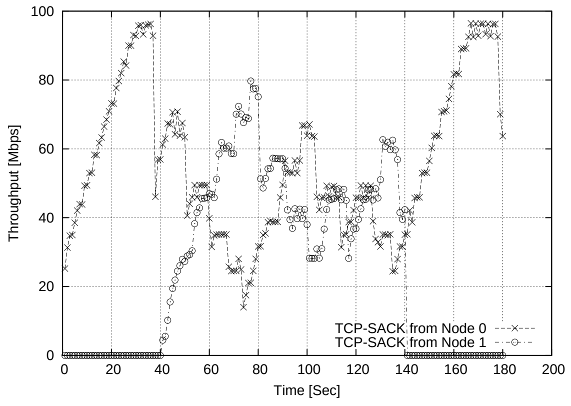
[1]We use the terms fairness and TCP-friendliness interchangeably.

it with that of a competing standard TCP flow. We use the NIST Net Emulator [49] to emulate a portion of an inter-network. The NIST Net Emulator emulates a network cloud by appropriately delaying and dropping packets if it is configured to do so. In our test-bed, all the end-host nodes and the NIST Net Emulator are running on Linux PCs. The PCs we use for experiments are Pentium 4 machines with 10/100Mbps or 1Gbps Ethernet network interface cards. Each 10/100Mbps Ethernet card has an output queue of length 100 packets by default. We set the TCP system parameters of the Linux end systems - such as `tcp_wmem` and `tcp_rmem` - according to the recommendations[2] in [57], rather than using system defaults.

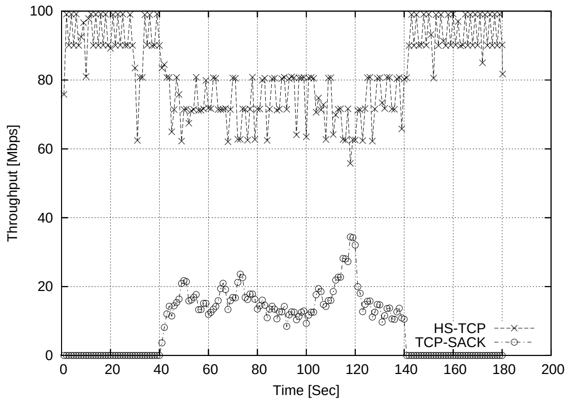
Sender Node 0 is used to evaluate the various high-speed TCP protocols. In this node, we installed Linux kernel 2.4, which we modified with patches from the authors' web-site to accommodate the various high-speed TCP protocols. We use these modified kernels without changing any parameters from their default settings. The other sender, Node 1, always runs on Linux kernel 2.4.20-8 to generate standard TCP cross traffic. In this kernel, the TCP behavior is by default based on TCP-SACK. The NIST Net node emulates an inter-network cloud. The NIST Net Emulator is running on Linux kernel 2.4.20-8. Receiver Node 2 also runs a standard TCP stack in Linux kernel 2.4.20-8. For TCP traffic generation and throughput measurement we use `iperf` [50], which offers great flexibility for traffic generation and measurements.

[2]System parameters of TCP sender and receiver nodes are configured as below unless told otherwise.

- `/proc/sys/net/core/wmem_max` = 8388608.
- `/proc/sys/net/core/rmem_max` = 8388608.
- `/proc/sys/net/ipv4/tcp_wmem` = "4096 65536 4194304".
- `/proc/sys/net/ipv4/tcp_rmem` = "4096 65536 4194304".
- TCP receivers' socket buffer sizes are set to 16MByte.



(a)

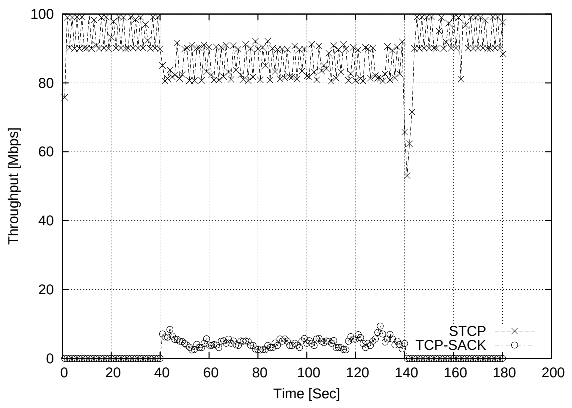


(b)

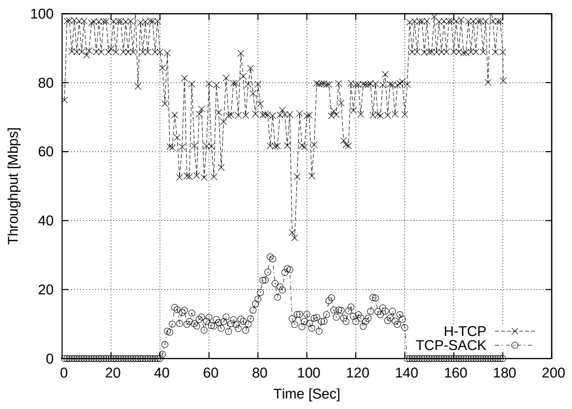
Fig. 21. TCP-SACK vs. (a) TCP-SACK (b) HS-TCP

The experiment runs are designed to evaluate effectiveness of high-speed TCP protocols first, and TCP friendliness in a second part: For the first 40 seconds of each run we let a high-speed TCP flow from Node 0 to Node 2 use the 100Mbps, 50msec RTT link without competition. High resource utilization during this phase indicates good effectiveness of the protocols. After 40 seconds Node 1 opens a TCP-SACK flow to Receiver Node 2. The TCP-SACK flow starts to compete for the resources of the bottleneck link with the high-speed TCP flow from Node 0 for the next 100 seconds. The bandwidth results from this phase of the experiment run give indicators about the fairness of the high-speed protocols. Throughputs of all the flows shown in this section are measured using Iperf at the senders every one second.

As a **base line**, Figure 21(a) shows experimental results with two standard TCP flows (TCP-SACK). The experimental results show that the two standard TCP flows share the bottleneck link fairly; each of the two flows achieves similar throughput during the 100 second period when both are active. This figure also shows how TCP-SACK increases its sending rate slowly and decreases it drastically. This behavior of standard TCP results in overall low utilization of the available bandwidth, especially



(a)



(b)

Fig. 22. TCP-SACK vs. (a) STCP (a) H-TCP

when the flow is alone in the test-bed network. In the following we repeat the same experiment using a number of high-speed TCP proposals.

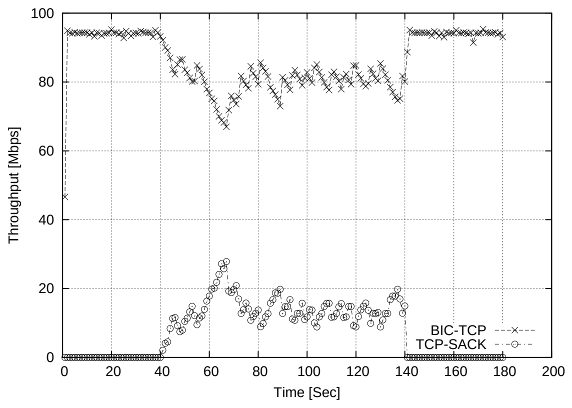
HS-TCP (HighSpeed TCP) [12] is one of the early TCP proposals for high bandwidth-delay product networks. HS-TCP increases its congestion window faster and reduces it less than standard TCP by changing the values for its AIMD parameters (which decide congestion window increase amount per non-duplicate ACK and the reduction proportion of the congestion window per congestion event) if the congestion window size is bigger than some threshold. Otherwise, the AIMD parameter values are those of standard TCP's. Figure 21(b) shows the experiment results when Node 0 opens a HS-TCP flow to Node 2. For this experiment we installed modified Linux kernel 2.4.27 obtained from the Web100 project in Node 0. Although the experiment results show that a single HS-TCP flow highly utilizes the bandwidth while it is alone in the test-bed, the figure clearly shows that HS-TCP is seriously unfair when it competes against a standard TCP flow: When Node 1 opens a single standard TCP-SACK flow at 40 sec, it only achieves about 19Mbps during the next 100 seconds. At the same time the HS-TCP flow from Node 0 consumes about 75Mbps.

STCP (Scalable TCP) [13] adopts a multiplicative-increase scheme for the congestion window size (instead of TCP's additive increase) to quickly grow the congestion window: If the congestion window size of a STCP flow is bigger than a threshold, the flow decides to increase its congestion window multiplicatively and reduces its congestion window less than a standard TCP flow does. Figure 22(a) shows how STCP at first effectively utilizes the available bandwidth when it is alone in the test-bed. When it competes against a standard TCP-SACK flow, however, the results show that STCP is unfair. The achieved throughput for the competing TCP-SACK flow is in fact less than when it competes against the HS-TCP flow.

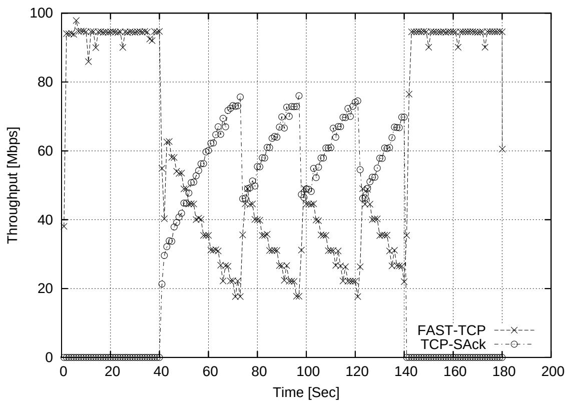
H-TCP (High-speed TCP) [14] was proposed to utilize high bandwidth-delay product networks effectively while retaining TCP-friendliness for backward compatibility. To do this, it has two modes of operations: If the durations of congestion events[3] are less than a given threshold, H-TCP retains a traditional AIMD scheme; otherwise, it sets the additive-increase parameter to a value bigger than that of standard TCP. It monitors the durations of congestion epochs to decide which mode to use. H-TCP also monitors the throughput before congestion events to decide the reduction proportion of the congestion window size. Figure 22(b) shows the experiment results with H-TCP where Node 0 opens an H-TCP flow to Node 2. The results show that a single H-TCP flow effectively utilizes the available bandwidth whenever it is alone in the test-bed. However, we can also see that this effectiveness comes at the cost of unfairness toward standard TCP-SACK flows.

BIC-TCP (Binary Increase Congestion control TCP) [15] uses a method called *binary search increase* to quickly utilize available bandwidth while reducing packet loss. BIC-TCP identifies a target window size and increases the congestion window

[3]A congestion epoch is a period that starts from a congestion event and finishes at the next one.



(a)



(b)

Fig. 23. TCP-SACK vs. (a) BIC-TCP (b) FAST-TCP

size using a binary search scheme. As a result, BIC-TCP in effect uses a logarithmic function to increase its congestion widow size. We installed BIC-TCP in Node 0 with the kernel 2.4.25 version patches from the authors' web-site. The experimental results in Figure 23(a) show that BIC-TCP utilizes bandwidth effectively when it is alone on the bottleneck. However, when a standard TCP-SACK flow joins, the BIC-TCP flow prevents it from achieving its fair throughput.

FAST-TCP (Fast Active queue management Scalable TCP) [11] is a variant of TCP-Vegas [10] for high bandwidth-delay product networks. It uses delay-variation information gathered at the sender side to decide whether it needs to change its congestion window size. The experimental results in Figure 23(b) show that FAST-TCP effectively utilizes the 100Mbps, 50msec RTT network with a single FAST-TCP flow when there is no cross traffic. However, when a single TCP-SACK flow from Node 1 joins the network, the FAST-TCP flow backs off and does not effectively compete against the loss-based TCP-SACK flow. Therefore, it achieves *less* throughput than the competing TCP-SACK flow during the overlapping 100 second period.

This result is somewhat expected because FAST-TCP uses a delay-based congestion avoidance mechanism similar to that used in TCP-Vegas, which slows down its sending rate to avoid congestion. In contrast, loss-based schemes such as TCP-SACK continuously increase their congestion windows (thus sending rates) until packet loss occurs. This behavioral difference of loss-based and delay-based schemes eventually results in lower throughput for delay-based congestion avoidance mechanisms, such as TCP-Vegas and FAST-TCP, whenever they compete against loss-based (and standard) TCP such as TCP-SACK. The lack of ability in these delay-based schemes to differentiate the delay increase caused by self-induced congestion from that by other flows also contributes to these results.

TCP-Africa [60] operates in two modes and uses congestion detection to change from *fast mode* to *slow mode*. The congestion detector in TCP-Africa is based on delay variations: TCP-Africa increases its congestion window linearly like for standard TCP whenever the estimated queue build-up is bigger than a threshold value. Otherwise, TCP-Africa behaves like as HS-TCP. We do not include experimental results with TCP-Africa because we could not find any kernel implementation of TCP-Africa. Instead, we implemented TCP-Africa in ns2 [37] simulator according to the algorithms described in [60]. The ns2 simulation results of TCP-Africa as well as other high-speed TCP proposals are shown in Section H.

In summary, the experimental results of this section show that it is hard to design a TCP congestion control scheme that effectively utilizes high bandwidth-delay product networks while remaining TCP-friendly to existing standard TCP. We argue in the following that the use of congestion detectors based on delay variations is not sufficient to ensure fairness, for two reasons. First, congestion detectors typically are unable to differentiate *self-induced congestion* (i.e., when no other flow competes for bandwidth, but data rates are simply too high) from *competition*. In such cases the detector is too conservative. Second, competing flows may not always be able to build up sufficient throughput to generate congestion, in which case the detector becomes ineffective.

In Section D we propose the use of a *competition* detector to control the mode of operation of the proposed TCP protocol. By "competition detection" we mean the detection of the existence of flows that can effectively compete for the same network resources.

C. Parallel TCP and TCP/k

The TCP protocol that we propose in this chapter operates in two modes (highly aggressive and non-aggressive) and uses a competition detector to change between them. In this section we first present a scheme to gracefully and controllably tune the aggressiveness of TCP. We will then propose the competition detector and how to combine the detector with the TCP aggressiveness control in the following section. We will describe an extension to traditional TCP that allows for a dynamically controllable aggressiveness through single parameter k, which we call "strength". The strength parameter k is a positive value that describes the aggressiveness (in number of flows) of the single flow as perceived by other TCP flows. We call this TCP scheme with controllable aggressiveness TCP/k.

TCP/k controls increase and decrease behavior of standard TCP to achieve the aggressiveness of k *parallel* TCP flows. By definition, parallel TCP flows are multiple, active TCP connections concurrently opened by a node to the same destination. Parallel TCP is a widely used scheme to work around the limitations of TCP over high bandwidth-delay product networks [61]. If an end-host opens N parallel TCP flows to the same destination, their total congestion window increase speed is N times faster than a single TCP flow and reduce total congestion window size less than half

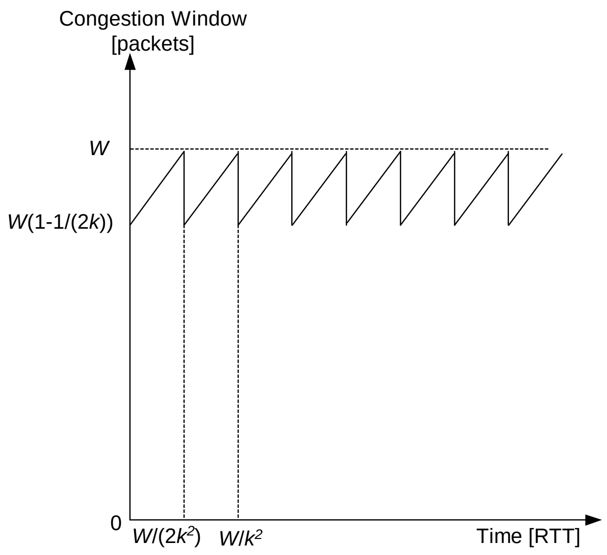


Fig. 24. Changes of the Congestion Window of TCP/k

when packet loss happens [17]. As a result, the achievable aggregate throughput of parallel TCP flows is significantly higher than that of a single TCP flow given a packet loss probability and round-trip time.

