Microscopic Behavior of Internet Congestion Control

Thesis by

Xiaoliang (David) Wei

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

2007

(Defended Feb 15, 2007)

Acknowledgements

I would like to thank my adviser, Prof. Steven Low, for his guidance and support. I have learned a lot on how to do rigorous theoretic research in working with Steven. Especially, I appreciate his support on my different opinions in the projects, even they were sometimes naive. Such support has been keeping me enthusiastic in the work. His patience and social grace with which he delivers his thoughts are very impressive. Under his guidance, I have learned much more than scientific knowledge.

Special thanks go to Prof. Pei Cao, my adviser in both Google and Stanford, for her inspiration and guidance in my experimental research. From the collaboration with Prof. Cao, I have learned how to do experimental research with convincing results. I really feel lucky to have got the chance to work with Pei and be exposed to the many challenging problems in large scale systems in the real world.

My gratitude also extends to Prof. Mani Chandy, Prof. John Doyle and Prof. Jason Hickey for serving in my committee for both the candidacy and the final exam. I benefit a lot from our discussions on the research progress.

I thank my officemates Dr. Anil Hirani and Jerome White. They shared so much experience with me that I was lucky to avoid many mistakes. Discussions with them also provided me a much larger view in the world beyond networking.

I thank all my colleagues and friends who shared my pain and happiness in NetLab: Dr. Lachlan Andrew, Dr. Lijun Chen, Dr. Cheng Jin, Dr. Lun Li, Dr. Mortada Mehyar, Dr. Kevin Ao Tang, Dr. Jiantao Wang and Dr. Bartek Wydrowski. Lachlan gave me many extremely helpful suggestions on the early drafts of this thesis. Also, I feel extremely fortunate to have Christine Ortega and Betta Dawson as our secretaries. They have offered such great help that all the paperwork and conference trips became

simple.

This thesis is based on the Latex class and Lyx template shared by Dr. Ling Li. Without his help, this thesis could not physically exist.

Finally, I would like to thank my parents and my girlfriend Chang Liu. Their support is the endless power that encourages me through the difficulties.

Abstract

The Internet research community has focused on the macroscopic behavior of Transmission Control Protocol (TCP) and overlooked its microscopic behavior for years. This thesis studies the microscopic behavior of TCP and its effects on performance. We go into the packet-level details of TCP control algorithms and explore the behavior in short time scales within one round-trip time. We find that the burstiness effects in such small time scales have significant impacts on both delay-based TCP and loss-based TCP.

For delay-based TCP algorithms, the micro-burst leads to much faster queue convergence than what the traditional macroscopic models predict. With such fast queue convergence, some delay-based congestion control algorithms are much more stable in reality than in the analytical results from existing macroscopic models. This observation allows us to design more responsive yet stable algorithm which would otherwise be impossible.

For loss-based TCP algorithms, the sub-RTT burstiness in TCP packet transmission process has significant impacts on the loss synchronization rate, an important parameter which affects the efficiency, fairness and convergence of loss-based TCP congestion control algorithms.

Our findings explain several long-standing controversial problems and have inspired new algorithms that achieve better TCP performance.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Transmission Control Protocol (TCP) is one of the most important protocols in the Internet protocol suites (often refereed as TCP/IP stack). It guarantees reliable and in-order data delivery from senders to receivers and is estimated to carry 70% to 95% of the Internet traffic in recent years. As the critical component that controls the TCP data transmission rate, the TCP congestion control algorithm plays a very important role in the performance of the Internet. There have been many studies on the TCP congestion control algorithm in terms of efficiency, stability, fairness, and scalability since its introduction in the late 1980's. There have been dozens of new proposals in the design and implementations of TCP congestion control. Most of these studies are based on models that focus on the macroscopic behavior of TCP congestion control algorithms. These models capture average data transmission rates in timescales of multiple round-trip times (RTT). They assume that the TCP data transmission process is a smooth and differentiable process. This assumption is, however, in sharp contrast to real TCP implementations, which produce bursty traffic in various timescales.

This thesis investigates the microscopic behavior of TCP. In particular, we study the packet-level details of TCP behavior in timescales that are within an RTT. Our study finds that the microscopic effects of window-based TCP implementation, (*ack-clocking* effects), have huge impacts on TCP's stability, fairness, and convergence. Our findings clarify several long-standing misconceptions in the network research community. For example:

- Stability of TCP Vegas
- Fairness of the Multiplicative-Increment-Multiplicative-Decrement (MIMD) algorithms
- The performance of TCP Pacing
- Friendliness between TCP and TCP Pacing

Our findings provide explanations to these questions, which are seemingly unrelated under the existing macroscopic models. Our study also suggests new algorithms that improve TCP performance in terms of responsiveness and fairness convergence.

1.1 Window-based implementation of TCP

Transmission control protocol (TCP) is a window-based protocol for reliable data transmission [1]. The sender sends a window of packets to the receiver and waits for acknowledgments from the receiver. The data packets are labeled with sequence numbers.[1] When the receiver receives a packet, it puts the packet into its buffer and sends back to the sender an acknowledgment packet, with one integer indicating the highest sequence number of consecutive packets received by the receiver. We say a data packet is *acknowledged* when the acknowledgment of this data packet or a later data packet arrives at the *sender*. If no packet is acknowledged within a certain time threshold[2], the sender assumes the previously sent packets are lost. The sender cannot send more packets until it receives the acknowledgments of some previously sent packets or it assumes some previously sent packets are lost.

In this process, the number of packets that have been sent by the sender but not acknowledged is called a "window". When the window size is fixed, the transmission of new data packet is triggered by the arrival of the acknowledgment of the previously

[1] More accurately, each octet (8 bit byte) is the basic unit of data in TCP and each octet has a unique sequence number. To simplify the discussion, we take the packet as the basic unit of data in this thesis.

[2] The time threshold is called RTO. RTO is always larger than an RTT and is usually equal to RTT plus four times of the variance in RTT [2].

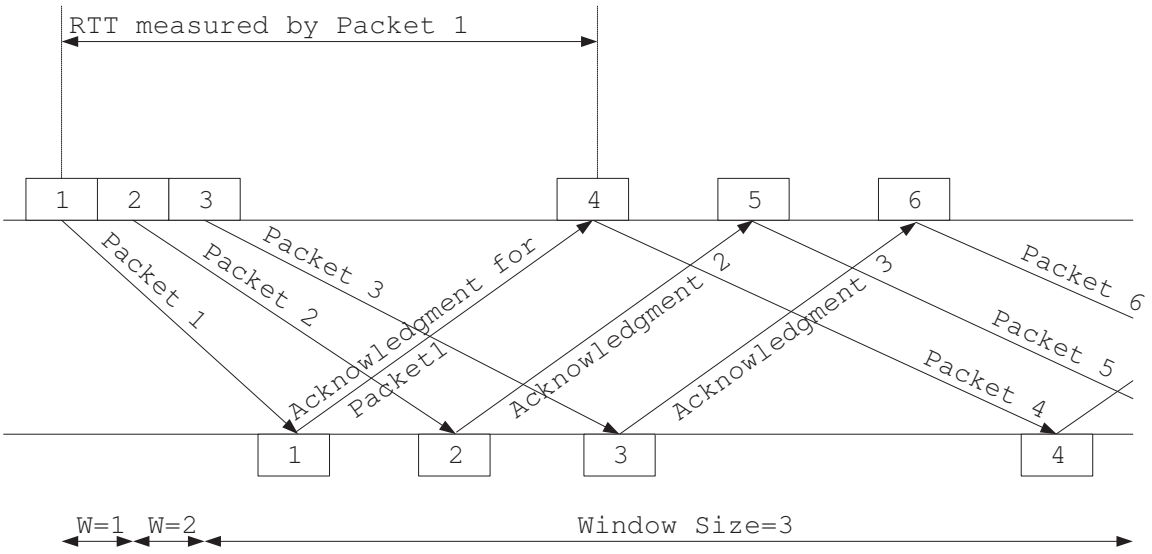


Figure 1.1: *Ack-clocking* effect in TCP data transmission

sent packets. This unique feature of TCP data transmission is called "*ack-clocking*". This process is illustrated in Figure 1.1.

With *ack-clocking*, the size of the window implies the average rate of the data transmission. Since there is always a window of packets that are sent out but not acknowledged, and the acknowledgment of each packet takes one round-trip from the sender to the receiver and back to the sender, only one window of packets can be sent in each round-trip time (RTT). Hence, the average rate of the data transmission process is $\frac{\text{window size}}{\text{RTT}}$.

The TCP congestion control algorithm was introduced to control the size of the "window" of each TCP flow so that the TCP packet transmission rates do not exceed the network capacity.[3] With the TCP congestion control algorithm, the "window" for a flow (flow i) is a function of time t. We denote it as $w_i(t)$, and it is controlled in RTT time scale by a congestion control algorithm according to the packet delay, or loss information measured by the sender. There have been many proposals on how to design congestion control algorithms with delay or loss information.

[3]This thesis focuses on the congestion in network. The concept of *window* in this thesis always means the congestion window. In real TCP, there is also a concept of *advertised window*, which is used to avoid end-host congestion.

Loss-based congestion control schemes use packet loss as a signal of network congestion. For each RTT in which a TCP source i does not detect a packet loss, a loss-based congestion control algorithm assumes that the network is under-utilized and gradually increases w_i to increase the throughput of flow i. For each RTT in which the TCP source detects one or more packet loss, the congestion control algorithm assumes that the network is congested and drastically decreases w_i to relieve network congestion. To avoid under-utilizing the network, loss-based algorithms have to periodically generate loss. Most of the existing congestion control algorithms are loss-based [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Delay-based algorithms use the change in RTT, measured by the delay between the packet transmission time from sender and the acknowledgment arrival time to sender, to infer the congestion level. When RTT exceeds a threshold, a delay-based congestion control algorithm assumes that the network is congested and reduces w_i; when RTT is below another threshold, the algorithm assumes the network is under-utilized and increases w_i. Some examples of delay-based congestion control algorithms are [13, 14, 15, 16, 17]. Both loss-based congestion control algorithms and delay-based congestion control algorithms control the average rate of a TCP flow in time scale of RTT. This time scale is natural since the congestion control algorithm is a feed-back control mechanism with a feedback delay of one RTT.