The goal of TCP/k is to emulate the aggressiveness of k parallel TCP flows with a single TCP. With a given strength k, a single TCP/k behaves as if it were composed of k parallel TCP flows. TCP/k realizes this through appropriate manipulation of the congestion window increase and decrease behavior: In slow-start mode, TCP/k increases its congestion window size, *cwnd*, by k per non-duplicate ACK as shown below.

$$cwnd \leftarrow cwnd + k. \tag{4.1}$$

In congestion avoidance mode, TCP/k increases its congestion window by the following amount per non-duplicate ACK:

$$cwnd \leftarrow cwnd + \frac{k}{cwnd}. \tag{4.2}$$

When a congestion event such as three duplicate ACKs is detected, TCP/k adjusts its congestion window size and slow-start threshold value (*ssthresh*) as follows:

$$\begin{aligned} ssthresh &\leftarrow \frac{cwnd}{2} \\ cwnd &\leftarrow cwnd\frac{2k-1}{2k} \end{aligned}. \tag{4.3}$$

As a result, the increase and decrease behavior of a single TCP/k flow closely reflects that of k parallel TCP flows: The congestion window increase speed is k times faster than that of standard TCP, and the reduction proportion of the congestion window with packet loss is $1/2k$ instead of $1/2$ in standard TCP. Figure 24 illustrates the changes of the congestion window size of a TCP/k flow. As the figure shows, the time needed to recover the congestion window in TCP/k is $\frac{W}{2k^2}$ round-trip times (RTTs) instead of $\frac{W}{2}$ RTTs in standard TCP. This reduction stems from the smaller congestion window size needed to recover ($\frac{W}{2k}$) and the higher rate at which the window size increases (increase speed is k packets per RTT in congestion avoidance mode.)

Note that the mechanisms used for TCP/k are similar to those used in MulTCP [21] and the strength parameter in TCP/k has the same meaning as the *multiplicity* parameter of MulTCP. However, differently from MulTCP, TCP/k does not limit the aggressiveness in slow-start mode because doing so may result in different behavior from k parallel TCP flows. Further, TCP/k sets *ssthresh before* adjusting *cwnd* as shown in Equation (4.3). This is because TCP/k is a special case of TCP-P/k, i.e., TCP/k is a single TCP-P/k flow.

To show that TCP/k indeed has the same aggressiveness as k parallel TCP flows, we implemented TCP/k in Linux kernel 2.4.20. The experimental results in Figure 25 illustrate how k parallel TCP flows resemble a single TCP/k flow. For the experiment we use the test-bed shown in Figure 20 with all 100Mbps links and 50msec round-trip propagation delay in the NIST Net Emulator. Every experiment was done for 100 sec to get average values and repeated 10 times with 5 sec waiting time after each

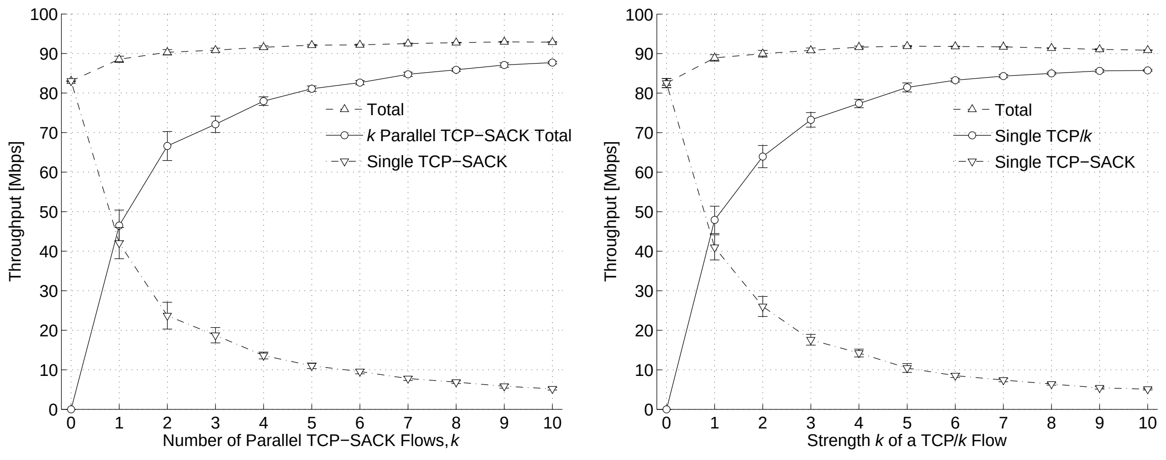
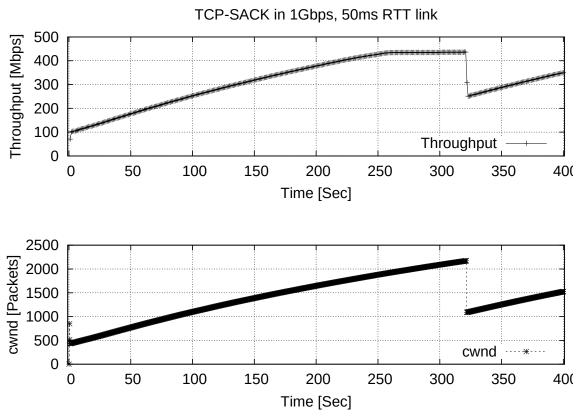


Fig. 25. TCP-SACK vs. (a) k Parallel TCP-SACK (b) TCP/k with Different k

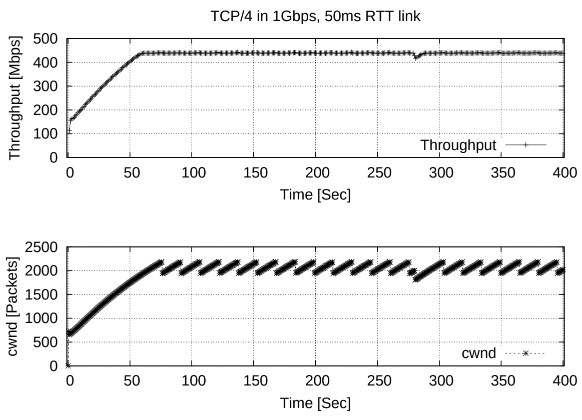
experiment. Error bars at the figures represent the 95% confidence interval of the mean value.

Figure 25(a) shows the aggregated throughput of k parallel TCP-SACK flows from Node 0 and the throughput of the competing single TCP-SACK flow from Node 1. In this experiment, Node 0 opens k parallel TCP-SACK flows to the receiver Node 2 for 100 seconds, while Node 1 opens a single TCP-SACK flow to Node 2 for the same period. In this figure, $k = 0$ denotes the case of no parallel TCP flows from Node 0, so that only the single TCP-SACK flow from Node 1 consumes all the network resources. From the figure we can see that Node 0 achieves higher aggregate throughput by increasing the number of parallel TCP flows. However, the throughput of the single TCP-SACK flow from Node 1 shrinks as the number of parallel TCP flows from Node 0 increases. The figure illustrates both the effectiveness and the unfairness of parallel TCP flows caused by the aggressiveness of the parallel TCP flows against the single TCP flow.

Figure 25(b) shows similar results with TCP/k in the same test-bed. Instead of increasing the number of parallel TCP flows, we increase the value of the strength pa-



(a) TCP-SACK



(b) TCP/4

Fig. 26. Throughput and Congestion Window over 1Gbps, 50ms RTT Link

rameter k. By comparing Figure 25(a) and Figure 25(b) we see that the aggressiveness of the TCP/k flow is accurately controlled by changing strength k. Moreover, TCP/k utilizes bandwidth as aggressively as k parallel TCP flows. The effectiveness of both k parallel TCP flows and the single TCP/k flow in utilizing the bandwidth comes from their aggressive behavior as shown in Figure 24. Unfortunately, this increased effectiveness comes at the cost of decreased throughput experienced by competing TCP flows as the number of parallel TCP flows increases or TCP/k increases k.

To evaluate the effectiveness of TCP/k in higher bandwidth-delay product networks, we change the link bandwidth between Sender Node 0 and the NIST Net Emulator node from 100Mbps to 1Gbps[4]. Now Node 0 opens a single TCP-SACK or a single TCP/k connection to the NIST Net emulator. The RTT between Node 0 and

[4]For the gigabit link, TCP system parameters are configured as follows:

- `/proc/sys/net/core/wmem_max` = 33554432.
- `/proc/sys/net/core/rmem_max` = 33554432.
- `/proc/sys/net/ipv4/tcp_wmem` = "4096 33554432 33554432".
- `/proc/sys/net/ipv4/tcp_rmem` = "4096 33554432 33554432".

NIST Net Emulator is set to 50msec by the NIST Net Emulator (NIST Net Emulator also works as a receiver in this experiment). Although the physical link between these two nodes is an optical Gigabit Ethernet, the hardware performance (533MHz front-side bus, 32bit PCI bus) of the sender and receiver nodes in our test-bed limited the maximum achievable throughput to about 440Mbps.

Figure 26(a) shows the achieved throughput by Node 0 when it opens a TCP-SACK flow to the destination and the changes of the congestion window size of the TCP-SACK flow during the experiment. The figure clearly shows that a single TCP-SACK flow is not effective in the utilization of available bandwidth: It takes about 250 sec to reach the maximum throughput, and the TCP-SACK flow decreases its congestion window drastically with a congestion event at about 320 sec. In contrast, Figure 26(b) shows the achieved throughput by Node 0 when it opens a TCP/k flow with $k = 4$ (i.e., TCP/4) to the destination and the changes of the congestion window size of the TCP/4 flow during the experiment. The figure shows that TCP/4 can utilize the available bandwidth more effectively than standard TCP: The TCP/4 flow reaches the maximum throughput at about 60 sec. The figure clearly shows that TCP/4 increases its sending rate faster and decreases its congestion window less than standard TCP.

However, the results in Figure 26 also illustrate the weaknesses of TCP/k (also k parallel TCP). First, TCP/4 flow is still not aggressive enough for the fast utilization of available bandwidth during the first 60 seconds. Second, there are many packet losses in TCP/4 when it reached maximum bandwidth because of its fixed aggressiveness. Third, the fixed strength of TCP/4 is clearly problematic when it competes against standard TCP flows since it is as unfair as four parallel TCP flows. Furthermore, it is hard to decide an adequate strength k for TCP/k (or number of flows for parallel TCP flows) so as to effectively and fairly use the dynamically changing network resources. From these observations, in the next section we propose a new congestion control scheme that automatically tunes its strength k in response to network situation to achieve both effective utilization of available bandwidth and fairness against standard TCP.

D. Competition Detection

While competition in the network not necessarily leads to congestion, it always causes *faster queue build-up*. We take advantage of this to formulate a competition detector. The rationale for this relies on two observations. First, *cwnd* increases at a rate proportional to RTT, which in turn increases only slightly with increasing queuing at the bottleneck link. Second, queue build-up happens as a result of incoming traffic to the queue. Therefore, *cwnd* for a given queue length will tend to be smaller when queue build-up is faster. We estimate the queue build-up speed by monitoring the congestion window size and the measured queuing delay. To do this we define *knee congestion window* (*knee_cwnd*) to be the congestion window size when estimated queue build-up ($srtt - srtt\_min$) exceeds some threshold, which is defined in terms of a portion of the estimated maximum build-up ($srtt\_max - srtt\_min$). Here, *srtt* is smoothed round-trip time and *srtt_min* and *srtt_max* is the minimum and maximum of the smoothed round-trip time, respectively. We use 20% of queuing delay increase as the reference level because it gives early notification with relatively small amount of queue build-up. The term *knee* and its concept is borrowed from [22], but A-TCP determines *knee* using a simple delay level checking instead of the delay gradient.

The effect of competition on *knee_cwnd* is illustrated in Figure 27 for the case of standard TCP-SACK. For this experiment, we use our test-bed shown in Figure 20. In the test-bed, all the links are set to 100Mbps and RTT is set to 50msec using

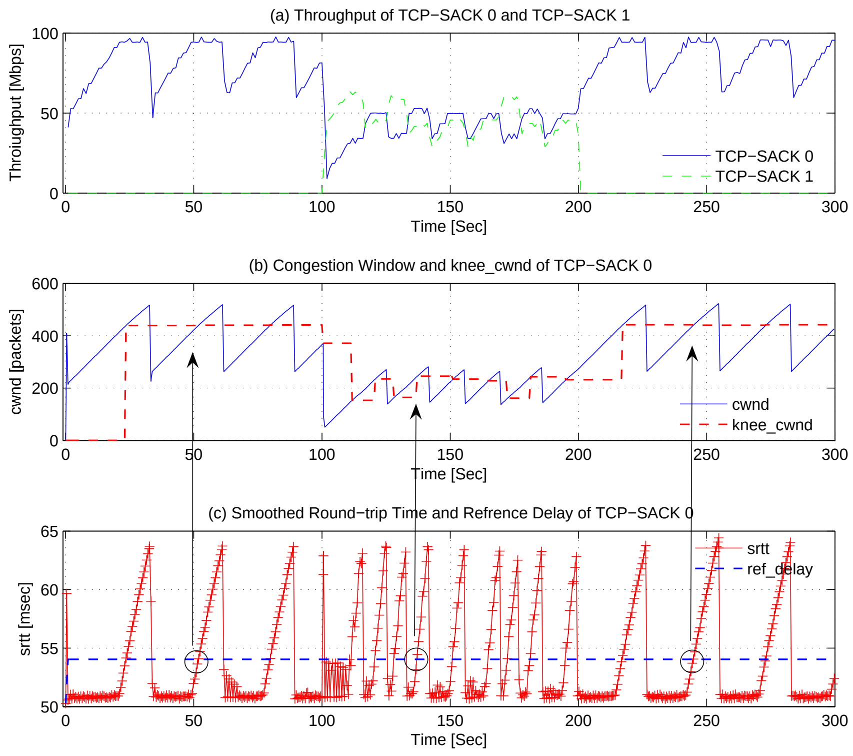


Fig. 27. TCP-SACK Flows in 100Mbps, 50msec RTT Network

the NIST Net Emulator. During the experiment, a TCP-SACK flow (TCP-SACK 0) from Node 0 sends traffic to Node 2 for 300 seconds while another TCP-SACK flow (TCP-SACK 1) from Node 1 joins the network by sending traffic to Node 2 after 100 sec for the next 100 seconds.

The throughputs achieved by the two TCP-SACK flows in Figure 27(a) show that the two TCP-SACK flows share the network bandwidth fairly during the overlapping 100 seconds. Figure 27(b) and (c) track the congestion window size and smoothed round-trip time (*srtt*) of the TCP-SACK flow from Node 0, respectively. Figure 27(b) also shows the changes of *knee_cwnd*, which is updated whenever *srtt* reaches the reference queuing delay *ref_delay*. The arrows between Figure 27(b) and Figure 27(c) indicate how *knee_cwnd* is updated. The *ref_delay* line in the Figure 27(c) represents the reference queuing delay level. Note that the TCP-SACK used in the experiment is not affected by these added variables, which are added solely to illustrate the competition detection mechanism, not to change the behavior of TCP-SACK.

During the first 100 seconds we can see that *knee_cwnd* remains almost the same value, i.e., *knee_cwnd* does not change much in each measurement. When the competing TCP-SACK flow from Node 1 joins the network at 100sec, *knee_cwnd* of the TCP-SACK flow from Node 0 decreases about half from the previous value. During the overlapping 100 seconds, *knee_cwnd* continues to change. After the competing TCP-SACK flow leaves the network at 200 sec, *knee_cwnd* becomes stable again. However, since *knee_cwnd* gets update only when the *srtt* crosses over the reference delay, the change in the competition does not get reflected until the next queue build-up. In Figure 27, for example, the next queue build happened 30 sec after tear down of the competing flow.

We use the changes of *knee_cwnd* illustrated in the experiment to formulate a competition detector: When there is no competition, *knee_cwnd* remains almost at the same value. If there is competition with a greedy flow like TCP, *knee_cwnd* continuously fluctuates especially when the competition starts and finishes. In the next section, we combine this simple competition detection scheme to TCP/k and propose a new TCP congestion control scheme that achieves both effective and fair utilization of high bandwidth-delay product networks.

E. Adaptive TCP

In this section, we show how the easy tunability of TCP/k can be used as the basis of an adaptive protocol that operates aggressively (large k) when no other flows compete for bandwidth, and reduces its aggressiveness (small k, $k = 1$) whenever it detects other flows. In order for such a scheme to work it must combine two characteristics:

accurate competition detection and effective adaptation of aggressiveness once competition is detected or bandwidth becomes available. To realize this we combine the tunable aggressiveness of TCP/k and the competition detection mechanism in the previous section (in addition to traditional congestion detection) and propose a new TCP congestion control scheme called *Adaptive TCP* (A-TCP).

The basic idea behind A-TCP is to quickly increase the strength k to effectively utilize available bandwidth (or unclaimed bandwidth by other flows) when there is no competition, and enter non-aggressive mode by setting the strength of A-TCP to $k = 1$ to ensure TCP-friendliness whenever it detects competition with other flows. Note that A-TCP (and TCP/k) is identical to standard TCP when its strength $k = 1$. In addition, A-TCP sets $k = 1$ during periods of congestion to reduce packet loss regardless of the result of competition detection. Here, by congestion we mean that the network queuing delay estimated by A-TCP is larger than a reference queuing delay level. We use the same value (i.e. 20% queue-build-up) for this as the threshold queuing delay level used to trigger the competition detector.

The details of A-TCP mechanics are shown in Algorithm 2, Algorithm 3, and Algorithm 4. As the algorithms suggest, A-TCP uses the queuing delay increase (indicating queue build-up) as an estimator for congestion and competition. It does so by keeping track of the minimum and maximum of the smoothed round-trip time[5] ($srtt\_min$ and $srtt\_max$, respectively) and triggers detection when the estimated queue build-up ($srtt - srtt\_min$) exceeds some threshold. (γ value in Algorithm 2 decides this threshold.)

One of the difficulties in the use of the delay variation information for the TCP

[5] i.e., $srtt$. This is also called moving averaged round-trip time, and it is updated whenever new RTT is available. $srtt \leftarrow (1 - w) * srtt + w * rtt$. Here, rtt is high-resolution version RTT and $w = \frac{1}{8}$. Also, with a new $srtt$, $srtt\_max \leftarrow srtt$ if $srtt\_max < srtt$, $srtt\_min \leftarrow srtt$ if $srtt\_min > srtt$.

Algorithm 2 Parameters and initialization of variables of A-TCP

```
Parameters:
  $\alpha \leftarrow 10$; //waiting time after the last checking
  $\delta \leftarrow 10$; //waiting time after packet loss
  $\beta \leftarrow 0.1$; //margin of error
  $\gamma \leftarrow 0.2$; //reference queuing delay level
Variables:
  $k \leftarrow 1$; //strength of A-TCP
  $knee\_cwnd \leftarrow 0$; //congestion window size at a knee
  $knee\_cwnd\_ref \leftarrow 0$; //reference $knee\_cwnd$
  $knee\_k \leftarrow 0$; //strength $k$ at a knee
  $check\_knee \leftarrow 0$; //check $knee\_cwnd$ when it is 1
  $low\_delay \leftarrow 0$; //low queuing delay when it is 1.
  $srtt\_max \leftarrow 0$; //maximum $srtt$ (smoothed round-trip time)
  $srtt\_min \leftarrow$ 0xffffffff; //minimum $srtt$
  $modify\_time \leftarrow$ now; //last time when $k$ was modified.
  $wait\_time \leftarrow 0$; //waiting time for the next checking.
```

congestion control is the coarse resolution of the timers in most TCP implementations: Most TCP implementations use 10msec resolution, which is not sufficiently accurate to capture delay variations. To overcome this, we use the high-resolution (1 micro second) system time for round-trip time estimation in A-TCP. We modified the round-trip time estimation functions in the Linux kernel to use additional micro second resolution system timer. Thus, the timing variables such as *srtt*, *srtt_max*, and *srtt_min* used in Algorithm 3 are in usec resolution[6]. However, note that the use of usec resolution system timer is only for experimental purpose in Linux kernel 2.4. We believe that we do not need to depend on this if we use 1 msec TCP timers in Linux kernel 2.6 versions.

Whenever A-TCP detects congestion (i.e., *srtt* crosses over *ref_delay*), A-TCP

[6]We do not modify timing behavior and time related TCP variables of existing TCP. The *srtt* in Algorithm 3 is added additionally for A-TCP, and it has a name of *srtt_usec* in actual A-TCP implemented kernel to differentiate it from existing *srtt*.

Algorithm 3 With a new ACK

```
1: update $srtt$, $srtt\_min$, and $srtt\_max$;
2: if now $-$ $modify\_time > wait\_time$ then //check if it is time to work.
3:   if $srtt \leq srtt\_min + \gamma \cdot (srtt\_max - srtt\_min)$ then //check if queuing delay is lower
     than this reference queuing delay level.
4:     if ($check\_knee = 1$) and ($knee\_cwnd\_ref = knee\_cwnd$) then //increase $k$ until a
       knee is reached.
5:       $k \leftarrow k + 1$;
6:     end if
7:     $low\_delay \leftarrow 1$;
8:   else
9:     if $low\_delay = 1$ then //queuing delay becomes bigger than the threshold.
10:      $knee\_cwnd \leftarrow$ current $cwnd$; //set new $knee\_cwnd$
11:      if $knee\_cwnd\_ref = 0$ then //set $knee\_cwnd\_ref$ with new $knee\_cwnd$
12:        $knee\_cwnd\_ref \leftarrow knee\_cwnd$;
13:      else if $((1-\beta) \cdot knee\_cwnd\_ref \leq knee\_cwnd \leq (1+\beta) \cdot knee\_cwnd\_ref)$ then
14:        $knee\_cwnd\_ref \leftarrow knee\_cwnd$;
15:      end if
16:      if $check\_knee = 1$ then //set new $knee\_k$ with current $k$.
17:        $knee\_k \leftarrow k$;
18:      end if
19:      $check\_knee \leftarrow 0$; //stop aggressive behavior (i.e., increasing $k$).
20:      $low\_delay \leftarrow 0$;
21:    end if
22:    $k \leftarrow 1$;
23:  end if
24:  $modify\_time \leftarrow$ now;
25:  $wait\_time \leftarrow \alpha \cdot srtt\_min$; //set waiting time until the next checking.
26: end if
```

records current *cwnd* as *knee_cwnd* and current k as *knee_k*. Then, it prepares imminent packet loss by setting $k = 1$ to reduce packet loss. *knee_k* is later used to decide the reduction amount of congestion window size when packet loss happens and competition is not detected. Also, A-TCP updates *knee_cwnd_ref* with the new *knee_cwnd* if *knee_cwnd* is less than ±10% different from *knee_cwnd_ref*. *knee_cwnd_ref* is used to track the changes of *knee_cwnd* by keeping previous value of *knee_cwnd*. The 10% margin is given for possible measurement errors and it is decided by β parameter in Algorithm 2. This parameter decides the sensitivity of competition detection because

Algorithm 4 With packet loss

```
1: if $first = 1$ then //ignore the first packet loss caused by initial slow-start.
2:     $k \leftarrow 1$; $knee\_k \leftarrow 1$; $knee\_cwnd \leftarrow 0$; $knee\_cwnd\_ref \leftarrow 0$; $first \leftarrow 0$;
3: else if $knee\_cwnd\_ref = knee\_cwnd$ then //enter aggressive mode.
4:     $k \leftarrow \max(\sqrt{knee\_k}, 1)$;
5:     $srtt\_max \leftarrow srtt\_min$;
6: else //enter non-aggressive mode.
7:     $k \leftarrow 1$; $knee\_k \leftarrow 1$; $knee\_cwnd\_ref \leftarrow 0$;
8: end if
9: $check\_knee \leftarrow 1$;
10: $low\_delay \leftarrow 0$;
11: $modify\_time \leftarrow$ now;
12: $wait\_time \leftarrow \delta \cdot srtt\_min$; //set waiting time until the next checking.
```

A-TCP might not detect changes of competition level if the margin is too big[7]. A-TCP uses the relationship between the two variables, $knee\_cwnd$ and $knee\_cwnd\_ref$, to determine its operation mode in the next congestion epoch: If the two variables are set to the same value, A-TCP will be highly aggressive when and after it responds to a congestion event such as three duplicate ACKs. Otherwise, A-TCP infers that competition along the path has changed and remains non-aggressive during the entire next congestion epoch by holding $k = 1$. Also, A-TCP resets $knee\_cwnd\_ref = 0$ to set $knee\_cwnd\_ref$ to new $knee\_cwnd$ that will be measured in the next congestion epoch as shown in Line 11 of Algorithm 3.

As Algorithm 3 describes, A-TCP keeps increasing k per each checking period until it detects congestion when competition is not detected in the previous congestion epoch (i.e. $knee\_cwnd\_ref = knee\_cwnd$). The checking period is defined in terms of multiples of the minimum smoothed round-trip time ($srtt\_min$) and it is one of the design parameters of A-TCP (α and δ) as shown in Algorithm 2. They decide

[7]While the 10% margin seems a reasonable trade-off between the sensitivity of the detection and possible measurement errors in real networks and gives good behavior in our experiments and simulations, it still needs more investigation.

how quickly A-TCP responds to network variations. In this work we use $\alpha = \delta = 10$, so that the checking period is 10 *srtt_min*. (Although these values for α and δ work fine in our experiments, they need more investigation in the future.) When a packet loss happens, A-TCP compares current $knee\_cwnd\_ref$ and *knee_cwnd before* responding to the packet loss. If competition has been detected (i.e. $knee\_cwnd\_ref \neq knee\_cwnd$) A-TCP resets variables: $k = 1$, $knee\_k = 1$, and $knee\_cwnd\_ref = 0$, and keep $k = 1$ during the entire next congestion epoch. Otherwise, it sets its strength k to $\sqrt{knee\_k}$ before reducing its congestion window. As a result, A-TCP reduces its congestion window by $\frac{cwnd}{2\sqrt{knee\_k}}$ instead of $\frac{cwnd}{2}$ when competition is not detected.

In the following sections, we will explain the details of the behavior of A-TCP using experimental examples and show that Algorithm 3 and Algorithm 4, in conjunction with high-resolution timers, provide an accurate estimation of network status and thus enable A-TCP to adequately control its strength k to achieve high utilization of available bandwidth with good TCP-friendliness.

F. Effectiveness of A-TCP

We first show how effectively A-TCP utilizes high bandwidth-delay product networks. For this, we repeat the same experiment done for Figure 26 in Section C with an A-TCP flow. In Figure 28 we show the achieved throughput and changes of congestion window of the A-TCP flow in the experiment. The achieved throughput by an A-TCP flow in Figure 28 shows that A-TCP utilizes the link bandwidth faster than a TCP-4 flow shown in Figure 26(b). We also show the changes of congestion window size of the A-TCP flow in Figure 28 to illustrate behavioral difference of A-TCP from standard TCP and TCP-4 shown in Figure 26. Further, in Figure 29, we show the changes of several other variables inside the A-TCP flow during the experiment, such

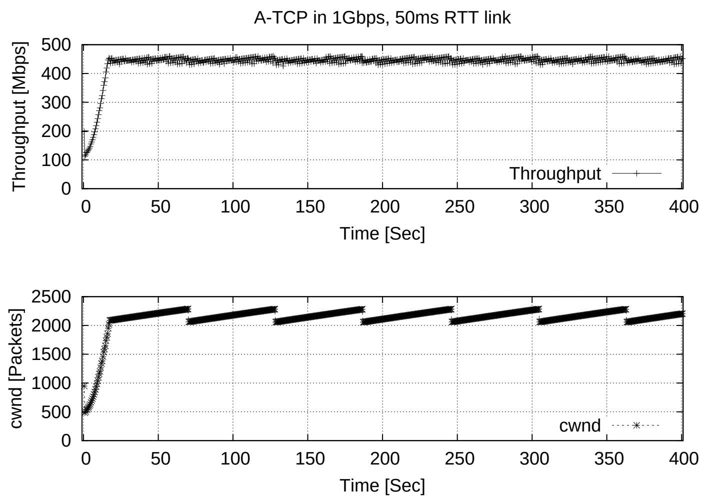


Fig. 28. A-TCP in 1Gbps, 50msec RTT Link

as strength k and $knee\_cwnd$.

We can see from Figure 29(c), the A-TCP flow starts in aggressive mode. The A-TCP flow increases its strength k rapidly up to $k = 34$. This in turn causes the congestion window size of the A-TCP flow to increase fast as well, as Figure 28 shows. This rapid initial increase of the strength k can be explained with the changes in $srtt$ and $knee\_cwnd$ shown in Figure 29(a) and Figure 29(b). During the first 20 seconds of Figure 29(a), $srtt$ is smaller than ref_delay line shown in the figure. ref_delay represents the reference queuing delay level in Line 3 of Algorithm 3. At about 20 sec, $knee\_cwnd$ and $knee\_cwnd\_ref$ in Figure 29(b) and $knee\_k$ in Figure 29(c) are updated with the current $cwnd$ and k at the time, then the strength k is set to one because the congestion detector triggers, i.e., $srtt$ exceeds the reference queuing delay level (this is caused by self congestion in this case).

The A-TCP flow stays non-aggressive by holding its strength k at one until the first packet loss happened at about 70 sec. With the packet loss event, A-TCP checks if $knee\_cwnd$ is equal to $knee\_cwnd\_ref$. Since it is true in this example, A-TCP sets its strength k to $\sqrt{knee\_k}$, i.e. 5, and it reduces the congestion window

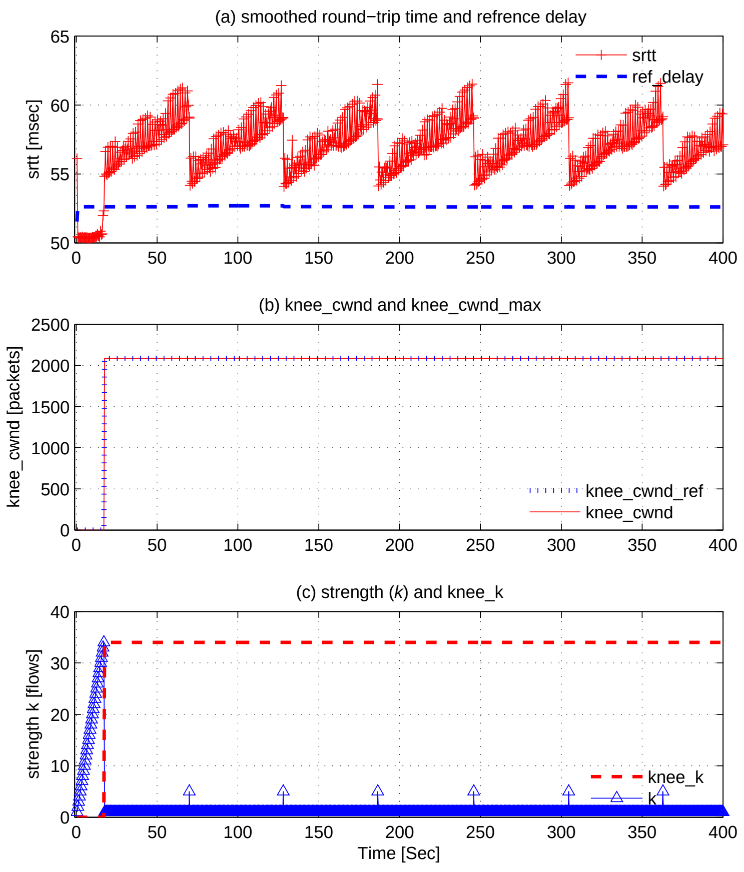


Fig. 29. Changes of the Variables Inside A-TCP

by $\frac{1}{2*k}cwnd = \frac{1}{10}cwnd$ instead of halving it. After that, the A-TCP flow resets its strength to one again because of the congestion detector. Note that *knee_k* and other variables are not updated at this time because the variables can only be updated when *srtt* becomes bigger that the reference queuing delay level as shown in Algorithm 3. This pattern keeps repeating and the achieved throughput by the A-TCP flow in Figure 28 remains stable.

Figure 30 illustrates A-TCP's congestion window changes in the steady-state. By steady-state we mean the period for which throughput of A-TCP remains stable because it does not detect competition (i.e., *knee_cwnd* remains equal to *knee_cwnd_ref*) as the example in Figure 28 shows. The behavioral differences between A-TCP shown

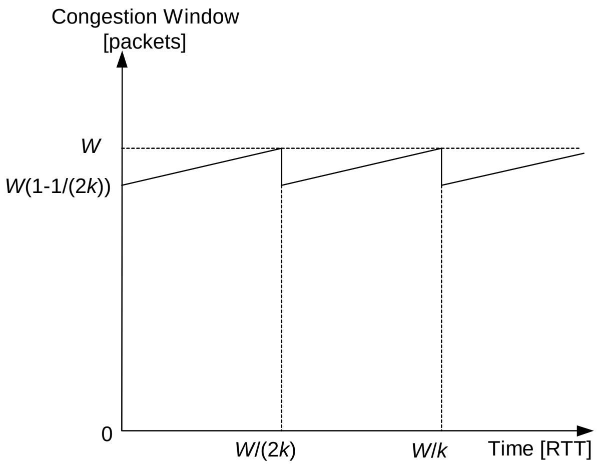


Fig. 30. Changes of the Congestion Window of A-TCP

in Figure 30 and TCP/k shown in Figure 24 are: (i) the congestion window increase speed of A-TCP is the same as that of standard TCP because the self-induced queuing delay, (ii) when A-TCP encounters packet loss, the strength k of A-TCP is decided by the $knee\_k$ value instead of the fixed k in TCP/k. Thus, it takes $\frac{W}{2k}$ RTT times to recover the congestion window compared to $\frac{W}{2k^2}$ of TCP/k.