Within one RTT, the underlying packet transmission process is controlled by the *ack-clocking* mechanism. *Ack-clocking* maintains a variable called "packets-in-flight" (p), which is defined as the number of packets that are sent, but not acknowledged. At any time t and for each flow i, *ack-clocking* always tries to match $p_i(t)$ with the window $w_i(t)$ specified by the congestion control algorithm. The behavior of ack-clocking can be described in Algorithm 1. When $p_i(t)$ is larger than $w_i(t)$, no packet is sent for the arrival of an acknowledgment. Whenever $p_i(t)$ is smaller than $w_i(t)$, ack-clocking implementation sends $w_i(t) - p_i(t)$ packets in a burst at the line-rate of the sender's network interface card (NIC) to fill the gap.[4] This happens when an

[4]Some implementations may even send this burst of packets in the speed of CPU, which is usually much faster than NIC speed.

Algorithm 1 Pseudo-code of *Ack-clocking*

When an acknowledgment that acknowledges k packets is received by flow i, or upon the start of the flow i:

1. $p_i \leftarrow p_i - k$;
2. $w_i \leftarrow F(w_i)$
3. while $(p_i < w_i)$
   - $p_i \leftarrow p_i + 1$
   - send a packet;

$F(w_i)$ is the response function of a congestion control algorithm. Besides w_i, the response function of a loss-based algorithm takes packet loss information as an input; the response function of a delay-based algorithm takes packet delay information as an input.

acknowledgment arrives (p_i is decreased) or the congestion window $w_i(t)$ is increased by the congestion control algorithm. This burst of packets introduces two levels of burstiness: micro-burst and sub-RTT burstiness.

1.1.1 Micro-burst

Whenever $w_i(t) - p_i(t) > 1$, multiple packets are sent into the network back-to-back. Such a burst, called *micro-burst* [18], has a peak rate higher than the bottleneck capacity and introduces an additional queueing delay to the router. If the bottleneck buffer size is smaller than the size of the micro-burst, some packets in the micro-burst are dropped. Otherwise, the micro-burst enters the bottleneck buffer and generates an additional queueing delay equal to the length of the burst.

There are two situations wherein micro-burst is formed:

The first is a sudden increment of the congestion window $w_i(t)$. In the start-up phase of a flow, TCP uses *slow-start* [4] to probe the bottleneck's available bandwidth. Slow-start doubles the congestion window every round-trip. This quick increment in the congestion window leads to a gap between $w_i(t)$ and $p_i(t)$ and results in micro-burst.

The second is a sudden decrement in the number of packets in flight $p_i(t)$. Acknowledgments are not reliably transmitted in the network; they can be delayed or dropped in their return paths, due to congestion. When acknowledgments are dropped in the reverse path of the network, the TCP sender cannot send any new packets. Once a later acknowledgment arrives, the sender recognizes that several packets have arrived at the receiver (since each acknowledgment acknowledges *all* the in-sequence packets that are received by the receiver) and drastically drops $p_i(t)$.[5] The gap between $w_i(t)$ and $p_i(t)$ results in micro-bursts.

Micro-burst is transient and can be mitigated by various methods, such as pacing [20, 21], burstiness control [22], or other mechanisms [18]. Since the TCP can only send, at most, a window of packets into the network, the size of a micro-burst, in term of number of packets, will never exceed the window size. Hence, the effect of micro-burst on packet loss can be eliminated by increasing buffer size. It has been suggested that the buffer size in the router should hold at least half of the maximum congestion window so that the the micro-burst triggered by slow-start, which is half of the congestion window, can be fully absorbed by the router's buffer without packet loss [4].

1.1.2 Sub-RTT burstiness

After being buffered in the bottleneck router, the back-to-back packets within a micro-burst are processed by the router at rate c (packet/second), the router's capacity. After one round-trip, the acknowledgments of these data packets return to the sender at rate c. The sender then sends the next window of packets at rate c and waits for the rest of the RTT until new acknowledgments come back. Hence, in sub-RTT time scales, the sending rate $x_i(t)$ can be approximated by an on-off process. In the *on* period, packets are transmitted at rate c packets per second, which is usually much higher than the average rate of $\frac{w_i}{RTT}$. We call this burstiness *sub-RTT level burstiness*.

Sub-RTT level burstiness does not introduce excessive packet loss or additional

[5] This situation is called *ack-compression* in [19].

queueing delay (since its peak rate is no greater than the bottleneck capacity c). However, *sub-RTT level burstiness* affects the packet arrival pattern of individual flows. The *on-off* pattern has significant impact on the fairness of loss-based congestion control algorithms, as we will explain in Chapter 3.

Once *sub-RTT level burstiness* is formed, the burstiness is maintained by *ack-clocking* and its effect cannot be eliminated by a large buffer size or high multiplexing level. It has been shown that sub-RTT level burstiness persists in scenarios with a single TCP flow as well as in daily Internet traffic (from router trace) where the number of flows is very large [23]. Figure 1.2 illustrates such an example. In this example, 16 TCP flows share a 100Mbps bottleneck with a delay of 10ms and a buffer size of 250 packets in NS-2 simulation. We record the data packet process of each flow at the bottleneck link. The data presented in the figure is collected 1300 RTTs after the flows start, when the flows have been in the congestion avoidance phase for a long time. A green dot $(t, i), i = 1 \cdots 16$ in the figure represents a packet from flow i going through the bottleneck at time t. The green dots would evenly distribute on a horizontal line if the data process was smooth. This figure intuitively shows that:

1. Within each RTT, almost all flows have data packet processes that are on-off;
2. This on-off pattern within a RTT (sub-RTT burstiness) is maintained throughout the life of connections.

Later on in this thesis, we use *bursty TCP* (or *current TCP*) to denote a normal TCP (with *ack-clocking*) and differentiate it from other special TCP implementations which eliminate *ack-clocking* effects by mechanisms such as *TCP pacing* (or *paced TCP*)[20].

1.2 Fluid models

The TCP research community has been developing several macroscopic models to understand the TCP behavior.

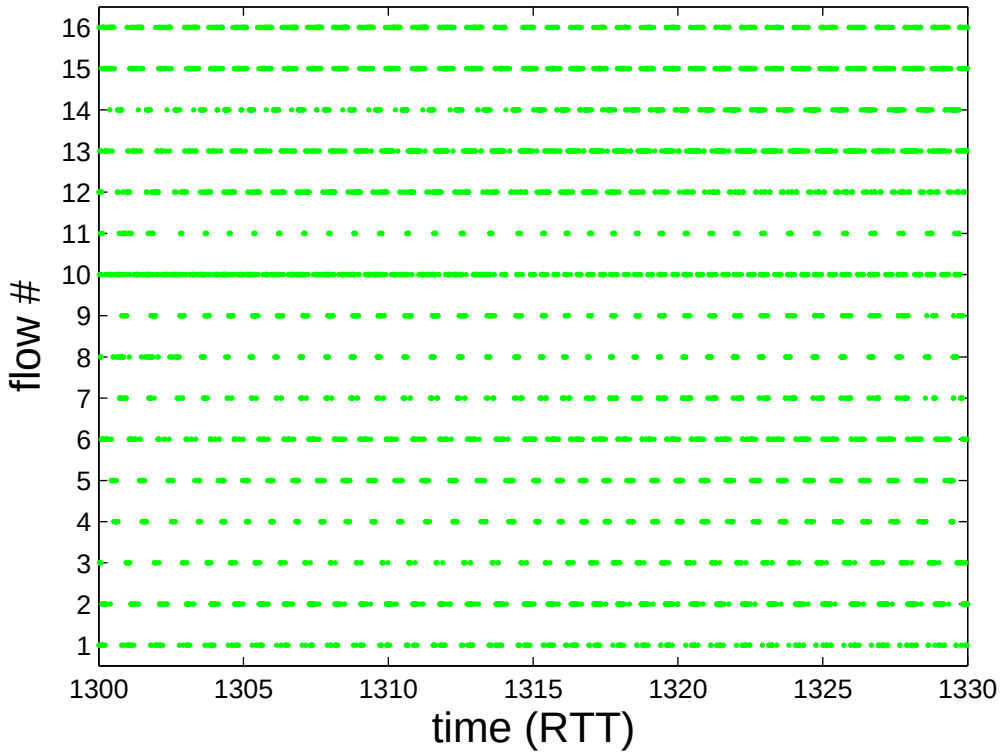


Figure 1.2: Sub-RTT level burstiness

A macroscopic model for average TCP throughput was proposed in [24]. The macroscopic model captures the relation between packet loss rate and average TCP throughput over a loss epoch. A series of work in late 1990s [25, 26] refined the macroscopic model and established a *fluid model* to study the macroscopic behavior of TCP Reno. Fluid model assumes that the data transmission rate of a flow i ($x_i(t)$) is differentiable and equal to the average throughput in a round trip ($\frac{\text{window size}}{\text{RTT}}$). Based on the fluid models, the TCP congestion control problems were mapped to the traditional control problems [26, 27]. This approach allows the research community to explore the dynamic properties of TCP/AQM in a rigorous manner. Results based on this mapping appear in current literature [27, 28, 29, 30].

We give a brief introduction of this model in the context of homogeneous flows. Given a network with N flows $\{s_1, s_2, \cdots, s_N\}$ and a single bottleneck link. Define the link's backlog at time t as $b(t)$ and its capacity as c. The path has a propagation delay of $\tau = \tau^F + \tau^B$, where τ^F is the forward propagation delay and τ^B is the backward propagation delay. The round trip time (RTT) of the path at time t is denoted by $D(t)$.