We use Figure 30 to analyze the steady-state throughput of A-TCP. We let S be the total amount of packets sent in each congestion epoch. From Figure 30, we can see that $S = W(1-\frac{1}{2k})\frac{W}{2k} + \frac{1}{2}\frac{W}{2k}\frac{W}{2k}$. Thus, the packet loss probability p equals to $\frac{1}{S}$, and $\frac{1}{p}$ can be expressed as:

$$\frac{1}{p} = S = \frac{W^2}{2k}\frac{4k-1}{4k} = W^2\frac{4k-1}{8k^2}. \tag{4.4}$$

As a result, $W = (\frac{1}{p}\frac{8k^2}{4k-1})^{1/2}$. Hence, the average throughput T [packets/sec] of an A-TCP flow in steady state can be modeled as:

$$\begin{aligned} T &= \frac{\text{data sent per congestion epoch}}{\text{time per congestion epoch}} \\ &= \frac{W^2(\frac{4k-1}{8k^2})}{\frac{W}{2k}RTT} \\ &= \frac{\sqrt{\frac{4k-1}{2}}}{RTT\sqrt{p}} \text{ [packets per sec].} \end{aligned} .$$

However, in contrast to the fixed strength k in TCP/k, the strength k of A-TCP is dynamically decided by *knee_k*. Hence, we can replace k with $\sqrt{knee\_k}$ and obtain the following equation:

$$T = \frac{\sqrt{\frac{4\sqrt{knee\_k}-1}{2}}}{RTT\sqrt{p}} \text{ [packets per sec].} \tag{4.5}$$

G. Fairness of A-TCP

1. 100Mbps Case

To illustrate how A-TCP works when there is competition by other flows and to evaluate the performance of A-TCP, we repeat the experiments done in Section B. All the links in the test-bed of Figure 20 are set to 100Mbps and RTT is set to 50msec using the NIST Net Emulator. Differently from the experiments done in Section B, now Node 0 opens an A-TCP flow and sends traffic to Node 2 for 300 seconds while a TCP-SACK flow from Node 1 joins the network by sending traffic to Node 2 at 100 sec for the next 100 seconds. Figure 31 and Figure 32 show the results of the experiment.

Figure 31 shows the achieved throughput by the A-TCP and TCP-SACK flows. From the experimental results in the figure we can see that A-TCP achieves high throughput when it is alone in the test-bed during the first 100 and the last 80 seconds.

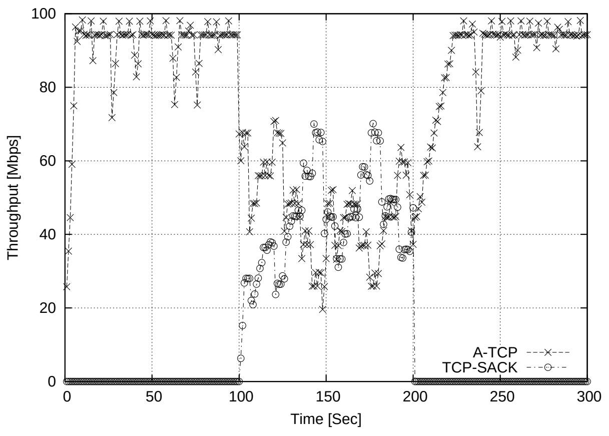


Fig. 31. A-TCP vs. TCP-SACK in 100Mbps, 50msec RTT Network

Further, the results also show that the A-TCP flow and the competing TCP-SACK flow share the bandwidth fairly during the overlapping 100 seconds. This was not the case for the other high-speed TCP proposals in Section B. By comparing the experimental results in the figure with the results of other high-speed TCP proposals in Figure 21, Figure 22, and Figure 23 we can clearly see that A-TCP is significantly fairer to the competing TCP-SACK flow than other proposals.

To illustrate how A-TCP behaves in the experiment, we also show variables inside the A-TCP flow, such as *srtt* and strength k in Figure 32. Figure 32(a) shows the changes of the congestion window sizes of the A-TCP and TCP-SACK flows. Figure 32(b) tracks *knee_cwnd* and *knee_cwnd_ref*, which in turn used to detect competition. The *ref_delay* line in the Figure 32(c) represents the reference queuing delay level. From the relationship between the changes of the variables, we can see how *knee_cwnd* is updated whenever the *srtt* reaches the reference queuing delay level. Figure 32(d) shows the changes of k and *knee_cwnd*.

From Figure 32(a) we can see that the A-TCP flow increases its congestion window size fast when it starts because its strength k increases rapidly as shown in Figure 32(d) until it detects queuing delay increase at about 7 sec as shown in Figure

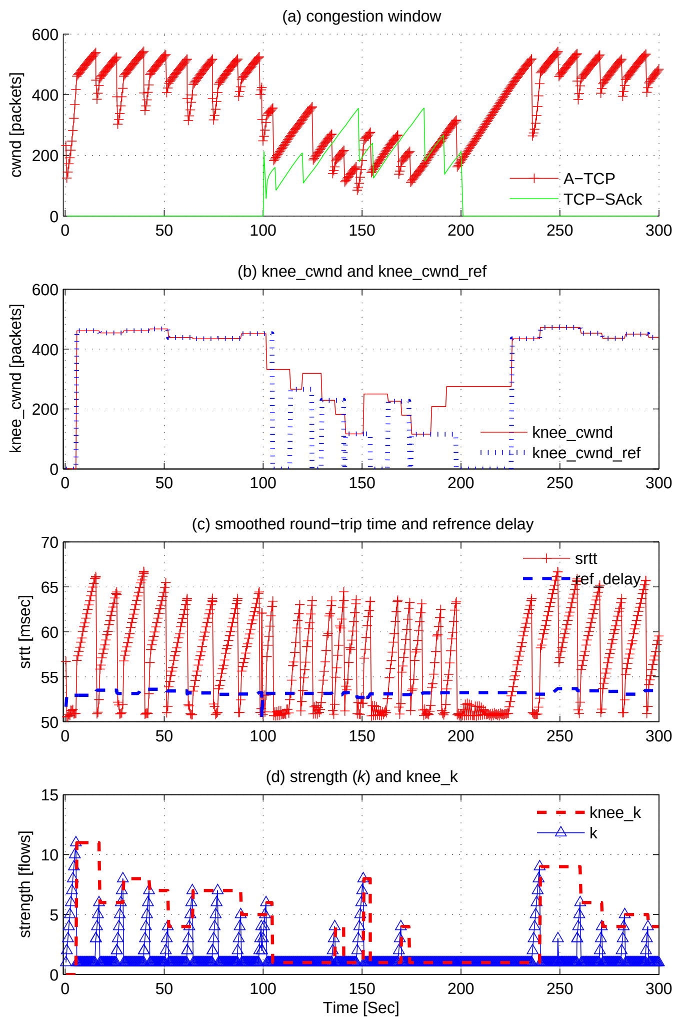


Fig. 32. Changes of the Variables Inside A-TCP

32(c). At this point A-TCP sets its strength k to one and keeps it until packet loss happens at about 12 sec. When packet loss eventually happens, A-TCP responds to the packet loss by setting $k = 3$ before reducing its congestion window because the relationship between $knee\_cwnd$ and $knee\_cwnd\_ref$ indicates no competition (i.e., $knee\_cwnd = knee\_cwnd\_ref$) and $knee\_k = 11$. As a result, the A-TCP flow reduces its congestion window by $\frac{cwnd}{6}$ instead of $\frac{cwnd}{2}$ and increases k after the reduction of the congestion window. This aggressive behavior of A-TCP is repeated until the competing TCP-SACK flow from Node 1 joins the test-bed network at 100 sec.

Just after a TCP-SACK flow from Node 1 joins the network at 100sec, Figure 32(b) shows that *knee_cwnd* of the A-TCP flow is set to a different value from *knee_cwnd_ref*. A-TCP infers this change as a signal of competition and enters non-aggressive mode during the next congestion epoch (until about 125 sec) as the changes of the congestion window shown in Figure 32(a). During the overlapping 100 seconds, as figure 32(b) shows, *knee_cwnd* and *knee_cwnd_ref* variables of A-TCP are set to have different values from each other continuously by the existence of the competing TCP-SACK flow. As a result, the A-TCP flow is forced to be in non-aggressive mode repeatedly although it tries to increase k several times (e.g., at around 150 sec) during the overlapping 100 seconds. This example illustrates how the competition detection mechanism drives A-TCP to being fair to competing flows even if the competing flows is single, standard TCP flow.

After the competing TCP-SACK flow leaves the network at 200 sec, *knee_cwnd* and *knee_cwnd_ref* are set to have the same value at about 230 sec. After that point the two variables remain equal, and A-TCP stays in aggressive mode. However, since *knee_cwnd* gets updated only when the *srtt* crosses up the reference delay, the change in the competition is not detected until the next queue build-up. This in turn leads to a delay in transition back to aggressive mode once the competition period has ended in 200 sec. In Figure 32(c), for example, the next queue build happened 30 sec after tear down of the competing flow.

To compare TCP-friendliness of A-TCP with other high-speed TCP proposals we repeat the experiments in Section B 10 times for A-TCP and other high-speed TCP proposals against a TCP-SACK flow. Figure 33 shows the achieved throughput by each flow when a TCP-SACK flow from Node 1 competes against a high-speed TCP flow of different types from Node 0 for the overlapping 100 seconds. The lengths

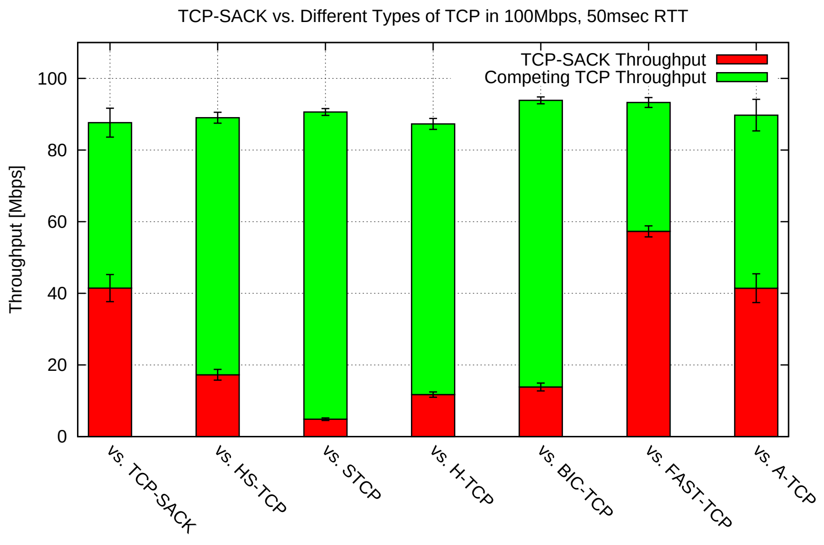


Fig. 33. Achieved Throughput by TCP-SACK and Various Types of TCP in 100Mbps, 50msec RTT Network

and error-bars of the histogram in the figure represent the average throughput and 95% confidence interval of the average achieved by each flow.

The first bar in Figure 33 shows that the crossing TCP-SACK flow from Node 1 achieves about 41Mbps when it competes against a TCP-SACK flow from Node 0. We use this value as a reference for our fairness evaluation of TCP proposals (the closer to this value, the better the fairness of the TCP proposal.) The experimental results in Figure 33 clearly show that the TCP-SACK flow from Node 1 achieves a much smaller throughput than its fair share when it competes against a HS-TCP, STCP, H-TCP, and BIC-TCP flow. Also, as expected, the TCP-SACK flow achieves much higher throughput than its fair share when it competes against a FAST-TCP flow. The last bar in the figure shows that the A-TCP flow allows the competing TCP-SACK flow to achieve similar throughput to its fair share.

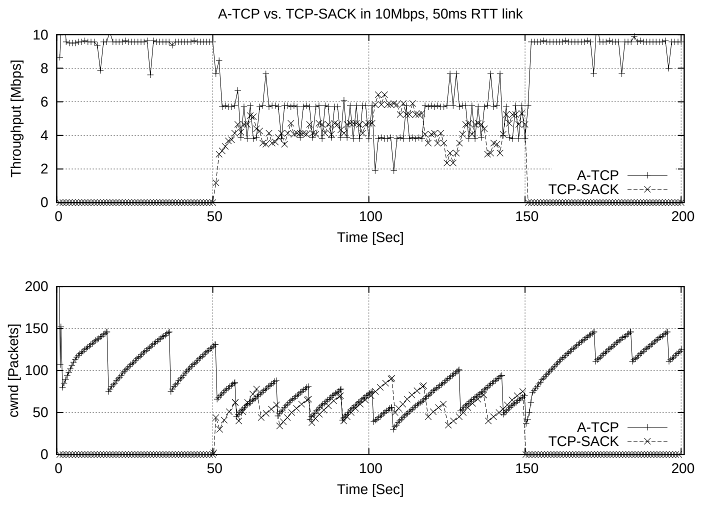


Fig. 34. A-TCP vs. TCP-SACK in 10Mbps, 50msec RTT Network

2. 10Mbps Case

In this section we repeat the experiments of the previous section with 10Mbps bottleneck link. For this experiment we changed the link bandwidth from NIST Net Emulator to the receiver Node 2 to 10Mbps in the test-bed shown in Figure 20. The bandwidths of other links are all 100Mbps, and the 50msec RTT delay is unchanged. Figure 34 shows experimental results of A-TCP against single TCP-SACK flow in the 10Mbps, 50msec bottleneck link. The achieved throughputs by A-TCP and TCP-SACK in Figure 34 shows that the A-TCP flow from Node 0 is fair to the competing TCP-SACK flow from Node 1. The changes of the congestion window size of the A-TCP flow in Figure 34 shows almost the same behavior as that of the competing standard TCP-SACK flow. This is because A-TCP does not have many chances to increase its strength k in this low bandwidth case. This result demonstrates the good adaptivity of A-TCP to network conditions: A-TCP does not increase the strength because a single TCP flow is enough for the high utilization of this not-so-high bandwidth-delay product network.

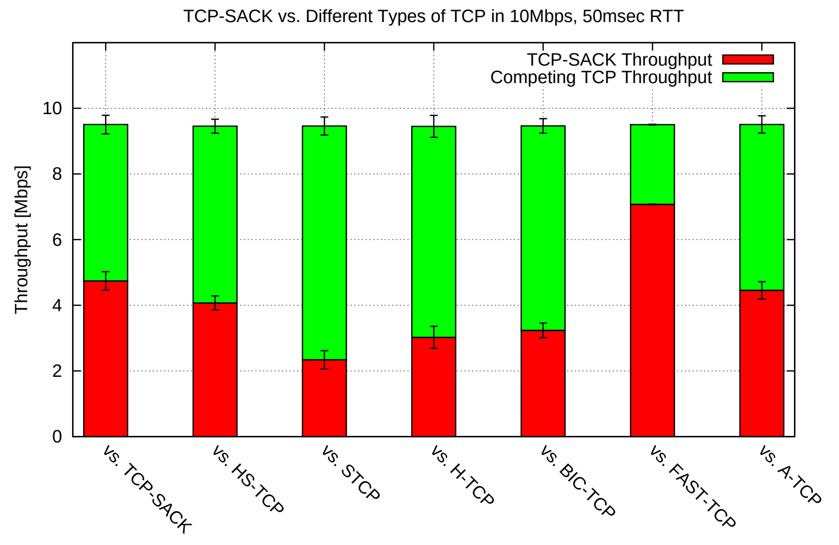


Fig. 35. Achieved Throughput by TCP-SACK and Various Types of TCP in 10Mbps, 50msec RTT Network

In Figure 35 we show experimental results obtained by repeating the above experiment to each high-speed TCP proposal. The first bar of the figure shows that the TCP-SACK flow from Node 1 achieves about 4.7Mbps when it competes against a TCP-SACK flow from Node 0. We use this value as a reference to evaluate the fairness property of TCP proposals in 10Mbps bottleneck, 50msec RTT network. In the figure, the achieved throughput by crossing TCP-SACK flow is less than its fare share when it competes against HS-TCP, STCP, H-TCP, and BIC-TCP. But it is better than the previous 100Mbps case. HS-TCP, especially, shows much improved fairness in this 10Mbps bottleneck link experiment. This is because HS-TCP becomes a standard TCP when its congestion window size is less than a threshold value (i.e. 38 packets [37]). Again, the TCP-SACK flow achieves much higher throughput than its fair share when it competes against FAST-TCP. The last bar in the figure shows that the crossing TCP-SACK flow from Node 1 achieves almost its fair share of throughput when it competes against an A-TCP flow from Node 0.

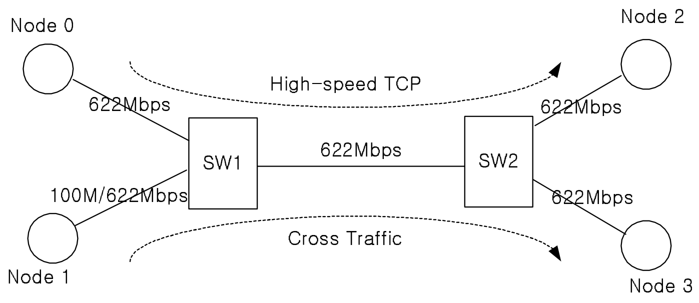


Fig. 36. ns2 Simulation Test-bed

H. Simulation Results

To further evaluate the performance of A-TCP and for the flexibility of the evaluation, we also implement A-TCP in ns2 Version 2.27 [37]. For the test-bed topology we use a simple dumbbell topology shown in Figure 36 with bottleneck link bandwidth of 622Mbps and 50ms RTT propagation delay between senders and receivers. The bottleneck switch SW 1 uses FIFO/Drop-Tail scheduling with 20% of the bottleneck bandwidth-delay product. We again use TCP-SACK for the competing standard TCP, and TCP receivers use delayed ACK. TCP initial congestion window size is set to two packets instead of one to remedy round-tip time measurement errors by the use of delayed ACKs when connections initiate. FTP is used for application to generate continuously back-logged, long-term flows.