Assume the sending rate of flow i is determined by its window $w_i(t)$ and the round trip delay $D(t)$ as

$$x_i(t) = \frac{w_i(t)}{D(t)} \tag{1.1}$$

where

$$D(t) = \tau + \frac{b(t)}{c} \tag{1.2}$$

the aggregate input rate for the link is $x(t) = \sum_i x_i\left(t - \tau^F\right)$. Then, the backlog process can be modeled by

$$\dot{b}(t) = \begin{cases} \sum_i x_i\left(t - \tau^F\right) - c & \text{if } b(t) > 0 \\ \max\left\{0, \sum_i x_i\left(t - \tau^F\right) - c\right\} & \text{if } b(t) = 0 \end{cases} \tag{1.3}$$

The congestion window $w_i(t)$ of source i is determined by the congestion control

function based on the feedback with the assumption that $w_i(t)$ is differentiable:[6]

$$\dot{w}_i(t) = F\left(w_i(t), q\left(t - \tau_i^B\right)\right) \tag{1.4}$$

where

$$q(t) = \frac{b(t)}{c} \tag{1.5}$$

is the feedback from the router. It can be either queueing delay or packet loss rate.[7]

(1.1), (1.2), (1.3), (1.4) and (1.5) form an ordinary differential equation (ODE) system. Traditional control theory can be applied to this system to analyze the dynamic properties of TCP. This fluid model has greatly inspired the network community and the control community. Hundreds of papers have been published to analyze the stability of different TCP systems in different scenarios. Based on this model, the results always show that there is a stability region for a TCP congestion control algorithm. The region depends on several parameters: the number of flows (N), the round trip delay (D), the bottleneck capacity (c), and some algorithm specific parameters. The systems usually become unstable when N is small, D is large, or c is large.

However, we are cautious of this approach as the fluid models only capture the macroscopic behavior of TCP. The fluid models assume that the TCP data transmission process $x_i(t)$ is smooth and differentiable. This is not true in reality. As introduced in Section 1.1, burstiness is very common in real TCP systems. Microburst corresponds to a pulse function in $x_i(t)$ and sub-RTT burstiness corresponds to a step function in $x_i(t)$. The stability of a system can be completely different if a pulse function or a step function is included. Furthermore, the predictions of the fluid models contradict with experimental results in some scenarios, as shown in Section 1.3. We have to go into the details of microscopic TCP behavior and understand these contradictions.

[6]Here the feedback delay of $q(t)$ is assumed to be the constant τ_i^B. This is not true since the feedback will be further delayed by the queueing delay. However, all the models used in [27, 28, 29] have the same assumption.

[7]If the feedback is packet loss rate, the linear relation between $q(t)$ and $b(t)$ only holds when Random Early Dropping (RED) [31] is applied.

1.3 Controversial problems

There have been contradictions between the prediction of the fluid models and other heuristic understandings and experimental results. We list some of them below.

1.3.1 Stability of TCP Vegas

Stability of delayed-based algorithms (e.g. TCP-Vegas) is a controversial topic. With an approximate fluid model, it has been shown that TCP-Vegas has a small stability region and an algorithm called *Stabilized Vegas* was suggested to stabilize TCP-Vegas in [29]. However, another analytical result based on an extended fluid model showed that neither TCP-Vegas nor *Stabilized Vegas* is stable [30]. Both results showed that TCP-Vegas is stable only small capacity and small delay. They imply that TCP-Vegas will be unstable with large enough capacity or large delay. The claims are supported by NS-2 simulation in [29] and by ODE-based MatLab calculation in [30]. However, in our NS-2 TCP-Linux simulation and real experiments, TCP-Vegas rarely oscillates. Even when it oscillates in NS-2 simulations, the oscillation in the congestion window is always very small, which can be the integer truncation effect and other effects in NS-2 TCP-Vegas implementation. Therefore, it is not clear how to interpret the stability results in the fluid model analysis to the real performance of TCP-Vegas.

1.3.2 Fairness of homogeneous MIMD congestion control algorithms

Multiplicative-Increment-Multiplicative-Decrement (MIMD) algorithms are a class of control algorithms which increase and decrease the congestion window multiplicative. When MIMD algorithms do not observe congestion, they increase the congestion window by a small percentage. When MIMD algorithms observe congestion, they decrease the congestion window by a large percentage. It has been proved with a static model that two MIMD flows with different window sizes cannot converge to a fairness point even when they share the same network path [32]. However, the

analysis based on the fluid model proves that Scalable-TCP, an MIMD algorithm, can converge to fairness [33] . In many experiments, it is observed that Scalable-TCP cannot converge to fairness [34, 17], so, it is not clear if MIMD algorithms are fair or not.

1.3.3 Effect of TCP pacing

TCP pacing is proposed in [20] for high speed long distance network. TCP pacing uses a rate control mechanism to replace *ack-clocking* and eliminates both micro-bursts and sub-RTT burstiness as we described in Section 1.1. Since its introduction, there has been a long debate on the effect of TCP pacing. Simulation results [20] show that TCP pacing can significantly improve the throughput of TCP flows in networks with large capacity, long delay, and small buffer. Simulations results [21] also show that TCP pacing does improve TCP performance in both efficiency and fairness. It is also shown, however, that TCP pacing might actually have lower average throughput in many cases [35]. Even worse, TCP pacing loses to normal TCP when the two co-exist. On the other hand, the network industry, especially the network interface card design industry, has increasingly adopted the TCP pacing technology into their products. Therefore, it is not clear what exactly are TCP pacing's effects on performance.

1.4 Scopes and limitations

We explore the microscopic behavior of TCP and find new answers to these questions. We focus on the performance of homogeneous TCP flows. Similar to Section 1.2, the scenario in our study has N homogeneous TCP flows sharing a dumb-bell topology with the following parameters:

- N: the number of flows;
- τ: the propagation delay of the path;
- c: the capacity of the bottleneck (in packets per second);

- B: the buffer size of the bottleneck;
- $b(t)$: the bottleneck queue size;
- $q(t)$: the queueing delay or loss rate of the path;
- $D(t)$: the round trip time of the path;
- $w_i(t)$: the window size of flow i at time t;
- $x_i(t)$: the packet transmission rate of flow i at time t.

This study is the first step to exploring the microscopic behavior of TCP; further studies that cover more general cases with heterogeneous flows and multiple bottleneck sets are expected in the future.

1.5 Summary of results

To summarize, our investigations in microscopic behavior of TCP congestion control has found that the *ack-clocking* has significant impacts on TCP performance.

For delay-based congestion control algorithms, micro-burst makes the queue converge much faster than the fluid model predicts. This fast queue convergence leads to better stability of delay-based congestion control algorithms. With a packet-level model, we can prove that a single TCP Vegas flow is always stable with any delay and any capacity. This is in sharp contrast to the prediction of fluid models, which implies that TCP Vegas will be unstable with small number of flows, long delay and large capacity. The new understandings also allow us to design more aggressive new algorithms which are both stable and responsive. Inspired by the fast queue convergence, we design a new delay-based congestion control algorithm, FAST, to achieve responsive convergence and stable queueing delay. We show that homogeneous FAST flows are stable with any delay and any capacity.

For loss-based congestion control algorithms, the combination of burstiness in packet loss process and sub-RTT burstiness in TCP data packet process lowers loss

synchronization rates among TCP flows. Intuitively, this means that only a small fraction of the TCP flows detect packet loss during a congestion event and most other TCP flows do not detect the congestion. A low synchronization rate has several implications in the performance of TCP. First, with a low synchronization, the link utilization TCP Reno flows is higher since fewer flows slow down in each congestion event. Hence, the aggregate rate of bursty TCP can be higher than the aggregate rate of paced TCP. Second, a low synchronization rate implies poor short-term fairness. Hence, the paced TCP is fairer than bursty TCP. Third, when TCP and paced TCP co-exist, paced TCP flows lose to TCP flows since the paced TCP flows do not have sub-RTT burstiness in their data packet processes and have higher probability to detect packet loss in each congestion event. Finally, with sub-RTT burstiness, a large number of TCP flows with different window sizes tend to have similar probability to detect a loss in congestion event. This similarity validates the synchronization assumptions in [32] and leads to the unfairness of MIMD algorithms. Based on these understandings, we propose to use TCP pacing to improve the fairness of TCP, especially MIMD algorithms.We also propose a new link algorithm which provides consistent ECN signals to increase the loss synchronization rate over all the bursty or paced flows.

1.6 Organization of this thesis

Chapter 2 details the effects of micro-burst on the stability of delay-based congestion control algorithms. Chapter 3 explores the sub-RTT level burstiness and its effects on loss-based congestion control algorithms. We explain the methodologies used in our research in Chapter 4. We summarize our conclusions and plans for future work in Chapter 5.

Chapter 2

Microscopic Effects on Delay-based Congestion Control Algorithms

We focus on the effects of micro-bursts on delay-based algorithms. As explained in Section 1.1.2, sub-RTT burstiness does not introduce additional delay, and we have not found any sub-RTT burstiness effect on delay-based congestion control algorithms. Micro-bursts, however, have significant effects on the stability of delay-based congestion control algorithms. We found that micro-bursts allow the queueing delay in the network system converge in a very short time and help to stabilize the system in the presence of feedback delay.

Stability of delay-based congestion control algorithms (e.g. TCP-Vegas [16]) has been a controversial topic in the past years. One approximate fluid model shows that TCP-Vegas has a small stability region and suggests an algorithm, *Stabilized Vegas*, to stabilize TCP-Vegas [29]. However, an extended fluid model considering RTT variation shows that both TCP-Vegas and *Stabilized Vegas* are not stable [30]. In both studies, fluid model analysis shows that TCP-Vegas is not stable with large enough capacity and delay. However, in all of our simulations and experiments, TCP-Vegas rarely oscillates. Even when it oscillates in simulations, the oscillation in the congestion window size is always smaller than ± 2 packets, which can be well explained by the integer quantization effect in implementation. Hence, there is no convincing experimental evidence to verify whether TCP Vegas is stable or not. On the other hand, there is no theory to prove the stability of TCP Vegas either.

This chapter provides a new answer that takes into consideration of micro-burst effects. With a packet level model, we prove that a single TCP-Vegas flow is always stable, regardless of the round trip delay and the bottleneck capacity. This result, which agrees with our observations in simulations and experiments, is in sharp contrast from the fluid model results.