1. Intra-Protocol Fairness

We first evaluate intra-protocol fairness between A-TCP flows. For this experiment both Node 0 and Node 1 use A-TCP. Node 0 opens an A-TCP connection (A-TCP 1) to Node 2 for 1500 seconds. Node 1 joins the experiment by opening a competing A-TCP flow (A-TCP 2) to Node 3 from 400 sec to 1000 sec. Figure 37 shows the achieved throughput by the two A-TCP flows and the changes of their congestion

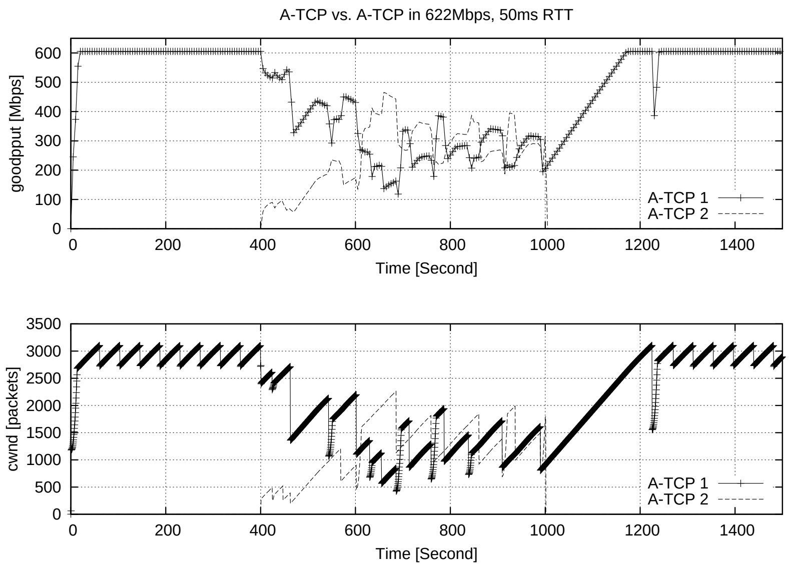
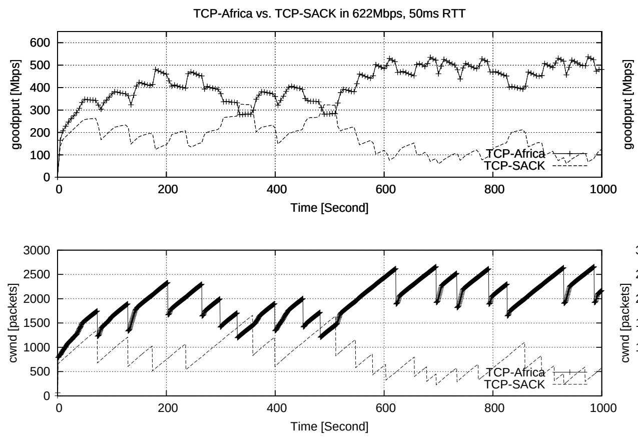


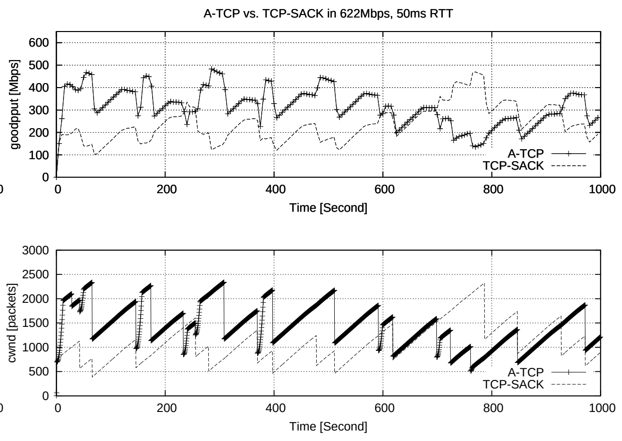
Fig. 37. A-TCP vs. A-TCP in 622Mbps, 50msec RTT Network

window sizes during the experiment. The figure shows that A-TCP 1 flow utilizes bandwidth effectively by keeping aggressive mode when it is alone during the first 400 seconds. After A-TCP 2 joins the network at 400 sec, we can see that the two A-TCP flows share the bottleneck link fairly.

2. Inter-Protocol Fairness

In this section, we evaluate and compare inter-protocol fairness of A-TCP with other high-speed TCP proposals including TCP-Africa using ns2. We first let an A-TCP flow and a standard TCP flow start to send traffic through the same bottleneck link at the same time. Then, we compare the test results with those of TCP-Africa, which controls its operation mode from fast to slow mode using a delay-based congestion detector only. When we test TCP-Africa using our ns2 implementation, we use the default α parameter value given in the authors' paper, and tcpTick_ of ns2 was set to 0.1ms to support high-resolution *srtt* that is required for TCP-Africa. For this simulation, Node 0 in Figure 36 opens an A-TCP or TCP-Africa connection for 1000





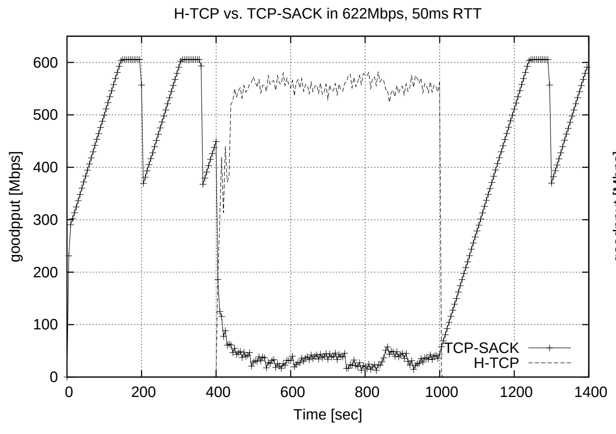
(a) (b)

Fig. 38. TCP-SACK and (a) TCP-Africa (b) A-TCP Start at the Same Time

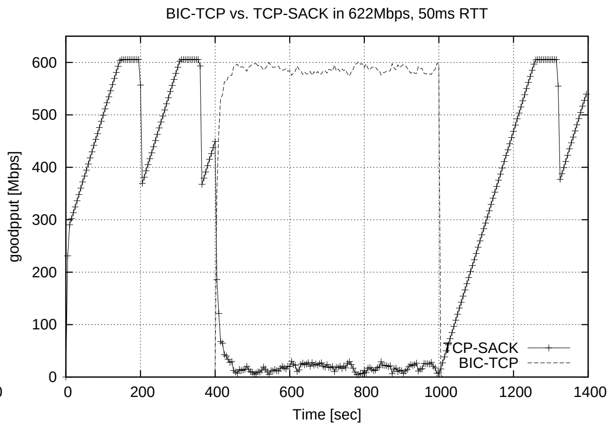
seconds to Node 2. Node 1 opens a TCP-SACK flow to Node 3 for the same period.

Figure 38(a) and Figure 38(b) show the simulation results. In the figures we show the changes of the achieved throughput and congestion window size of the participating TCP flows. The achieved throughput of the TCP-Africa and the competing TCP-SACK flows in Figure 38(a) show that TCP-Africa eventually dominates the network and not allow the competing TCP-SACK its fare share of bandwidth. The changes of the congestion window size of the TCP-Africa shown in the bottom of Figure 38(a) clearly show that congestion detection alone is not sufficient to mitigate aggressiveness of TCP-Africa. In contrast to the simulation results with TCP-Africa, the simulation results of A-TCP in Figure 38(b) show that A-TCP is fairer to the competing TCP-SACK flow than TCP-Africa. The changes of the congestion window size of A-TCP shows that the aggressiveness of the A-TCP flow is repeatedly set to that of a standard TCP. This is because the competition detection mechanism in A-TCP detects the existence of the competing TCP repeatedly during the simulation.

Next we test A-TCP's fairness with standard TCP when A-TCP joins network later than a TCP-SACK flow. We also compare the experimental results with other



(a)

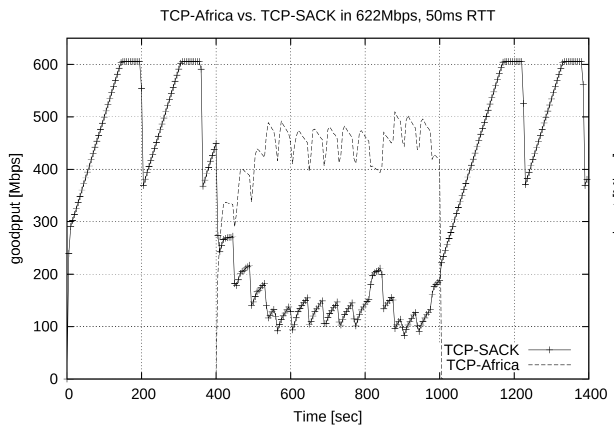


(b)

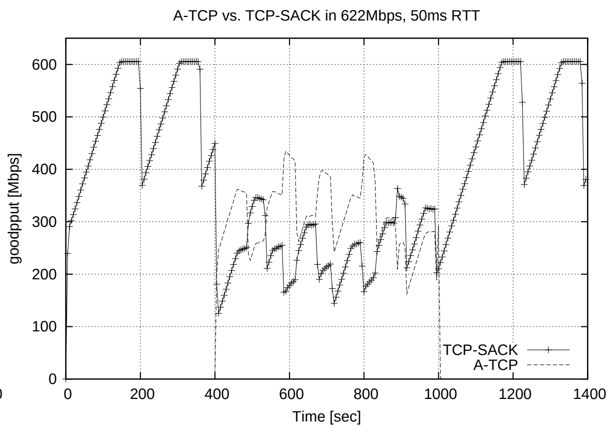
Fig. 39. TCP-SACK vs. (a) H-TCP (b) BIC-TCP

high-speed TCP proposals such as BIC-TCP, H-TCP, and TCP-Africa. For BIC-TCP and H-TCP, we use ns2 codes from the authors without changing their default parameter settings. For this experiment Node 1 in Figure 36 opens a TCP-SACK connection for 1400 seconds to Node 3. Node 0 joins the experiment by opening a high-speed TCP flow to Node 2 from 400 sec to 1000 sec. In Figure 39 and Figure 40 we show the results of the experiments.

From the experimental results of H-TCP and BIC-TCP shown in Figure 39(a) and 39(b) we can see that when a BIC-TCP or H-TCP flow joins the test-bed network, they dominate the network resources and would not allow the competing TCP-SACK flow to consume fair amount of the bandwidth. As a result, the competing TCP-SACK flow achieves only a small portion of the bottleneck bandwidth. The experimental results of TCP-Africa shown in Figure 40(a) show that TCP-Africa is also significantly unfair to the competing TCP-SACK although it is better than H-TCP and BIC-TCP. In contrast, as Figure 40(b) shows, when an A-TCP flow joins the test-bed network, the A-TCP flow does allow the competing TCP-SACK flow much higher bandwidth than it could achieve against other high-speed TCP proposals.



(a)



(b)

Fig. 40. TCP-SACK vs. (a) TCP-Africa (b) A-TCP

In Figure 41 we also show another simulation result that illustrates TCP-friendliness of A-TCP when A-TCP and TCP-SACK have different access link bandwidth. For this test we change the access link bandwidth for the competing standard TCP flow to 100Mbps. The bandwidths of all other links remain at 622Mbps. The achieved throughput by the A-TCP flow and the TCP-SACK flow as well as the changes in the congestion window size of the two flows are shown in Figure 41. The simulation results in the figure show that the competing TCP-SACK flow achieves almost full bandwidth utilization of its access link. A-TCP does not steal bandwidth from the standard TCP flow with the smaller bandwidth access link. From the figure we can see that A-TCP adapts to the change in the available bandwidth when the competing TCP-SACK starts to send traffic at 400 sec. After the tear down of the competing TCP-SACK flow at 1000 sec, A-TCP tries to adapt to the newly available bandwidth. Note that, however, it took one congestion epoch for A-TCP to reclaim the available bandwidth because the competition detection mechanism in A-TCP holds A-TCP to be non-aggressive for the period because of the detection of competition level change.

To test how well or how quickly A-TCP responds to the changes of network

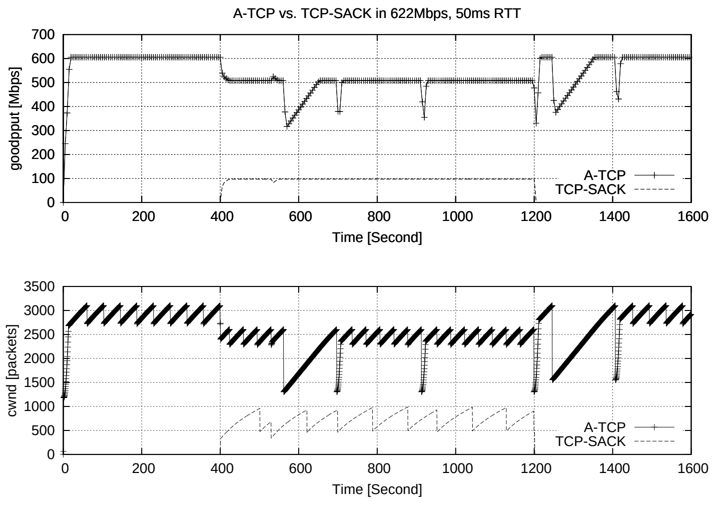


Fig. 41. A-TCP vs. TCP-SACK from 100Mbps Link

situations, we now use a non-responsive UDP connection as the cross traffic from Node 1 of Figure 36. In this simulation, the bandwidths of all links are set to 622Mbps and the propagation delay between senders and receivers remain 50msec. First, Node 0 opens an A-TCP flow to Node 2 for 1600 seconds. At 400 sec Node 1 opens a UDP connection of 300Mbps to Node 3 for the next 700 seconds. In Figure 42 we show the achieved goodput by the A-TCP flow and the UDP flow. We also show the changes of the congestion window size of the A-TCP flow in the figure. From Figure 42 we can see that the A-TCP flow from Node 0 quickly adapts to the new network situation caused by the non-responsive 300Mbps UDP cross traffic generated by Node 1 from 400 sec to 1100 sec. A-TCP adapts to the changes in the network caused by the start and cease of the non-responsive UDP connection. Except the periods for A-TCP to adapt to the changes in the network, the goodput achieved by the A-TCP flow remains stable.

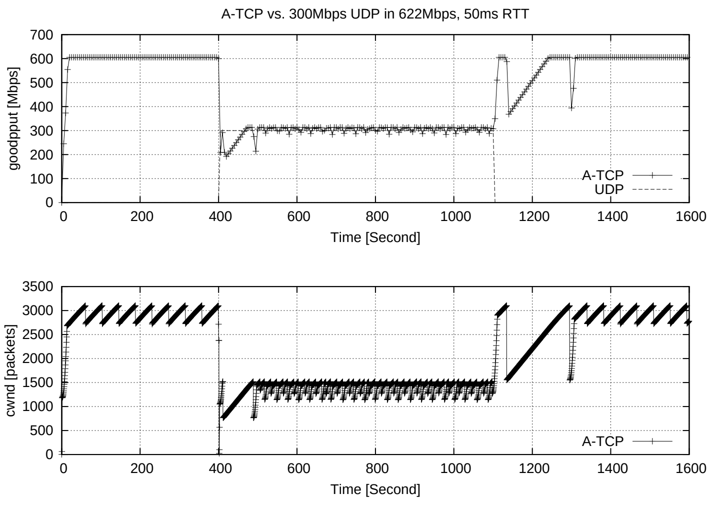


Fig. 42. A-TCP vs. 300Mbps UDP

I. Summary

In this chapter we proposed a new TCP congestion control scheme called Adaptive TCP (A-TCP) for effective and fair utilization of high bandwidth-delay product networks. A-TCP automatically adapts its strength to control its aggressiveness according to network status through accurate estimation of network situations using high resolution round-trip times and a competition detection mechanism. A-TCP increases its strength k to achieve high utilization of available, or unclaimed bandwidth, while it holds the strength $k = 1$ and becomes a standard TCP when it detects competition with other flows. As a result, A-TCP achieves both effective and fair utilization of network resources. Further, A-TCP is a purely sender-based TCP because it does not require any special support from networks or modification of receivers.

CHAPTER V

ADAPTIVE AGGREGATED AGGRESSIVENESS CONTROL ON GROUPS OF TCP FLOWS

A. Introduction

Opening parallel TCP flows (i.e. multiple concurrent TCP connections to the same destination) is an easy way to effectively utilize available bandwidth because it does not require modification of TCP itself. With parallel TCP flows, users can work around the throughput limitations for single-flow TCP in high bandwidth-delay product networks. However, it is difficult to determine the number of parallel TCP flows needed to effectively utilize available bandwidth without unduly hurting competing single TCP flows, especially in dynamically changing networks. There have been several efforts to achieve high performance using parallel TCP flows while mitigating the unfairness of parallel TCP flows. The Congestion Management (CM) architecture [28], COCOON [29], and Fractional/Combined TCP flows [18] are some examples. In this chapter, however, we show through extensive experimentation that improving performance of parallel TCP flows while maintaining fairness (or TCP-friendliness) to single TCP flows is not easy to achieve.