The process of the proof reveals many properties of *ack-clocking* and convergence of queue. The new understandings allow us to design new delay-based congestion control algorithms that are much more responsive yet stable, which would be impossible according to the analysis of the fluid models. In particular, we design a new delay-based congestion control algorithm, *FAST*, to achieve much faster convergence than TCP Vegas while maintaining good stability. We prove that *FAST* is stable with homogeneous sources in a network with any capacity and any delay.

2.1 Stability of a single TCP-Vegas flow

We propose a detailed packet-level model for a single TCP flow controlled by a delay-based congestion control algorithm. This model reveals several interesting properties of *ack-clocking* in a single bottleneck link. It leads to a stability proof of a single TCP-Vegas flow.

2.1.1 Modeling *ack-clocking*

We model *ack-clocking* at packet level. With the scenario of a single flow, we are able to capture the timing of each individual packet in the model and to understand the details of the *ack-clocking* effect.

2.1.1.1 Assumptions

We make the following three assumptions:

1. The bottleneck router has a deterministic capacity of c and an infinite buffer;

2. The links on the path have no packet loss and produce consistent round trip propagation delay of d sec; this round trip propagation delay includes the router's packet processing time $\frac{1}{c}$, and hence, $d \geq \frac{1}{c}$;

3. The source can send a micro-burst of packets instantaneously when the congestion window is larger than the number of packets in flight; after a micro-burst of packets are sent, the number of packets in flight is equal to the congestion window.

The first two assumptions are very common in models for delay-based protocols. These two assumptions also appear in fluid model analysis such as [29]. The third assumption is the key point of our packet level model. Our model allows a micro-burst to be sent instantaneously. In fluid model, the third assumption is replaced by the fluid assumption that the sending rate $x_i(t)$ is a differentiable process.

The window size of a flow $w(t)$ is a given process in the model.

2.1.1.2 A packet level model for *ack-clocking*

We label the packets sent in the life of a connection with consequent integer numbers. The packet numbers form a sequence $\{j \,|j \in Z \text{ and } j \geq 0\}$.

For each packet j:

$s(j)$ is the sending time of the packet. By the definition of packet label j, we have

$$\forall j : s(j) \leq s(j+1) \tag{2.1}$$

$p(j)$ is the number of packets in flight after packet j is sent. It is an integer number. By definition of packet label j, $p(j)$ can increase by at most one per packet:[1]

$$1 \leq p(j) \leq p(j-1) + 1 \tag{2.2}$$

$a(j)$ is the arrival time of the acknowledgment of packet j. For simplicity, we call

[1]Since $p(i)$ is defined on packet and more than one packets can be sent at the same time, this constraint still allows the congestion window to increase by more than one at the same time.

"the arrival of the acknowledgment of packet j" as "the arrival of packet j" in the rest of this chapter. This refers to the time $a(j)$.

$b(j)$ is the backlog experienced by packet j; hence, $\frac{b(j)}{c}$ is the queueing delay experienced by packet j.

In this model, $w(t)$ is a given process that satisfies: $w(t) \geq 1$.

Initially, we have $a(0) = 0$, $p(1) = 1$, and $b(1) = 0$.

Given the initial condition and $w(t)$ sequence, we can uniquely determine $p(j)$, $s(j)$, $b(j)$ and $a(j)$ from the following four equations:

$$p(j) = \max_{0 \leq k \leq p(j-1)} \{p(j-1) - k + 1 \,|\, p(j-1) - k + 1 \leq w(a(j-1-p(j-1)+k))\} \tag{2.3}$$

$$s(j) = a(j - p(j)) \tag{2.4}$$

$$b(j) = \max\{b(j-1) + 1 - [s(j) - s(j-1)]\, c, 0\} \tag{2.5}$$

$$a(j) = s(j) + d + \frac{b(j)}{c} \tag{2.6}$$

Note:

- (2.3) is based on the *ack-clocking* algorithm described in Algorithm 1. k is the number of acknowledgments that the sender receives between $s(j-1)$ and $s(j)$. $a(j-1-p(j-1)) = s(j-1)$ is the sending time of the $(j-1)$-st packet. Hence, $a(j-1-p(j-1)+k)$ is the arrival time of the k-th acknowledgment after $s(j-1)$ and $w(a(j-1+k-p(j-1)))$ is the window size at that time. $w(a(j-1-p(j-1)+k))$ upper-bounds $p(j)$ if packet j is to be sent at this time. $p(j-1) - k$ is the number of packets in flight after the sender receives k acknowledgments. $p(j)$ cannot be higher than $p(j-1) - k + 1$ since sending packet j can only increase the number of packets in flight by 1. A quick corollary

from (2.3) is

$$p(j) = \max_{0 \le k \le p(j-1)} \{p(j-1) - k + 1 \,|\, p(j-1) - k + 1 \le w(s(j))\} \tag{2.7}$$

since $s(j) = a(j - p(j))$.

- (2.4) states that packet j should be sent at the arrival of the acknowledgment of a packet that were sent one RTT ago. Since $p(j)$ packets are sent in one RTT, $j - p(j)$ is the packet that is sent one RTT ago. A quick corollary of (2.4) is

$$p(j) = |\{k : s(k) \le s(j) < a(k)\}| \tag{2.8}$$

  This conforms to the definition of $p(j)$ which is the number of packets that are sent but not acknowledged right after packet j is sent. Also note that when $p(j) = p(j-1)+1$, we have $s(j-1) = s(j)$. This corresponds to the case with two packets in the same micro-burst sent out instantaneously by Assumption 3. The combination of (2.3) and (2.4) guarantees that $s(j-1) \le s(j)$.

- (2.5) is a discrete version of the backlog process in network calculus [36]:

$$b(t) = \max_{s \le t} \int_s^t [x(u) - c]\, du \tag{2.9}$$

During the time $s(j-1)$ to $s(j)$, at most $[s(j) - s(j-1)]\, c$ packets are processed by the bottleneck and leave the queue. One more packet enters the queue.

- (2.6) is based on the definition of round trip time, which equals to the round trip propagation delay (d) plus the queueing delay ($\frac{b(j)}{c}$).[2]

2.1.2 Properties of *ack-clocking*

From the packet level model, we have three properties of *ack-clocking*:

[2] We assume that each data packet will results in one acknowledgment. There is no acknowledgment compression or delayed acknowledgment.

- Whenever a packet is sent, the number of packets in flight is always equal to the window size after the whole micro-burst is sent.
- The acknowledgments are always paced out by the bottleneck, even the corresponding data packets have entered the bottleneck in bursts.
- The queueing delay experienced by a packet is *directly* bounded by the number of packets in flight.

These properties are important for us to understand the microscopic behavior of TCP.

2.1.2.1 Relation between the number of packets in flight and the window size

Theorem 2.1.2.1:

At any time $s(j)$ in which a packet is sent into the network,

$$p(j) \leq w(s(j)) \tag{2.10}$$

And there always exists a packet $j^* ::= j^*(j)$ which is sent at the same time ($s(j) = s(j^*)$), and

$$p(j^*) = w(s(j^*)) \tag{2.11}$$

Furthermore, if $w(s(j^*)) \geq w(s(j^*+1))$,

$$p(j^*+1) = w(s(j^*+1)) \tag{2.12}$$

(All proofs are in Appendix 6.3.)

This theorem reflects the assumption that the sending TCP sends all packets in a micro-burst at the same time.

First, (2.10) shows that *ack-clocking* guarantees that the number of packets in flight is always no greater than the congestion window size, at any time when a packet is sent;

Second, j^* is the last packet in the micro-burst. (2.11) says that the ack-clocking algorithm synchronizes the number of packets in flight with the window size at any time some packet is sent;

Third, (2.12) shows that the size of the micro-burst will be one packet if the congestion window does not increase.

2.1.2.2 Pacing of acknowledgments

Theorem 2.1.2.2:

$$\forall j : a(j) - a(j-1) \geq \frac{1}{c} \tag{2.13}$$

The equality holds if, and only if, $s(j) \leq s(j-1) + \frac{b(j-1)+1}{c}$.

This theorem implies that the acknowledgment packets are always paced out by the bottleneck router, no matter how fast the corresponding data packets have arrived at the bottleneck.

Corollary 2.1.2.2:

$$j_1 > j_2 \Leftrightarrow a(j_1) > a(j_2) \tag{2.14}$$

2.1.2.3 Upper bound of queue increment

Theorem 2.1.2.3:

For $\forall 1 \leq j' < j$, If $p(j')$,$p(j'+1), \cdots p(j)$ are non-decreasing,

$$b(j) \leq b(j') + p(j) - p(j')$$

This theorem upper bounds the increment of queue length. It says the increment of queue length is no greater than the increment of the number of packets in flight.

2.1.2.4 Lower bound of queue

Theorem 2.1.2.4:

$$d + \frac{b(j)}{c} \geq \frac{p(j)}{c} \tag{2.15}$$

The equality holds if, and only if, $\forall k$ that satisfies $j - p(j) + 1 < k \leq j : a(k) - a(k-1) = \frac{1}{c}$.

This theorem says that the delay experienced by a packet is always lower-bounded by the number of packets in flight.

Notes:

- Since $p(j^*) = w(s(j^*))$, Theorem 2.1.2.3 and 2.1.2.4 show that the congestion window has a direct effect on queueing delay. When the window size is large, the queueing delay experienced by the last packets in the micro-burst will be lower-bounded by $\frac{w(s(j^*))}{c}$;
- If packet j is not the first packet in the micro-burst, it might experience higher delay than $\frac{p(j)}{c}$ due to the extra queueing delay introduced by micro-burst, unless the system is in some special state. The next subsection will detail this state.

2.1.3 Queue convergence

Since the window size $w(t)$ directly affects the queueing delay, the queue converges at a much faster speed than that predicted by fluid model. With a single flow, the queue converges to a stable state in one RTT if the congestion window remains larger than bandwidth propagation delay product in one round trip.

2.1.3.1 Definition of *Stable-Link* state

Definition 2.1.3.1:

We say the system is in a *stable-link* state upon the arrival of packet j if the system satisfies

$$\forall k \text{ that satisfies } j - p(j) < k \leq j : a(k) - a(k-1) = \frac{1}{c}$$

The *stable-link* state is an indication that all the bottleneck links on the path is saturated. It has several properties.