We proposed TCP-P earlier, which allows adjusting the aggressiveness of a group of parallel TCP flows in a controllable fashion, using a runtime adjustable *strength* parameter k. The "strength" k is a scalar value that describes how big the group (in number of flows) is perceived by other TCP flows. TCP-P in effect controls the aggressiveness of parallel TCP flows by separating the *aggressiveness* of parallel TCP flows from the *number* of parallel TCP flows that can be opened by users almost without limits. However, deciding on an adequate value for the strength k for a group of parallel TCP-P flows in a particular network situation still remains. The system operator therefore is left with the task to explicitly manage the aggressiveness of TCP-P flow groups. In this chapter we present an approach to have the system adaptively determine a good value for the aggressiveness in a management free fashion.

Most of recent high-speed TCP proposals, such as HS-TCP, STCP, HTCP, and BIC-TCP, have shown to exhibit serious fairness problems with respect to standard TCP. A-TCP is one of the recent high-speed TCP proposals. Differently from other high-speed TCP proposals, however, A-TCP does not cause fairness problem with competing single TCP flows because it automatically adapts its aggressiveness to network situations using a competition detection mechanism. A-TCP increases its strength (or aggressiveness) k to achieve high utilization of available, or unclaimed bandwidth, while holding its strength at one when it detects competition with other flows. As a result, A-TCP achieves both effective and fair utilization of network resources. However, if multiple A-TCP flows compose a group of parallel TCP flows, the aggregated aggressiveness of the parallel A-TCP flows will be at least that of the parallel standard TCP flows because the minimum aggressiveness of A-TCP is the same as that of standard TCP. Given the relatively poor fairness provided by other high-speed TCP schemes for the single-flow case, we expect them to perform even worse in the case of multiple parallel flows.

In this chapter, we propose a new parallel-TCP scheme, called Adaptive TCP-P (A-TCP-P), whose effectiveness and fairness are not affected by the number of the parallel TCP flows. A-TCP-P realizes effective and fair parallel TCP by combining TCP-P with the competition detection mechanism in A-TCP. When the competition detection mechanism detects the changes in the competition level along its path, the parallel A-TCP-P flows become TCP-friendly by holding the strength *of the group* at $k = 1$. When $k = 1$, A-TCP-P regulates the aggregated aggressiveness of parallel

TCP flows to be comparable to that of single, standard TCP flow using the aggregated aggressiveness control mechanism in TCP-P. (In this case, every flow in the group is less aggressive than a single standard TCP flow.) Otherwise, A-TCP-P allows the parallel TCP flows to increase the strength of the group to effectively utilize available bandwidth. In addition, A-TCP-P reacts when it detects imminent packet loss and holds the group strength of the parallel TCP flows at $k = 1$ to reduce upcoming packet losses by estimating queuing delay variations.

The remainder of this chapter is organized as follows: Section B proposes Adaptive TCP-P. Section C experimentally shows that A-TCP-P effectively and fairly utilizes available bandwidth. We also compare its performance with other parallel TCP schemes. In Section D we compare the fairness of parallel high-speed TCP flows with that of parallel A-TCP-P flows. Section E summarizes this chapter.

B. Adaptive Aggregated Aggressiveness Control

Fairness in the context described here is based on the ability to detect and adequately respond to competition for resources in the network. We previously developed a competition detection and response scheme for the TCP single-flow case, and implemented it in form of the A-TCP protocol. In this chapter we describe our experiences of extending competition detection and response to the parallel-flow case. To adopt A-TCP-style competition detection into TCP-P, we extend the meaning and usage of the variables used in A-TCP to be group-wide. The competition detector used here is based on the observation that the queue build-up rate is higher the more flows are present along the path. Queue build-up in turn can be measured by monitoring the changes of *g_knee_cwnd*.

g_knee_cwnd is the aggregated congestion window size of all the member TCP

Algorithm 5 Parameters and initialization of variables of A-TCP-P

```
Parameters:
  $\alpha \leftarrow 10$; //waiting time after the last checking
  $\delta \leftarrow 10$; //waiting time after packet loss
  $\beta \leftarrow 0.1$; //margin of error

  $\gamma \leftarrow 0.2$; //reference queuing delay level

Variables:
  $k \leftarrow 1$; //strength of a group of A-TCP-P flows
  $g\_knee\_cwnd \leftarrow 0$; //group congestion window at a knee
  $g\_knee\_cwnd\_ref \leftarrow 0$; //reference $g\_knee\_cwnd$
  $g\_knee\_k \leftarrow 0$; //group strength $k$ at a knee
  $g\_check\_knee \leftarrow 0$; //check $knee\_cwnd$ when it is 1
  $g\_low\_delay \leftarrow 0$; //low congestion when it is 1
  $gsrtt \leftarrow 0$; //group-wide smoothed round-trip time
  $gsrtt\_max \leftarrow 0$; //maximum $gsrtt$
  $gsrtt\_min \leftarrow$ 0xffffffff; //minimum $gsrtt$
  $g\_modify\_time \leftarrow$ now; //last time when $k$ was modified

  $g\_wait\_time \leftarrow 0$; //waiting time for the next checking.
```

flows when *gsrtt* passes over the reference queuing delay level shown in Line 3 of Algorithm 6, which is based on the group-wide round-trip delay information such as $gsrtt\_min$ and $gsrtt\_max$. A-TCP-P determines if there are competing TCP flows by checking whether $g\_knee\_cwnd\_ref$ is equal to g_knee_cwnd or not. When packet loss is detected by any member A-TCP-P flows and if $g\_knee\_cwnd\_ref$ is different from g_knee_cwnd, the flow reduces total congestion windows size of the group by half and the parallel A-TCP-P flows will behave as if a standard TCP flow during the next congestion epoch. The next congestion epoch ends when any A-TCP-P flow experiences packet loss. Otherwise, the group will enter aggressive mode: *i*) The total congestion window size of the group (g_cwnd) of the parallel TCP flows will be reduced by $\frac{g\_cwnd}{2\sqrt{g\_knee\_k}}$ instead of halving (g_knee_k is the strength of the parallel A-TCP-P flows when the estimated queuing delay becomes bigger than the reference queuing delay level). *ii*) After the reduction, the strength k of the group will increase

Algorithm 6 With a new ACK in any member A-TCP-P flow of a group

```
1: update $gsrtt$, $gsrtt\_min$, and $gsrtt\_max$;
2: if now $- g\_modify\_time > g\_wait\_time$ then //check if it is time to work.
3:   if $gsrtt \leq gsrtt\_min + \gamma \cdot (gsrtt\_max - gsrtt\_min)$ then
4:     if $(g\_check\_knee = 1)$ and $(g\_knee\_cwnd\_ref = g\_knee\_cwnd)$ then
5:       $k \leftarrow k + 1$; //increase $k$ until a knee is reached.
6:     end if
7:     $g\_low\_delay \leftarrow 1$;
8:   else
9:     if $g\_low\_delay = 1$ then //queuing delay becomes bigger than the threshold.
10:      $g\_knee\_cwnd \leftarrow$ current $g\_cwnd$;
11:      if $g\_knee\_cwnd\_ref = 0$ then
12:        $g\_knee\_cwnd\_ref \leftarrow g\_knee\_cwnd$;
13:      else if $((1-\beta) \cdot g\_knee\_cwnd\_ref \leq g\_knee\_cwnd \leq (1+\beta) \cdot g\_knee\_cwnd\_ref)$
         then
14:        $g\_knee\_cwnd\_ref \leftarrow g\_knee\_cwnd$;
15:      end if
16:      if $g\_check\_knee = 1$ then
17:        $g\_knee\_k \leftarrow k$;
18:      end if
19:      $g\_check\_knee \leftarrow 0$; //stop aggressive behavior (i.e., increasing $k$).
20:      $g\_low\_delay \leftarrow 0$;
21:    end if
22:    $k \leftarrow 1$;
23:  end if
24:  $g\_modify\_time \leftarrow$ now;
25:  $g\_wait\_time \leftarrow \alpha \cdot gsrtt\_min$; //set waiting time until the next checking.
26: end if
```

quickly if the estimated queuing delay is less than the reference queuing delay level.

The parameters and variables used for A-TCP-P are shown in Algorithm 5 and details of the mechanics for adaptive aggregated aggressiveness control are shown in Algorithm 6 and Algorithm 7. Because the usage of all the variables shown in Algorithm 5 is group-wide, any A-TCP-P flow can access and modify the variables of its own group and the changes of these variables affect the behavior of the entire parallel TCP flows in the group. For example, *gsrtt* is the smoothed round-trip time based on the round-trip time measured by all the parallel TCP flows. *gsrtt*

Algorithm 7 With packet loss in any member A-TCP-P flow of a group

```
1: if $first = 1$ then //ignore the first packet loss caused by initial slow-start.
2:     $k \leftarrow 1$; $g\_knee\_k \leftarrow 1$; $g\_knee\_cwnd \leftarrow 0$; $g\_knee\_cwnd\_ref \leftarrow 0$; $first \leftarrow 0$;
3: else if $g\_knee\_cwnd\_ref = g\_knee\_cwnd$ then //enter aggressive mode.
4:     $k \leftarrow \max(\sqrt{g\_knee\_k}, 1)$
5:     $gsrtt\_max \leftarrow gsrtt\_min$
6: else //enter non-aggressive mode.
7:     $k \leftarrow 1$
8:     $g\_knee\_k \leftarrow 1$
9:     $g\_knee\_cwnd\_ref \leftarrow 0$
10: end if
11: $g\_check\_knee \leftarrow 1$;
12: $g\_low\_delay \leftarrow 0$;
13: $g\_modify\_time \leftarrow$ now;
14: $g\_wait\_time \leftarrow \delta \cdot gsrtt\_min$; //set waiting time until the next checking.
```

is updated when any A-TCP-P flow updates its own smoothed round-trip time *srtt*. Also the maximum and minimum of *gsrtt*, i.e. *gsrtt_max* and *gsrtt_min*, are updated whenever new *gsrtt* is bigger than *gsrtt_max* or smaller than *gsrtt_min*. The way to update *gsrtt* of a group of parallel A-TCP-P flows is similar to updating *srtt* in each flow:

$$gsrtt(t+) = (1 - w) * gsrtt(t) + w * rtt(t). \tag{5.1}$$

The parameter w assigns a weight to the new round-trip time measurement (rtt) and is set to $\frac{1}{8}$. In this implementation, all the variables are embedded in a group information management structure and pointers are used for member A-TCP-P flows to access these variables.

In the following sections we describe a series of experiments where we evaluate the performance of A-TCP-P (in terms of effectiveness and fairness) and explain how A-TCP-P adaptively adjusts its group strength according to network situations.

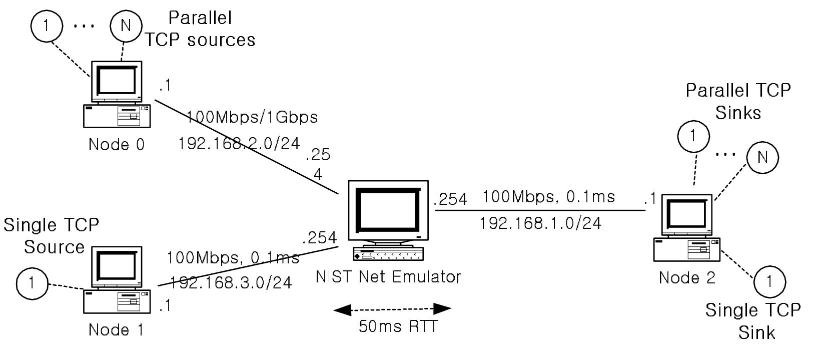


Fig. 43. Experiment Network

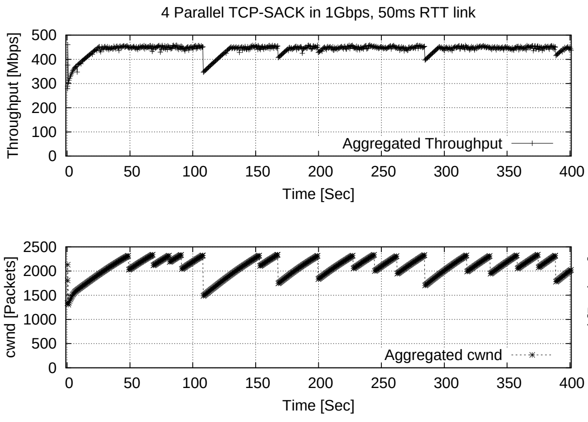
C. Evaluation

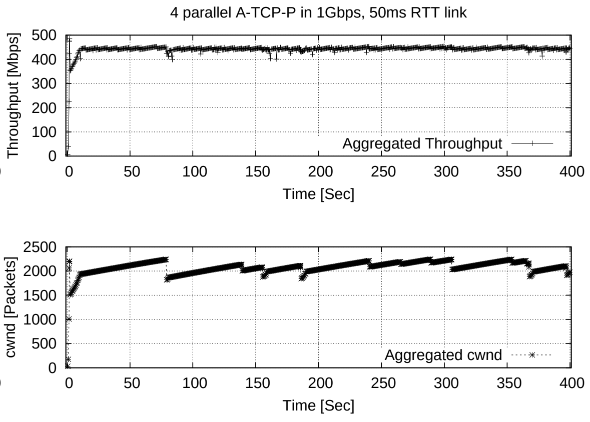
1. Experiment Setup

For all the experiments in this chapter, we use the test-bed shown in Figure 43. In the test-bed, we use NIST Net Emulator [49] to emulate link delays in the Internet. All the end-host nodes and the NIST Net Emulator are running on Linux PCs. The PCs we used for these experiments are Pentium 4 machines with 10/100Mbps Fast Ethernet or Gigabit Ethernet network interface cards. In a sender Node 0, we installed Linux kernel 2.4.20-8 and modified Linux kernels for A-TCP-P. The other machines run on unmodified Linux kernel 2.4.20-8. For traffic generation and throughput measurement we use `iperf` Version 1.7.0 [50]. By default, Linux TCP behavior is based on TCP-SACK [1].

2. Effectiveness of A-TCP-P

First, we evaluate the effectiveness of A-TCP-P in using the available bandwidth. To do this, we use a 1 Gbps link between Sender Node 0 and the NIST Net Emulator node to 1Gbps. To illustrate the behavioral difference between parallel A-TCP-P and unmodified parallel TCP, we let Node 0 open either four parallel unmodified





(a) 4 Parallel TCP-SACK (b) 4 Parallel A-TCP-P

Fig. 44. 4 Parallel (a) TCP-SACK (b) A-TCP-P Flows in 1Gbps, 50msec RTT Link

TCP-SACK flows or four parallel A-TCP-P flows to the NIST Net node (NIST Net emulator works as a receiver for this experiment). The round-trip propagation delay between Node 0 and NIST Net Emulator is set to 50msec by the NIST Net Emulator. Although the physical link technology between these two nodes is optical Gigabit Ethernet, the hardware performance (533MHz front-side bus speed, 32bit PCI bus) of the sender and receiver machines in our test-bed limited the maximum achievable throughput to about 440Mbps.

In Figure 44(a) we show the aggregated throughput achieved by Node 0 when it opens four parallel TCP-SACK flows to the NIST Net node and the changes of the aggregated congestion window size of the four parallel TCP-SACK flows during the experiment. In comparison, Figure 44(b) shows the aggregated throughput achieved by Node 0 when it opens four parallel A-TCP-P flows to the NIST Net node and the changes of the aggregated congestion window size of the four parallel A-TCP-P flows. By comparing the two figures we can see that A-TCP-P can utilize the available bandwidth more effectively than standard TCP: A-TCP-P flows reach the maximum throughput at about 10 sec compared to about 50 sec of unmodified four TCP-SACK

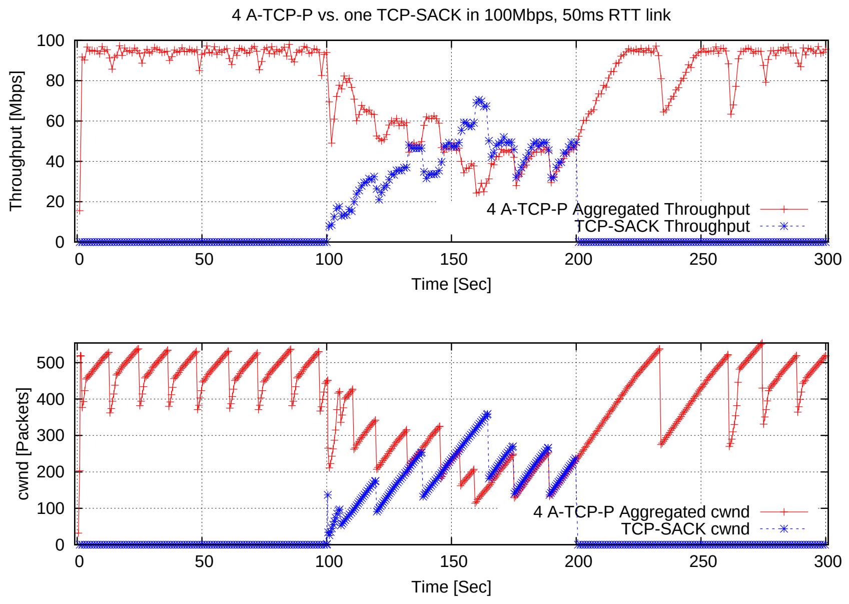


Fig. 45. 4 Parallel A-TCP-P vs. One TCP-SACK in 100Mbps, 50msec RTT Network

flows. Further, the figures also show that A-TCP-P results in less packet loss than the unmodified parallel TCP by reducing the group's aggressiveness before packet loss happens.