First, in the *stable-link* state, the number of packets in flight is equal to bandwidth delay product (BDP);

Second, the *stable-link* state persists as long as the number of packets in flight is larger than or equal to bandwidth propagation delay product ;

Third, the single flow system enters *stable-link* state when the number of packets in flight is higher than or equal to bandwidth propagation delay product for at least one RTT.

2.1.3.2 The number of packets in flight and BDP

Theorem 2.1.3.2:

$$\text{The system is in stable-link state upon the arrival of packet } j \Longleftrightarrow p(j) = cd + b(j) \tag{2.16}$$

This is the equality case of Theorem 2.1.2.4.

Note that $b(t) = w(t) - cd$ is the equilibrium state of the link in fluid models. In fluid models, it takes many round-trips for the links to converge to this equilibrium state. This theorem says that this equilibrium state holds at any time when a packet is sent, once the system is in *stable-link* state.

2.1.3.3 Persistence of *Stable-Link* state

Theorem 2.1.3.3:

If the system is in stable link state upon the arrival of packet j and $p(j+1) \geq cd$, then the system is in stable link state upon the arrival of packet $j+1$.

This theorem says that as long as the number of packets in flight is larger than or equal to bandwidth propagation delay product, the stable-link state persists.

2.1.3.4 Entrance of Stable-Link state

Theorem 2.1.3.4:

If $\forall k : j - p(j) < k \leq j : p(k) > cd$; the system enters stable-link state upon the arrival of j.[3]

This theorem provides a sufficient condition for a system to enter stable-link state. It says as long as the number of packets in flight is larger than bandwidth propagation delay product for one RTT, the system will be in stable-link state.

This theorem implies that the queue dynamic converges within one round-trip time, which is a sharp difference from the prediction of fluid models.

2.1.3.5 Pacing of micro-burst

Theorem 2.1.3.5:

If $\forall k : j - p(j) < k \leq j : p(k-1) \geq p(k)$ and $p(j) \leq cd$, the system has $b(j) = 0$.

This theorem says that micro-burst can be smoothed by the bottleneck within one RTT, if the number of packets in flight does not increase.

2.1.4 Properties of congestion control in RTT timescale

In general, a delayed-based congestion control algorithm can be modeled as follows: the source makes a decision on new value of the congestion window at the arrival time of some packets, whose sequence numbers form a sub-sequence $\{\tau_k \,|k \in Z \text{ and } \tau_k < \tau_{k+1}\}$ of the packet number sequence $\{j\}$. We call these packets *decision packets*. Initially, $\tau_0 = w(0)$.[4]

Whenever a *decision packet* τ_k arrives (at time $a(\tau_k)$), the congestion window control algorithm makes the window update decision based on the window size when τ_k is sent ($w(s(\tau_k))$) and backlog experienced by τ_k ($b(\tau_k)$):[5]

[3] A more general version of this theorem which replaces the condition $p(k) > cd$ by $p(k) \geq cd$ also holds .

[4] A complete TCP congestion control algorithm usually includes three phases: slow-start, congestion-avoidance, and loss-recovery. In the context of delay-based congestion control algorithms, the model assumes an infinite bottleneck buffer size and hence no packet loss occur. The loss-recovery phase is not considered. The slow-start phase is an initial and transient phase for delay-based congestion control algorithms. The system will stay in congestion-avoidance phase after running for a long enough time. Hence, we only model the congestion avoidance phase in the study. The initial time in this model can be regarded as the starting time of the congestion-avoidance phase.

[5] Here we use $\alpha = \beta$, as in [29]. The proof can be extended to the cases with $\alpha < \beta$.

$$\Delta w(\tau_k) = R(w(s(\tau_k)), b(\tau_k)) \tag{2.17}$$

R is called the *response function*. It depends on the history of the window size and the measured delay

the window size is then changed to:

$$w(a(\tau_k)) = w(s(\tau_k)) + \Delta w(\tau_k) \tag{2.18}$$

and the next decision packet τ_{k+1} is defined as : [6]

$$\tau_{k+1} = \tau_k + w(s(\tau_k)) + \max\{\Delta w(\tau_k), 0\} \tag{2.19}$$

For any time other than the arrival of a decision packet, the window size does not change:

$$w(t) = w(a(\tau_k)) \text{ if } a(\tau_k) < t < a(\tau_{k+1}) \tag{2.20}$$

Hence, the congestion control algorithm changes the window size on a timescale of RTT. This control timescale is general for all existing congestion control algorithms.

Corollary 2.1.4:

From (2.19),

$$\tau_{k+1} \geq \tau_k + w(s(\tau_{k+1}))$$

2.1.4.1 Timing of the decision packets

Theorem 2.1.4.1:

$$a(\tau_k) \leq s(\tau_{k+1}) < a(\tau_{k+1})$$

[6] This definition is based on those implementations which use a special packet to indicate the end of one RTT. Some implementations have a different value of τ_{k+1}. For example, Linux with delayed ack will have $\tau_{k+1} = \tau_k + 2w(s(\tau_k))$. We ignore these variants but note that the proof holds as long as $\tau_{k+1} \geq \tau_k + w(s(\tau_k)) + \max\{0, \Delta w(\tau_k)\}$.

This theorem says that in RTT-timescale window control, a decision packet is sent out only when the last decision packet has arrived. By (2.20), we have:

$$w\left(s\left(\tau_k\right)\right) = w\left(a\left(\tau_{k-1}\right)\right) \tag{2.21}$$

2.1.4.2 Equivalence of the window size and the number of packets in flight

Theorem 2.1.4.2:

$$\forall \tau_k : w\left(s\left(\tau_k\right)\right) = p\left(\tau_k\right)$$

This theorem says that each decision packet τ_k is the last packet sent in the micro-burst. And the window control formula (2.17) can be rewritten as a function of $p\left(\tau_k\right)$:

$$\Delta w\left(\tau_k\right) = R\left(p\left(\tau_k\right), b\left(\tau_k\right)\right) \tag{2.22}$$

2.1.4.3 Link convergence upon decision packets

Theorem 2.1.4.3:

$b\left(\tau_k\right) \leq \Delta w\left(\tau_k\right)$ or the system is in the stable-link state upon the arrival of τ_k.

This theorem says that either $b\left(\tau_k\right) \leq \Delta w\left(\tau_k\right)$ or $b\left(\tau_k\right) = p\left(\tau_k\right) - cd$, according to the properties of the stable-link state.

According to Theorem 2.1.4.2, we have:

$$b\left(\tau_k\right) = w\left(s\left(\tau_k\right)\right) - cd$$

or

$$b\left(\tau_k\right) \leq \Delta w\left(\tau_k\right)$$

The theorem establishes a very different understanding on the effect of feedback delay in TCP system. In the traditional fluid models, the queueing delay has slow dynamics and converges asymptotically to a new equilibrium when the congestion window changes. Due to this slow dynamic, there is a difference between queueing

delay observed by a source and the queueing delay at the bottleneck. Hence, the congestion control algorithm might oscillate its congestion window due to overacting upon the observed queueing delay. Even worse, the longer the propagation delay, the slower the queue converges, and the easier the congestion control algorithm oscillates. The feedback delay plays an important role in this system.

However, by capturing the micro-burst which leads to fast queue dynamic, Theorem 2.1.4.3 assures that as long as the queueing delay is higher than the change of the congestion window, the observed queueing delay equals the queueing delay at the bottleneck. Hence the process of the fast queue convergence process within in one round-trip is negligible and the queue size can be modeled by a static function in the form of $b = w - cd$ instead of a differential equation.

This understanding holds for all delay-based congestion control algorithms. We also believe that it can be extended to loss-based congestion control algorithms.

2.1.5 Stability of TCP Vegas

TCP Vegas ([16]) is a particular delay-based congestion control algorithm. Its response function is

$$R\left(w\left(s\left(\tau_k\right)\right), b\left(\tau_k\right)\right) = \left\{ \begin{array}{cc} 1 & \text{if } \frac{w(s(\tau_k))}{d} - \frac{w(s(\tau_k))}{D(\tau_k)} < \alpha \\ -1 & \text{if } \frac{w(s(\tau_k))}{d} - \frac{w(s(\tau_k))}{D(\tau_k)} > \alpha \text{ and } w\left(a\left(\tau_k\right)\right) > 1 \\ 0 & \text{Otherwise} \end{array} \right\}$$

where

$$D\left(\tau_k\right) = d + \frac{b\left(\tau_k\right)}{c} \tag{2.23}$$

With this response function, $\Delta w\left(\tau_k\right) \leq 1$. Hence, the size of micro burst introduced by the change of congestion window will be always smaller than one packet. Intuitively, once the system enters into a state in which $b\left(t\right) > 1$, the bottleneck queue size can be modeled by a static function and the stability of TCP Vegas does not depend on feedback delay. Theorem 2.1.5 confirms this intuition and shows that

a single TCP Vegas flow is always stable.

Theorem 2.1.5:

Given the ack-clocking model described in (2.3)(2.4)(2.5)(2.6) and the TCP Vegas congestion control algorithm described in (2.17)(2.23)(2.18)(2.19)(2.20), a single TCP flow converges to equilibrium regardless of capacity c, propagation delay d and initial state. That is:

If $\alpha d > 1$, given any initial state, we have

$$\exists J : \forall j > J : cd + \alpha d - 1 < w\left(s\left(j\right)\right) < cd + \alpha d + 1 \text{ and } \alpha d - 1 < b\left(j\right) < \alpha d + 1$$

Particularly, if $(cd + \alpha d) \in Z$, then $\forall j > J : w\left(s\left(j\right)\right) = cd + \alpha d$ and $b\left(j\right) = \alpha d$.

2.1.6 Validation

We run simulations with TCP Vegas implementation from Linux kernel and validate our results. To eliminate the effect of inaccurate base RTT estimation, we hard code the base RTT to be the propagation delay. With a single flow, TCP Vegas is stable with very long delay and high bottleneck capacity, as shown in Figure 2.1. Although the fluid model analysis predicts that the long delay and high capacity in the scenario leads to instability of TCP Vegas [29], the congestion window of the single TCP Vegas flow converges to equilibrium and remains stable in the region of $[19067, 19068]$ as predicted in Theory 2.1.5.

We repeat simulations with multiple homogeneous TCP Vegas flows and confirm that TCP Vegas is stable with different delay. We first repeat the simulations by Choe and Low in [29], with the more realistic TCP Vegas implementation from Linux kernel. Figure 2.2 shows the average congestion window trajectory for 100 TCP Vegas flows and the queue trajectory. From the full traces, we can see that both the congestion window and the queue converge. In fact, we inspect the congestion window of each *individual* flow and confirm that the congestion window of each individual flow also converges. In the enlarged versions, we inspect the oscillation at the packet level.

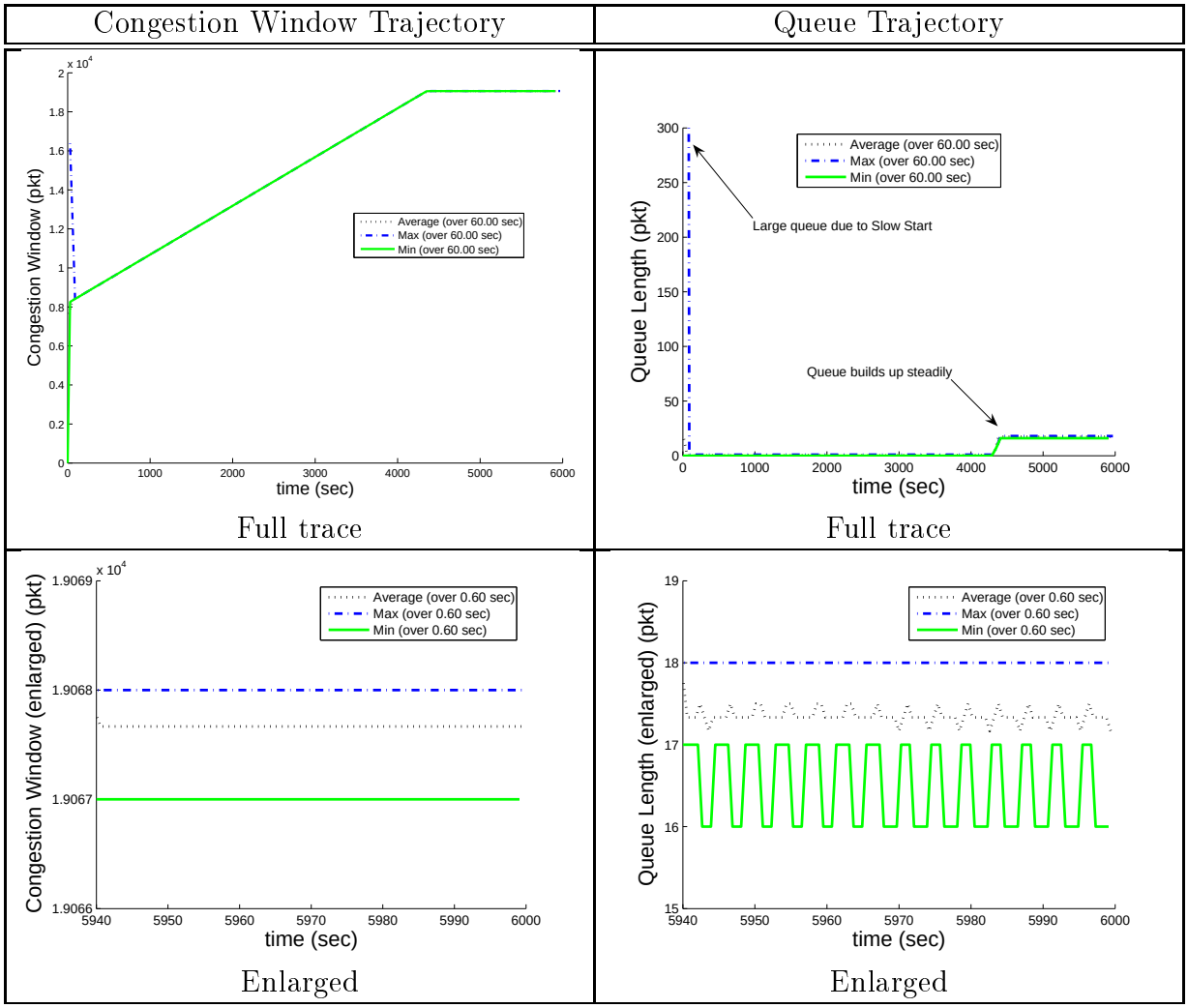


Figure 2.1: A single TCP Vegas flow using a path with a bottleneck capacity of 800Mbps and a propagation delay of 200ms. The packet size in the simulation is 1000 bytes per packet.

The oscillation is between 114.6 and 115.6. This is because the Linux uses an integer variable to store the congestion window size. When cd is not an integer, the congestion window has one packet of oscillation. In the enlarged version of queue trajectory, we observed that the queue length oscillates between 1935 packets and 2035 packets. This 100 packet worth of oscillations are due to the one packet worth of oscillation of each congestion window of each of the 100 flows.

To further confirm that our observed oscillations are due to integer effects only, we run two other sets of simulations, in which we double the round trip propagation delay. According to the prediction of fluid model, we expect a more severe oscillation. However, the simulation results show that the queue length only oscillates in the same region and the average congestion window oscillates within one packets, as shown in Figure 2.3 and Figure 2.4.

2.2 FAST algorithm and its stability

The packet level model for single source TCP-Vegas gives many new understandings of the queue dynamics. Intuitively, a change in the congestion window can result in a very quick change in the queue and controlling the congestion window directly controls the queue. This is different from the intuition from the fluid model in which controlling window only indirectly controls the queue via the rate process. This new understanding has inspired the design of a new algorithm, FAST.

2.2.1 FAST algorithm

FAST algorithm can be viewed as a high speed version of TCP Vegas. It has the same equilibrium state as TCP-Vegas. However, it converges much faster, and hence, is able to fully utilize the bottleneck capacity. The design of the FAST algorithm has been inspired by the concept of quick queue convergence.

The FAST algorithm can be summarized in the following equation where $w_i(t)$ is adjusted once every two round trips:

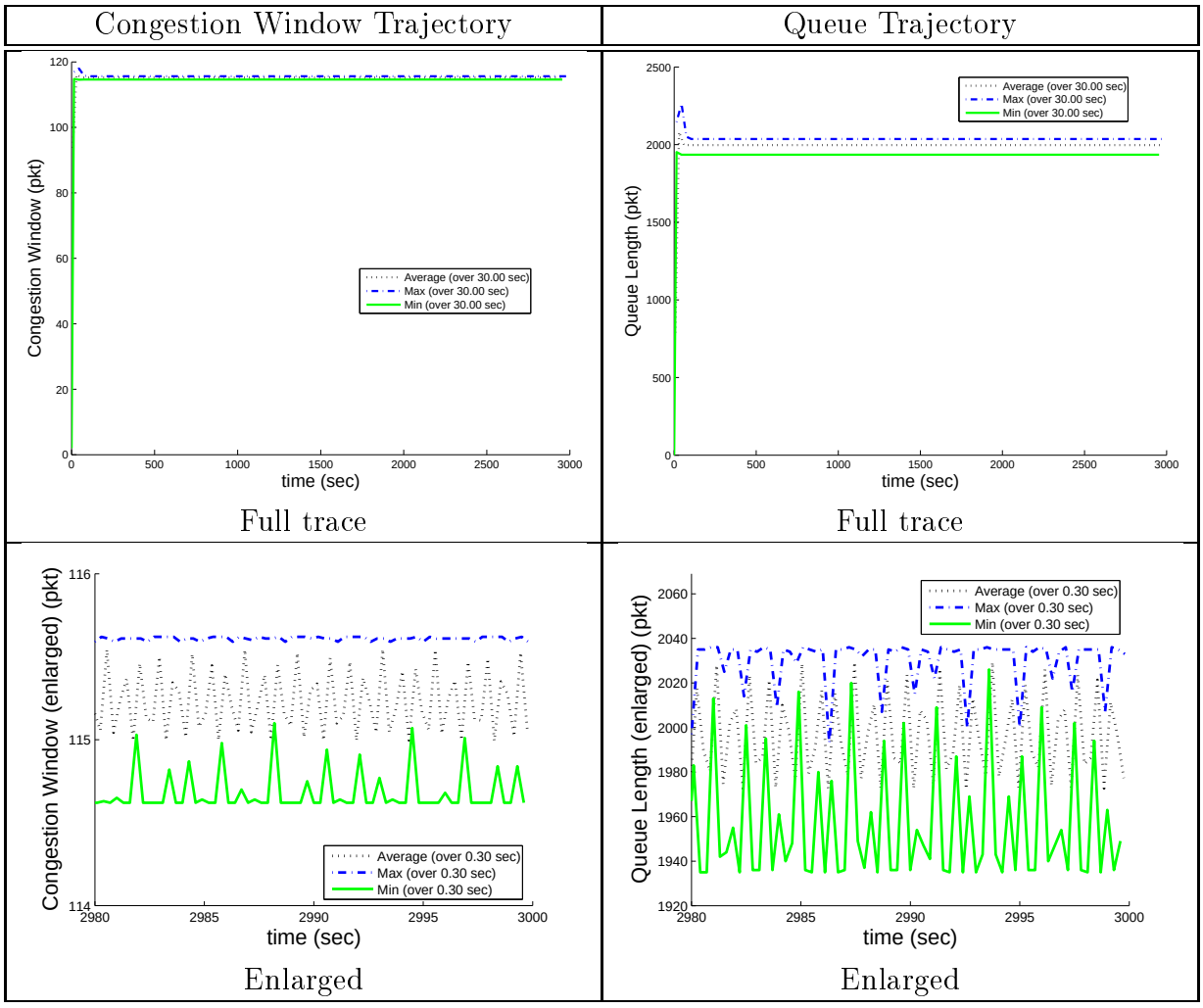


Figure 2.2: 100 Homogeneous TCP Vegas flows sharing a path with a bottleneck capacity of 800Mbps and a propagation delay of 200ms. The packet size in the simulation is 1000 bytes per packet.