3. Fairness of A-TCP-P

To illustrate how A-TCP-P behaves in a dynamic network situation, we let four parallel A-TCP-P flows from Node 0 compete against a single TCP-SACK flow from the other sender node (Node 1). All the links in the test-bed are now set to 100Mbps and RTT between the sender nodes and the receiver node is set to 50msec using the NIST Net Emulator. Node 0 now opens four parallel A-TCP-P flows to Receiver Node 2 for 300 seconds. After 100 sec, Node 1 opens a TCP-SACK flow to Node 2 for the next 100 seconds.

In Figure 45 we show the aggregated throughput achieved by the four parallel A-TCP-P flows and the throughput achieved by the competing single TCP-SACK

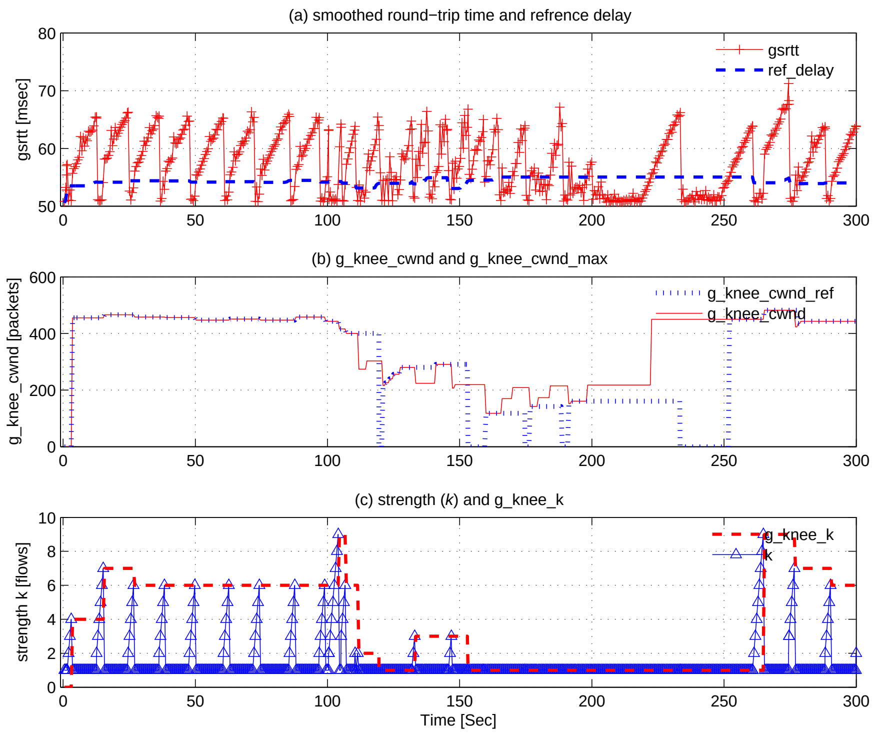


Fig. 46. Variables Inside A-TCP-P

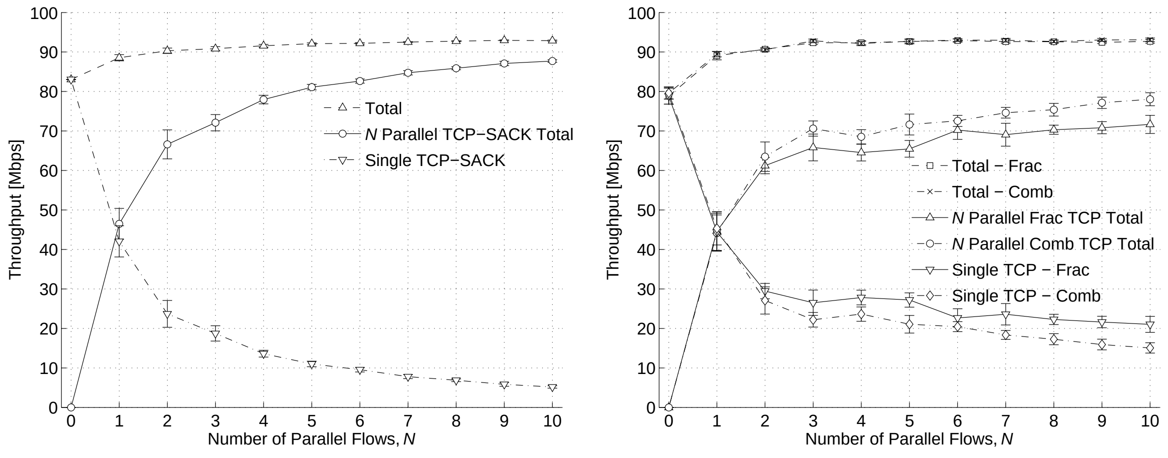
flow. The figure also shows the changes of the aggregated congestion window size of the four parallel A-TCP-P flows and the changes of the congestion window of the competing TCP-SACK flows. From the experimental results in the figure we can see that A-TCP-P achieves high throughput when it is alone in the test-bed during the first 100 and the last 50 seconds. Further, the results also show that the four A-TCP-P flows and the competing TCP-SACK flow fairly share the bandwidth during the overlapping 100 seconds.

To explain the behavior of A-TCP-P in this experiment, we also show the changes of the variables inside the parallel A-TCP-P flows, such as *gsrtt* and strength k, in Figure 46. The *ref_delay* line in the Figure 46(a) represents the reference queuing delay level. Figure 46(b) tracks *g_knee_cwnd* and *g_knee_cwnd_ref* used to detect competition. Figure 46(c) show the changes of k and *knee_k*. By comparing Figure 45 and Figure 46(c), we can see that the initial quick increase of aggregated congestion window of parallel A-TCP-P flows is caused by the rapid increase of the group strength k. This aggressive behavior of A-TCP-P is repeated after each packet loss until the competing TCP-SACK flow from Node 1 joins the test-bed network at 100 sec because g_knee_cwnd remains equal to $g\_knee\_cwnd\_ref$ as shown in Figure 46(b).

During the overlapping 100 seconds, as Figure 46(b) shows, g_knee_cwnd and $g\_knee\_cwnd\_ref$ variables of A-TCP-P are set to have different values from each other repeatedly by the competition detection mechanism. As a result, A-TCP-P is forced to be in non-aggressive mode repeatedly. This example clearly illustrates how the competition detection mechanism in A-TCP-P drives parallel A-TCP-P flows to fairly share the bandwidth even with one standard TCP flow. After the competing TCP-SACK flow leaves the network at 200 sec, g_knee_cwnd and $g\_knee\_cwnd\_ref$ are set to have the same value again at about 250 sec. The two variables remain equal after that point and A-TCP-P stays in aggressive mode.

4. Performance Comparison with Other Parallel TCP Schemes

In this section, we first compare A-TCP-P with other parallel TCP schemes such as Fractional/Combined TCP, COCOON, CM, and TCP-P. Node 0 now opens from zero to 10 parallel TCP flows for each scheme to the receiver Node 2 for 100 seconds while the other sender Node 1 opens one standard TCP-SACK flow to the same receiver for the same period. When the number of parallel TCP flows from Node 0 is zero, Node 0 does not open any TCP flow to Node 2, so that only the single TCP-SACK flow from Node 1 consumes all the network resources without competition. By increasing the number of parallel TCP flows from Node 0, we measure how much bandwidth Node 0 gains at the cost of Node 1. In all the following experiments, every experiment was repeated 10 times with a 5 sec waiting time after each experiment to get an average



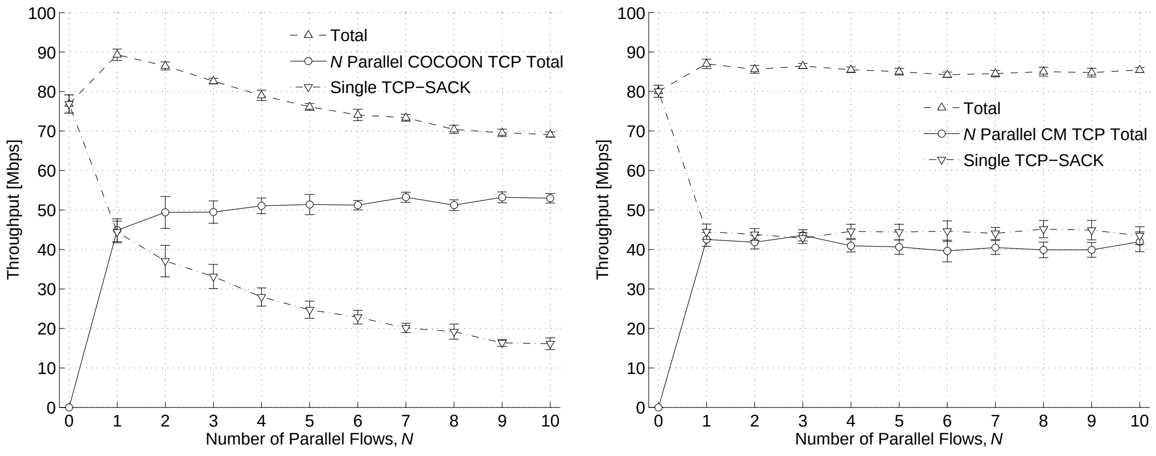
(a) TCP-SACK (b) Fractional/Combined TCP

Fig. 47. TCP-SACK vs. Parallel (a) TCP-SACK (b) Frac./Comb. TCP

value. Error bars in the figures represent 95% Confidence Interval of the data.

For comparison we first show how much bandwidth unmodified parallel TCP *steals* from the competing standard TCP flow. Figure 47(a) shows the experiment results with an increasing number of unmodified parallel TCP flows from Node 0. From the figure we can clearly see that, by opening more TCP connections to the same destination, the unmodified parallel TCP flows achieve higher throughput by stealing bandwidth from the competing TCP-SACK flow of Node 1.

Fractional and *Combined* TCP schemes [18] were proposed to mitigate unfairness of unmodified parallel TCP flows. Fractional TCP scheme reduces the increase speed of the congestion window sizes of parallel TCP flows' according to the number of parallel flows. The Combined TCP scheme suggests to include one unmodified TCP to a group of Fractional TCP flows to compensate for the conservativeness of Fractional TCP flows. We used our implementation of Fractional and Combined TCP in Node 0 based on the authors' paper [18]. Figure 47(b) shows that Fractional and Combined



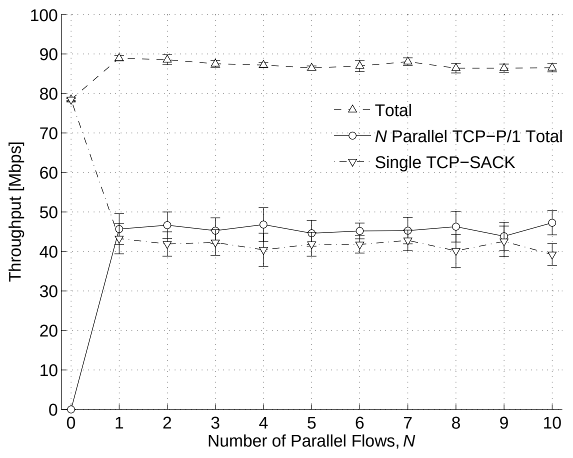
(a) COCOON TCP (b) CM TCP

Fig. 48. TCP-SACK vs. Parallel (a) COCOON TCP (b) CM TCP

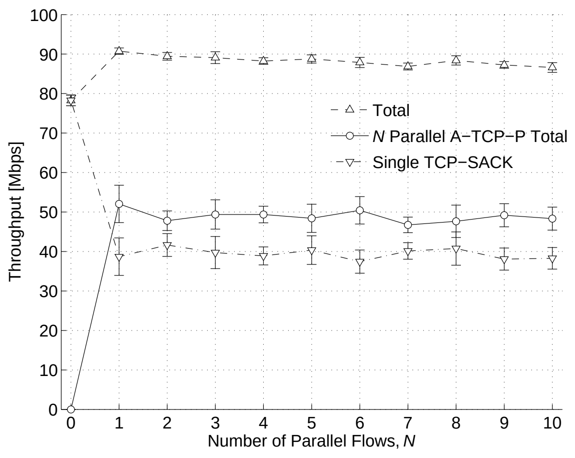
TCP flows are less aggressive than unmodified TCP flows, while Combined TCP flows show higher throughput than Fractional TCP, at the cost of more throughput decrease in the competing TCP flow. As we can see from the figures, although the unfairness is reduced compared to that of unmodified parallel TCP flows, this scheme still steals significant amounts of bandwidth from the competing TCP flow.

COCOON [29] employs a coordinated congestion control to a group of TCP flows. Specifically, when a flow in the group experiences congestion, it adjusts the congestion windows of member TCP flows whose outstanding packet sizes are larger than the congestion window size of the TCP flow. To test COCOON we use modified kernel 2.4.22 with the kernel patch from the authors. Figure 48(a) shows that COCOON also becomes unfair when it opens more than one TCP connection to the receiver. Furthermore, the experimental results also show that parallel COCOON TCP flows do not effectively utilize the bandwidth taken from the competing TCP-SAck flow.

Congestion Management (CM) architecture [28] uses single congestion window



(a) TCP-P/1



(b) A-TCP-P

Fig. 49. TCP-SACK vs. Parallel (a) TCP-P/1 (b) A-TCP-P

for a group of parallel TCP flows. We used modified Linux kernel 2.2.18 from the authors in Node 0[1]. We also used unmodified kernel 2.2.18 in Node 1 for fair comparison. Figure 48(b) shows the experimental results with parallel CM TCP flows. The experimental results in the figure show that the aggregated throughput of parallel CM TCP flows from Node 0 and the throughput of the competing TCP flow from Node 1 are not much changed even if Node 0 opens 10 CM TCP flows, so that CM can be considered to be a fair parallel scheme.

Differently from Fractional TCP, which controls only increase behavior, and COCOON, which controls only decrease behavior, TCP-P controls both. Parallel TCP-P flows with strength k (i.e., TCP-P/k) appear to other TCP flows like k separate TCP flows. Figure 49(a) shows experimental results with TCP-P with strength $k = 1$ (i.e. TCP-P/1). From the figure we can see that a group of parallel TCP-P flows with

[1]The CM kernel implementation had some bugs that cause TCP hang. We disabled flood of kernel debugging messages and increased the socket buffer size of TCP to 2MB from the default 64 kB.

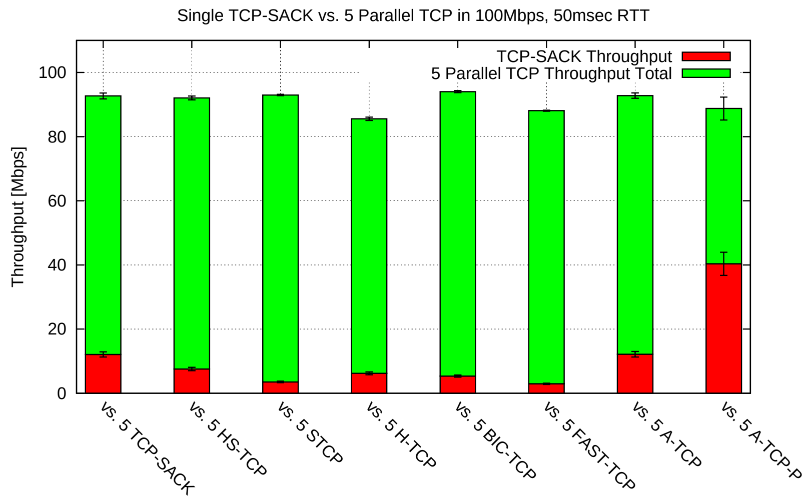


Fig. 50. TCP-SACK vs. 5 Parallel TCP of Various Types

strength $k = 1$ has similar fairness property to that of CM even though TCP-P allows each TCP-P flow to maintain its own congestion window.