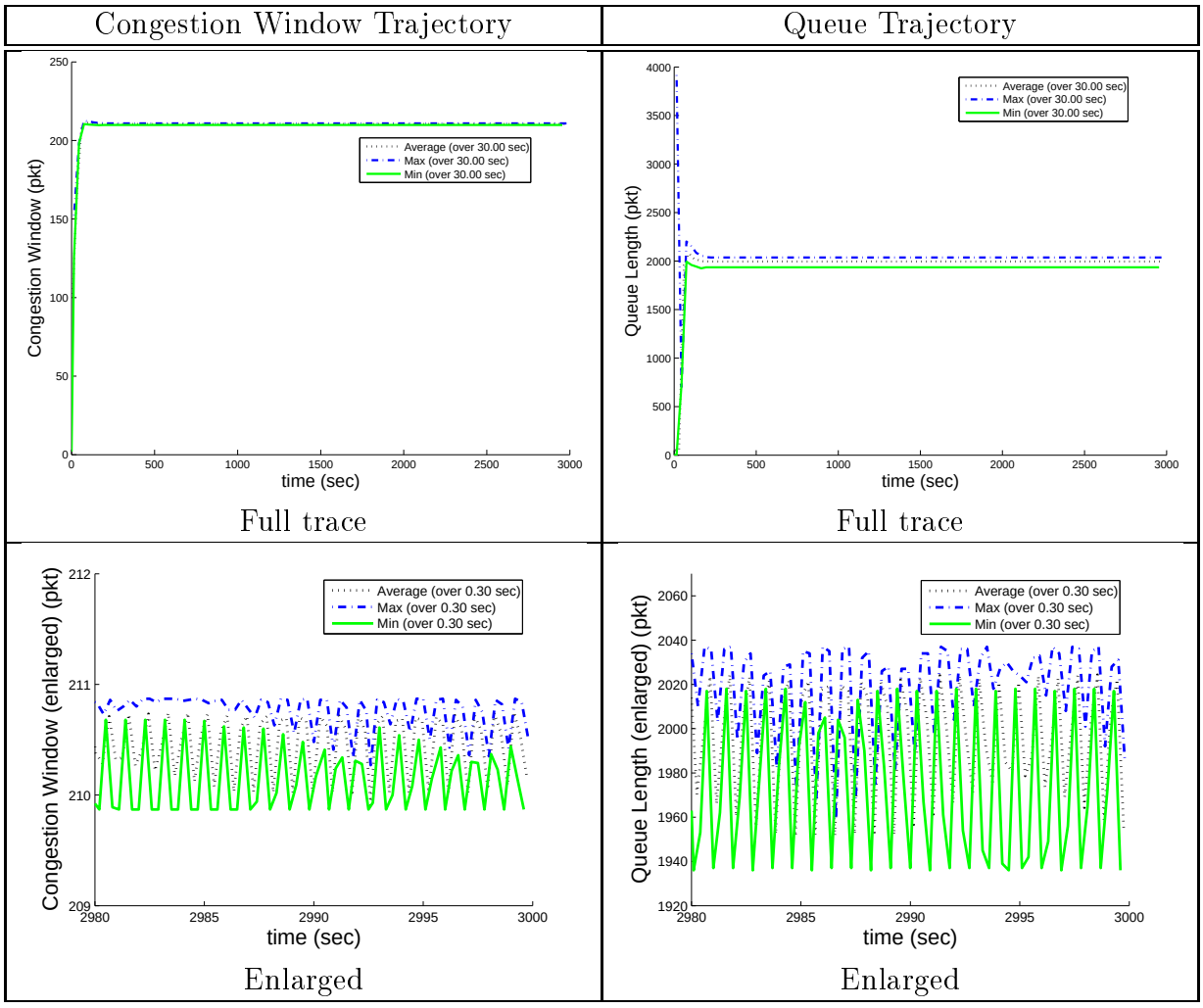


Figure 2.3: 100 Homogeneous TCP Vegas flowssharing a path with a bottleneck capacity of 800Mbps and a propagation delay of 200ms. The packet size in the simulation is 1000 bytes per packet.

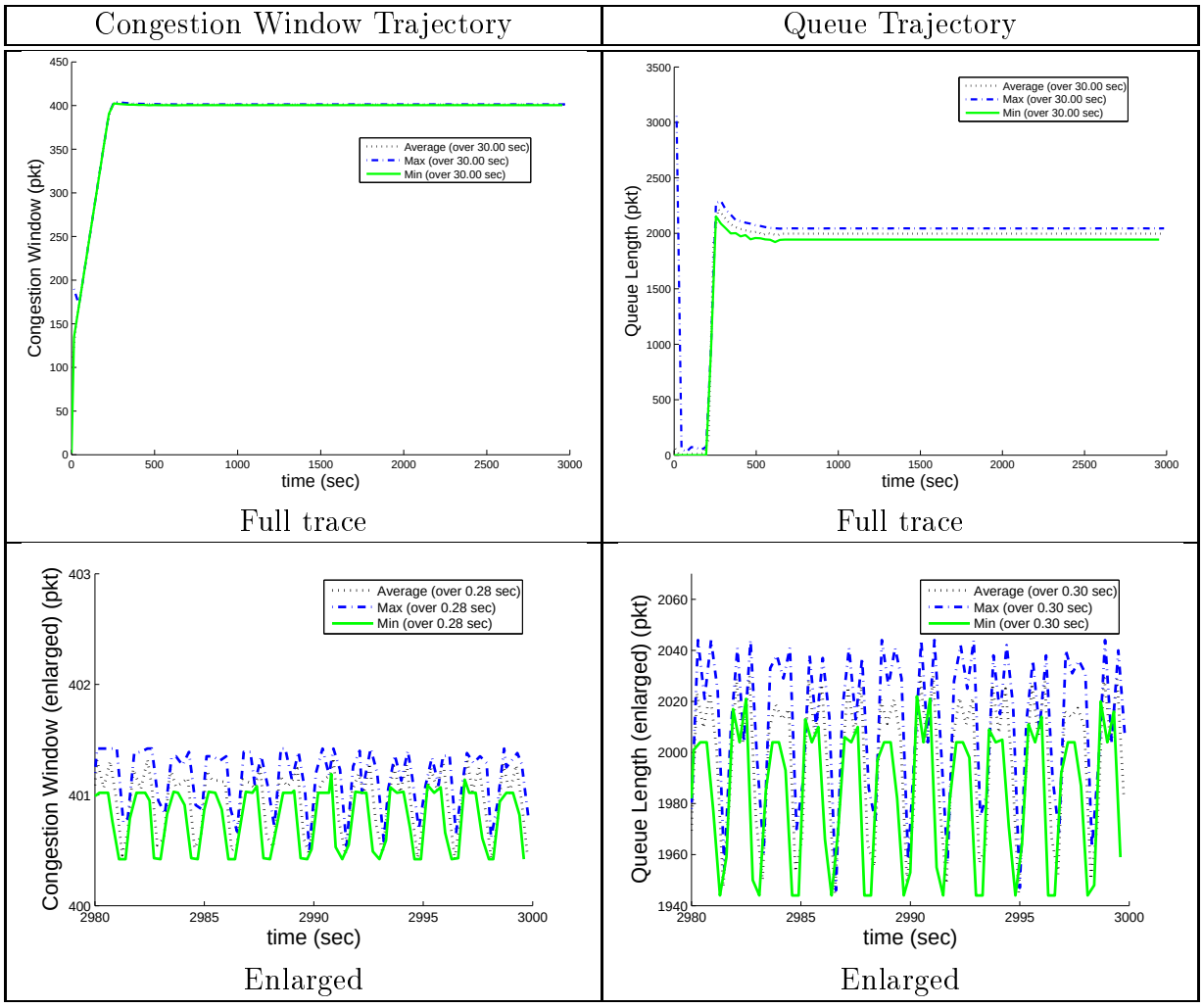


Figure 2.4: 100 Homogeneous TCP Vegas flowssharing a path with a bottleneck capacity of 800Mbps and a propagation delay of 200ms. The packet size in the simulation is 1000 bytes per packet.

$$\Delta w_i = \gamma \left[\frac{w_i\left(t - D_i\left(t\right)\right)}{d_i + q_i\left(t\right)} d_i + \alpha_i - w_i\left(t\right) \right] \tag{2.24}$$

where w_i is the congestion window of source i;

d_i is the round-trip propagation delay of source i;[7]

q_i is the queueing delay observed by source i at time t;

$D_i\left(t\right) = d_i + q_i\left(t\right)$ equals the round-trip time;

γ is the parameter for convergence speed, which is recommended to be $\frac{1}{2}$;

α_i is the parameter for fairness. It specifies the number of packets that each source tries to maintain in the bottleneck queue.

The details of the algorithm can be found in Algorithm 2.

The algorithm assumes the throughput achieved in the last round trip ($\frac{w_i(j)}{D_i(j)}$) to be the available bandwidth, and add α_i packets to the bandwidth propagation delay product.

As extensively evaluated in [37, 17], FAST has achieved much better responsiveness and maintained the same stability as TCP Vegas.

We extend the packet level model to a discrete model for homogeneous flows and analyze its stability in this context.

2.2.2 Model for homogeneous flows

As observed in Theorem 2.1.3.4, the queue can quickly converge to an equilibrium due to micro-burst. We extend this observation to a more general assumption that the queue converges within one RTT so that the stability analysis of congestion window control algorithms can ignore the convergence time of the queue and assume that the queue converges to an equilibrium instantly. This equilibrium can be described as

$$q\left(t\right) = \max\left\{ \frac{\sum_i w_i\left(t\right) - cd}{c}, 0 \right\} \tag{2.25}$$

[7] We use the minimum observed round-trip time as an approximation of d_i. The discussion on the noise of measurement can be found in [37].