Figure 49(b) shows the experimental results with different number of parallel A-TCP-P flows. Although A-TCP-P is slightly more aggressive than TCP-P and CM, the experimental results indicate that parallel A-TCP-P flows also maintain good TCP-friendliness to the competing TCP-SACK flow. However, note that the aggressiveness of TCP-P/1 or CM is fixed to that of a single TCP, so that their effectiveness can not be better than that of single TCP. In contrast, A-TCP-P changes the strength of the group automatically for the effective utilization of the high bandwidth-delay product networks whenever it does not detect competition.

D. Parallel High-speed TCP

Although the fairness of TCP proposals are considered as an important factor for the evaluation of the TCP proposals, it is usually not questioned what will happen if many of such TCP flows form parallel TCP flows and compete against standard, single TCP flows. Since there is almost no limit to the number of connections a user can open concurrently, users may easily achieve more than its fair share by simply opening multiple of such TCP flows regardless of how *fairly* the TCP was designed. Thus, we believe that parallel-level performance of TCP proposals should be considered as an important criteria during evaluation. In this section we investigate the parallel-level fairness of recently proposed high-speed TCP schemes when multiple flows of each TCP proposal compose parallel high-speed TCP flows.

For this, Node 0 in our test-bed now opens five parallel TCP flows of each high-speed TCP proposal including HS-TCP, STCP, H-TCP, BIC-TCP, A-TCP, and FAST-TCP to Node 2 for 100 seconds. The other sender Node 1 opens one TCP-SACK flow to Node 2 for the same period. The histogram in Figure 50 shows the experimental results. The bar length for each high-speed TCP proposal in the figure represents the total throughput achieved by the five parallel high-speed TCP flows. In the figure, for comparison, we also show the experimental results with 5 parallel TCP-SACK flows.

The experimental results in the figure clearly show that parallel TCP flows of high-speed TCP proposals except for A-TCP-P are seriously unfair against the competing TCP-SACK flow. The throughput achieved by the single TCP-SAck flow when it competes against five parallel A-TCP-P flows is significantly higher than other cases. Interestingly, the aggregate throughput achieved by five parallel FAST-TCP flows is not much different from that of other parallel high-speed TCP proposals although a single FAST-TCP flow showed weaker aggressiveness than that of a standard TCP flow. The results demonstrate that parallel-level performance of TCP proposals can be quite different from their per-flow performance unless their parallel-level performance is considered in its design aspect as A-TCP-P.

E. Summary

While parallel TCP is an effective means to effectively make use of network resources, in particular in high bandwidth-delay settings, balancing the number of flows against fair use of the networks is difficult, in particular in dynamically changing network environments. In this chapter, we propose a method to decouple the number of flows from the aggregated aggressiveness of the group of flows in a self managing fashion. Based on this scheme, we implemented A-TCP-P, which adequately adjusts the group strength k of parallel TCP flows to achieve high utilization of available bandwidth while maintaining TCP-friendliness against single TCP flows. A-TCP-P uses a competition detection mechanism to adaptively adjust aggregated aggressiveness of parallel TCP according to network situations. As a result, we provided a parallel TCP scheme for effective and fair utilization of network resources independently from the number of parallel flows.

CHAPTER VI

CONCLUSION

In this work, we have suggested several congestion control schemes for single and parallel TCP flows.

The collaborative congestion control scheme in Chapter II has shown the usefulness and benefit of collaboration among parallel TCP flows by sharing dynamic congestion information. By comparing the experimental results of the scheme with other schemes such as ECN-based TCP congestion control, we show that the collaborative congestion control has performance that is comparable (in terms of packet loss, delay, and jitter) with architecturally significantly more expensive schemes.

In Chapter III, we have suggested a new congestion control scheme for parallel TCP flows, called TCP-P, which controls aggregated aggressiveness of parallel TCP flows by regulating their total aggressiveness (or unfairness) to be comparable to that of 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 through appropriate manipulations of increase and decrease behavior of the congestion windows of the TCP flows in the group. The experimental results with the implementation of TCP-P in Linux kernel show that the proposed scheme effectively controls the aggregated aggressiveness of parallel TCP flows.

In Chapter IV, we focus on the fairness (or TCP-friendliness) issues of recently proposed high-speed TCP proposals for high bandwidth-delay product networks. We have suggested a new TCP congestion control protocol, called Adaptive TCP (A-TCP), which adaptively controls its aggressiveness according to network situations using a competition detection mechanism. If no competing flows are detected, A-TCP increases its aggressiveness in order to effectively utilize the network. Otherwise, it becomes a standard TCP flow for the fair sharing of the network with other flows during the next congestion epoch. As a result, A-TCP effectively utilizes high bandwidth-delay product networks while maintaining TCP-friendliness to standard TCP flows. Experimental results with the implementation of A-TCP in Linux kernel and ns2 show that A-TCP indeed achieves high utilization with better TCP-friendliness than other high-speed TCP proposals.

In Chapter V, we have proposed a congestion control scheme whose parallel-level effectiveness and fairness are independent from the the number of parallel connections. For this we combined TCP-P's aggregated aggressiveness control mechanism for parallel TCP flows with A-TCP's adaptive aggressiveness control mechanism for single TCP flows. We have shown that the proposed parallel TCP scheme called adaptive aggregated aggressiveness control (A-TCP-P) utilizes available bandwidth effectively and fairly by adaptively adjusting the aggregated aggressiveness of parallel TCP flows.

As a result, we have shown that we can realize effective and fair congestion control schemes for both single and parallel TCP flows by adaptively adjusting their aggressiveness according to network situations in self-managed way. Further, the proposed TCP congestion control schemes do not require special support from network equipment or modification of TCP receivers because all the proposed schemes work based on the information which can be gathered by the sender-side. Thus, we believe that the proposed schemes can be applied in a variety of network settings without causing deployability issues.

REFERENCES

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

[2] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[3] D. Katabi, M. Handley, and C. Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks," in *Proc. ACM SIGCOMM*, Aug. 2002, pp. 89–102.

[4] R. Braden ,D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: An Overview," *IETF Internet RFC 1633*, June 1994.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," *IETF Internet RFC 2475*, December 1998.

[6] C. Dovrolis and P. Ramanathan, "A Case for Relative Differentiated Services and the Proportional Differentiation Model," *IEEE Network*, vol. 13, no. 5, pp. 26–34, September 1999.

[7] A. Leon-Garcia and I. Widjaja, *Communication Networks*, New York, NY: McGraw-Hill, 2000.

[8] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," *IETF Internet RFC 2582*, April 1999.

[9] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgement Options," *IETF Internet RFC 2018*, Oct. 1996.

[10] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," in *Proc. ACM SIGCOMM*, Aug. 1994, pp. 24–35.

[11] C. Jin, D. X. Wei, and S. H. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2490–2501.

[12] S. Floyd, "HighSpeed TCP for Large Congestion Windows," *IETF Internet RFC 3649*, Dec. 2003.

[13] T. Kelly, "Scalable TCP: Improving Performance in Highspeed Wide Area Networks," *ACM CCR*, vol. 32, no. 2, pp. 83–91, Apr. 2003.

[14] D.J. Leith and R. Shorten, "H-TCP: TCP for High-Speed Long Distance Networks," in *Proc. Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2004, http://dsd.lbl.gov/DIDC/PFLDnet2004/papers/Leith.pdf, accessed Dec. 27, 2005.

[15] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control for Fast Long-Distance Network," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2514–2524.

[16] S. Bhandarkar, S. Jain, and A. L. N. Reddy, "Improving TCP Performance in High Bandwidth High RTT Links Using Layered Congestion Control," in *Proc. Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2005, http://www.globus.org/alliance/publications/papers/dpss_and_gridftp.pdf, accessed on Dec. 27, 2005.

[17] S. Floyd and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet," *IEEE/ACM Transactions on Networking*, vol. 7, no. 4, pp.

458–472, Aug. 1999.

[18] T. Hacker, B. Noble, and B. Athey, "Improving Throughput and Maintaining Fairness Using Parallel TCP," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2480–2489.

[19] Akamai Technologies, Inc, "Akamai: The Trusted Choice for Online Business," http://www.akamai.com, accessed on Dec. 27, 2005.

[20] P. Karbhari, M. Ammar and E. Zegura, "Optimizing End-to-End Throughput for Data Transfers on an Overlay-TCP Path," in *Proc. IFIP Networking*, May 2005, pp. 943–955.

[21] J. Crowcroft and P. Oechslin, "Differentiated End-to-End Internet Services Using a Weighted Proportionally Fair Sharing TCP," *ACM CCR*, vol. 28, no. 3, pp. 53–67, Jul. 1998.

[22] R. Jain, "A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks," *ACM CCR*, vol. 19, no. 5, pp. 56–71, Oct. 1989.

[23] Z. Wang and J. Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Taeho BSD TCP Congestion Control Algorithms," *ACM CCR*, vol. 22, no. 2, pp. 9–16, Apr. 1999.

[24] J. Martin, A. Nilsson, and I. Rhee, "The Incremental Deployability of RTT-based Congestion Avoidance for High Speed TCP Internet Connections," in *Proc. ACM SIGMETRICS*, Jun. 2000, pp. 134–144.

[25] D. Maltz and P. Bhagwat, "TCP Splicing for Application Layer Proxy Performance," Tech. Rep. RC21139, IBM Research, Mar. 1998.

[26] R. Braden, "T/TCP - TCP Extensions for Transactions, Functional Specification," *IETF Internet RFC 1644*, July 1994.

[27] L. Eggert, J. Heidemann, and J. Touch, "Effects of Ensemble-TCP," *ACM CCR*, vol. 30, no. 1, pp. 15–29, Jan. 2000.

[28] H. Balakrishnan, H. Rahul, and S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts," in *Proc. ACM SIGCOMM*, Sep. 1999, pp. 175–187.

[29] Y. Gao, G. He, C. Hou, and S. Paul, "COCOON: An Alternate Approach to End-Host Congestion Management," http://stat.bell-labs.com/who/yuangao/papers/cocoon.pdf , accessed on Dec. 27, 2005.

[30] A. Venkataramani, R. Kokku, and M. Dahlin, "TCP-Nice: A Mechanism for Background Transfers," in *Proc. OSDI*, Dec. 2002, pp. 329–343.

[31] A. Kuzmanovic and E. Knightly, "TCP-LP: A Distributed Algorithm for Low Priority Data Transfer," in *Proc. IEEE INFOCOM*, Mar. 2003, pp. 1691–1701.

[32] J. Tang, G. Morabito, I. F. Akyildiz, and M. Johnson, "RCS: A Rate Control Scheme for Real-time Traffic in Networks with High Bandwidth-Delay Products and High Bit Error Rates," in *Proc. IEEE INFOCOM*, Apr. 2001, pp. 114–122.

[33] S. Yang and G. de Veciana, "Size-based Adaptive Bandwidth Allocation: Optimizing the Average QoS for Elastic Flows," in *Proc. IEEE INFOCOM*, Jun. 2002, pp. 657–666.

[34] S. Floyd, "TCP and Explicit Congestion Notification," *ACM CCR*, vol. 24, no. 5, pp. 10–23, Oct. 1994.

[35] B. Halabi, *Internet Routing Architectures*, Indianapolis, IN: Cisco Press, 2nd edition, 2000.

[36] T. S. E. Ng and H. Zhang, “Towards Global Network Positioning,” in *Proc. ACM SIGCOMM Internet Measurement Workshop*, Nov. 2001, pp. 25–29.

[37] VINT project at UCB/LBL, Xerox PARC, and USC/ISI, “The Network Simulator - ns-2,” http://www.isi.edu/nsnam/ns, accessed on Dec. 27, 2005.

[38] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang, “Recommendations on Queue Management and Congestion Avoidance in the Internet,” *IETF Internet RFC 2309*, Apr. 1998.

[39] D. Chiu and R. Jain, “Analysis of the Increase and Decrease Algorithm for Congestion Avoidance in Computer Networks,” *Journal of Computer Networks and ISDN*, vol. 17, no. 1, pp. 1–14, Jun. 1989.

[40] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker, “Dynamic Behavior of Slowly-responsive Congestion Control Algorithms,” in *Proc. ACM SIGCOMM*, Aug. 2001, pp. 263–273.

[41] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger, “Dynamics of IP Traffic: A Study of the Role of Variability and the Impact of Control,” in *Proc. ACM SIGCOMM*, August 1999, pp. 301–313.

[42] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan, and S. Tuecke, “Applied Techniques for High Bandwidth Data Transfers across Wide Area Networks,” in *Proc. International Conference on Computing in High Energy and Nuclear Physics*, Sep. 2001,

http://www.globus.org/alliance/publications/papers/dpss_and_gridftp.pdf, accessed on Dec. 27, 2005.

[43] M. Allman, H. Kruse, and S. Ostermann, "An Application-Level Solution to TCP's Satellite Inefficiencies," in *Proc. The First International Workshop on Satellite-based Information Services (WOSBIS)*, Nov. 1996, http://www.icir.org/mallman/papers/wosbis.ps, accessed on Dec. 27, 2005.

[44] H. Sivakumar, S. Bailey, and R. L. Grossman, "PSockets: The Case for Application-level Network Striping for Data Intensive Applications Using High Speed Wide Area Networks," in *Proc. ACM/IEEE Conference on Supercomputing*, Nov. 2000, CD-ROM.

[45] T. Dunigan, "Net 100 Project," http://www.csm.ornl.gov/d̃unigan/netperf/parallel.html, 2004, accessed on Dec. 27, 2005.

[46] V. Paxson, "End-to-End Internet Packet Dynamics," in *Proc. ACM SIGCOMM*, Sep. 1997, pp. 139–154.

[47] P. Sarolahti and A. Kuznetsov, "Congestion Control in Linux TCP," in *Proc. Usenix*, Jun. 2002, pp. 49–62.

[48] M. Mathis, J. Semke, J. Mahdavi, and K. Lahey, "The Rate Halving Algorithm for TCP Congestion Control," http://www.psc.edu/networking/rate_halving.html, Jun. 1999, accessed on Dec. 27, 2005.

[49] M. Carson and D. Santay, "NIST Net: A Linux-based Network Emulation Tool," *ACM CCR*, vol. 33, no. 3, pp. 111–126, Jul. 2003.

[50] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf Version 1.7.0," http://dast.nlanr.net/Projects/Iperf/, accessed on Dec. 27, 2005.

[51] S. Ostermann, "tcptrace 6.6.1," http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html, accessed on Dec. 27, 2005.

[52] H. Uijterwaal and M. Santcroos, "tcpdump 3.8.3," http://www.tcpdump.org/, accessed on Dec. 27, 2005.

[53] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *ACM CCR*, pp. 67–82, Jul. 1997.

[54] S. Floyd, M. Handley, and J. Padhye, "A Comparison of Equation-Based and AIMD Congestion Control," http://www.aciri.org/tfrc/, May 2000, accessed on Dec. 27, 2005.

[55] V. Jacobson, "Congestion Avoidance and Control," in *Proc. ACM SIGCOMM*, Aug. 1988, pp. 314–329.

[56] T. Dunigan, M. Mathis, and B. Tierney, "A TCP Tuning Daemon," in *Proc. ACM/IEEE Conference on Supercomputing*, Nov. 2002, pp. 1–16.

[57] M. Mathis, J. Heffner, and R. Reddy, "Web100: Extended TCP Instrumentation for Research," *ACM CCR*, vol. 33, no. 3, pp. 65–79, Jul. 2003.

[58] H. Bullot, R. L. Cottrell, and R. Hughes-Jones, "Evaluation of Advanced TCP Stacks on Fast Long-Distance Production Networks," in *Proc. Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2004, http://dsd.lbl.gov/DIDC/PFLDnet2004/papers/Bullot.pdf, accessed Dec. 27, 2005.

[59] R. L Cottrell, S. Ansari, P. Khandpur, R. Gupta, R. Hughes-Jones, M. Chen, L. McIntosh, and F. Leers, "Characterization and Evaluation of TCP and UDP-based Transport on Real Networks," in *Proc. Workshop on Protocols for Fast Long-Distance Networks*, Feb. 2005, http://www.slac.stanford.edu/cgi-wrap/getdoc/slac-pub-10996.pdf, accessed Dec. 27, 2005.

[60] R. King, R. Riedi, and R. Baraniuk , "TCP-Africa: An Adaptive and Fair Rapid Increase Rule for scalable TCP," in *Proc. INFOCOM*, Mar. 2005, pp. 1838–1848.

[61] 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*, vol. 28, no. 5, pp. 749–771, May 2002.

VITA

Name: Soohyun Cho

Address: Department of Computer Science, Texas A&M University, 301 Harvey R. Bright Bldg, College Station, TX 77843-3112, U.S.A.

E-mail Address: s0c6496@cs.tamu.edu

Education:

Ph.D. in Computer Science, May 2006, Texas A&M University, U.S.A.

M.S. in Electronic Engineering, August 1997, Korea University, Korea

B.S. in Electronic and Computer Engineering, February 1990, Korea University, Korea

Work Experience:

Research Engineer, Korea Telecom, September 1997 - August 2001

Research Engineer, Samsung Electronics, January 1990 - August 1995.

The typist for this thesis was Soohyun Cho.