Algorithm 2 FAST algorithm

For each source i:

1. Initialization:
   (a) $count_i = w_i$;
   (b) $fastOn_i = 1$
2. On the transmission of each data packet j:
   (a) $w_i(j) = w_i$;
   (b) $s_i(j) = T$. (T is the system time.)
3. On the arrival of each acknowledgment (that acknowledges packet j):
   (a) If $fastOn_i == 1$:
      i. Calculate RTT $D_i(j) = T - s_i(j)$;
      ii. Calculate $\Delta w_i = \gamma\left[\frac{w_i(j)}{D_i(j)}d_i + \alpha_i - w_i\right]$
      iii. If $\Delta w_i \geq 1$: $w_i = w_i + 1$
      iv. If $\Delta w_i \leq -1$: $w_i = w_i - 1$
   (b) $count_i = count_i - 1$
   (c) If $count_i \leq 0$: (One RTT is finished)
      i. $fastOn_i = 1 - fastOn_i$
      ii. $count_i = w_i$

$fastOn_i$ indicates whether $w_i(t)$ needs to be adjusted in the current RTT ; $count_i$ is the counter to detect the end of an RTT.

Based on this assumption, we propose a discrete time model to analyze the stability of homogeneous FAST flows. For the sources, we use a model that assumes that every source makes its decision on discrete time points $\tau_1, \tau_2, \cdots$, we have:

$$w_i(\tau_{k+1}) = F(w_i(\tau_k), q(\tau_k)) \tag{2.26}$$

In the case with homogeneous FAST flows, τ_i corresponds to the number of RTTs the flows have been in the system. Based on (2.25) and (2.26), we can analyze the convergence of a system.

2.2.3 Stability of FAST in homogeneous network

We prove that the congestion window of each source exponentially converges to the equilibrium regardless of capacity, delay and number of flows. In the proof, we only consider the situation when the link is fully utilized. If the link is not fully utilized, queueing delay equals zero. FAST algorithm will always increase the congestion window until the link is fully utilized. Let $q(t)$ denote the queueing delay at the bottleneck router. We can prove that $\exists T > 0, q(T) > 0 \Rightarrow \forall t \geq T : q(t) > 0$, if the network configuration does not change.

By (2.24), the window update function is

$$w_i(t) = \gamma\left(\frac{w_i(t-1)}{q(t-1)+d}d + \alpha_i\right) + (1-\gamma)w_i(t-1) \tag{2.27}$$

Since the bottleneck is fully utilized, by (2.25), we have $\sum_i \frac{w_i(t)}{q(t)+d} = c$. Hence,

$$q(t) = \frac{\sum_i w_i(t)}{c} - d \tag{2.28}$$

Define $W(t)$ to be the sum of windows over all the sources:

$$W(t) = \sum_i w_i(t) \tag{2.29}$$

2.2.3.1 Convergence of the sum of windows

Theorem 2.2.3.1:

By (2.27), (2.28) and (2.29), if $\gamma \in (0,1)$, $W(t)$ is globally stable, and the equilibrium is $\alpha + cd$.

Proofs for all the theorems can be found in Appendix Section of Wei[37].

Theorem 2.2.3.1 shows that $W(t)$ converges to $\sum_i \alpha_i + cd$ exponentially.

2.2.3.2 Convergence of individual flows

Theorem 2.2.3.2:

$\forall \eta > 0$, $\exists T_0$: $\forall t > T_0$, $\left| w_i(t) - \frac{\alpha_i}{\alpha}(\alpha + cd) \right| < \eta$.

Theorem 2.2.3.2 shows that the window size of each individual FAST flow converges to the equilibrium $\frac{\alpha_i}{\alpha}(\alpha + cd)$.

Hence, FAST is globally stable in the case with a single bottleneck link and homogeneous sources.

Chapter 3

Microscopic Effects on Loss-based Congestion Control Algorithms

Both micro-burst and sub-RTT burstiness affect the performance of loss-based congestion control algorithms. This chapter focuses on the effect of sub-RTT burstiness, which is not well understood.

The effect of micro-burst on loss-based congestion control algorithms has been well understood [21, 20, 18]. When the bottleneck buffer size is too small to absorb all the packets in a micro-burst, the bottleneck has to drop some of the packets, even when the average input rate is lower than its capacity. This situation happens in slow-start phase of a loss-based congestion control algorithm. During slow-start, the TCP sources generate micro-bursts of sizes up to half of the maximum window size. If the bottleneck buffer size is not large enough to hold these packets, the TCP sources exit slow start prematurely and take a long time to reach equilibrium. As explained in Section 1.1.1, micro-burst effect is transient and can be eliminated by large buffers. For network with small bottleneck buffers, several algorithms have been proposed to eliminate the micro-bursts. Some examples are TCP Pacing [20, 21], burstiness reduction [18], and burstiness control [22].

The effect of sub-RTT burstiness, however, is less clear. This chapter focuses on the effect of sub-RTT burstiness on loss-based congestion control algorithms. Our study finds that the sub-RTT burstiness has direct impact on *loss synchronization rate*, an important parameter that affects the fairness convergence, friendliness and

link utilization of the loss-based congestion control algorithms. As explained in Section 1.1.2, the effect of *sub-RTT* burstiness is persistent and cannot be eliminated by large buffers.

We proposed a model to understand the relation of *sub-RTT level burstiness* and *loss synchronization rate*. The model takes a signal sampling perspective. The key idea is to view a loss-based congestion control's congestion detection as a sampling process: a TCP flow detects a congestion signal through the loss of its own data packets. Hence, the bursty pattern in the TCP data process directly affects the probability that a TCP flow detects a packet loss in a congestion event. With *sub-RTT level burstiness*, it is very likely that some of the TCP flows do not observe any packet loss in a congestion event. These flows will be more aggressive than those flows that detect the packet loss.

This understanding explains several interesting problems, such as the convergence of loss-based MIMD algorithms, friendliness between bursty TCP and paced TCP, etc. It has also inspired the design of a new link algorithm which significantly increases the loss synchronization rate.

3.1 A model for loss synchronization rate

Most modern loss-based TCP algorithms react to *loss event*s, instead of *individual packet losses*.[1] A *loss event observed by a TCP flow* is defined as a round trip time in which at least one packet loss is detected by the TCP source. The TCP source reduces its congestion window only once for each *observed loss event*, even if there are multiple packet losses in this round trip time. With this process, it is the loss event rate observed by a TCP flow, instead of per-packet loss rate, that affects the performance of a loss-based TCP. *Loss synchronization rate* is introduced to capture the probability that a TCP flow observes a loss event when congestion happens in the router.

[1] These modern TCPs include TCP NewReno [6], FACK TCP [7], HighSpeed TCP [8], Scalable TCP [33], BIC TCP [38], H-TCP [11], CUBIC [10]and etc. The only known TCPs that react to individual packet losses are TCP-Tahoe [4] and TCP-Reno [5].

We focus our study in scenarios with homogeneous flows sharing the same path with a common RTT. In this context, we can define a *loss synchronization rate*.

We define *loss synchronization rate* as the probability that a flow detects at least one loss signal in a *loss event*. A *loss event* is defined as an RTT in which at least one packet is dropped by the bottleneck router due to congestion (buffer overflow). Different flows may have different loss synchronization rates (λ_i). We use $\lambda = \frac{1}{N}\sum \lambda_i$ to denote the average loss synchronization rate among N flows.

The concept of loss synchronization rate was first introduced to model the aggregate throughput and instantaneous fairness (variance of instantaneous rate) in [39]. Many TCP performance analysis have been based on the concept of loss synchronization rate. For example, Bacelli and Hong point out that the short term fairness of TCP flows highly depends on the loss synchronization rate among all TCP flows [39]. Leith and Shorten experimentally demonstrate that loss-based high speed TCPs have very different fairness properties with different synchronization rates [40].

However, there is no clear understanding on the loss synchronization rate itself. Previous studies use different assumptions to model λ. For example, λ is an outside input to the model in [40]. On the other hand, λ_i is modeled as a function of window size w_i in [39] with the assumption that all the packets have the same per-packet loss probability. This assumption is equivalent to the fluid assumption. It is important to have a clear understand on the loss synchronization rate, given the many results based on this concept.

We model the loss synchronization rate with the consideration of *sub-RTT burstiness*. Our model has two major assumptions:

1. The data packet arrival process ($x_i(t)$) of each TCP flow i is bursty in sub-RTT timescale and $x_i(t)$ can be modeled by an on-off process in each RTT.

2. The packet loss process ($l(t)$) is bursty in sub-RTT timescale and $l(t)$ can be modeled by another on-off process in each RTT.

Sub-RTT burstiness in TCP packet arrival processes is well documented. One example of these observations is presented by Jiang and Dovrolis [23].

General burstiness in the packet loss processes is also well-documented [31, 41, 42]. However, in our assumption, we further claim that the loss process is bursty in *sub-RTT* timescale. We support this assumption with evidence from our measurements in NS-2 simulation, Dummynet emulation, and PlanetLab.

Based on these two assumptions, we model the loss synchronization rate as the detection probability using one on-off process (TCP data packets) to sample another on-off process (packet loss process).

Our model predicts that the combination of bursty TCP flows and a drop-tail router (bursty loss process) yields very low and uniform synchronization rates among TCP flows with different congestion window sizes and leads to poor fairness convergence. Our model also suggests that the use of pacing at the TCP sources and/or the use of random dropping algorithms in the link (e.g. RED [31]) can increase synchronization rate.

3.1.1 Burstiness in the packet loss process

We studied sub-RTT level burstiness in the packet loss processes in three different environments: a simulation network (via NS-2 [43]), an emulation network (via Dummynet [44]), and the Internet (via PlanetLab [45]).

From all these three measurement sources, we found significant burstiness in sub-RTT time scales.

3.1.1.1 Measurement

We measured the timing of each packet loss in three different environments: simulation network (NS-2), emulation network (Dummynet), and the Internet (PlanetLab). The NS-2 simulation simulates a single ideal bottleneck shared by heterogeneous sources. The Dummynet system emulates a single bottleneck link shared by heterogeneous sources.[2] The PlanetLab experiments measure the realistic situations in the Internet. For each loss trace, we calculated the time interval between two consecutive

[2]The bottleneck link emulated by Dummynet processes packets in burst of 1ms.

lost packets, called the loss interval, and analyzed the loss processes by plotting the cumulative distribution function (CDF) and the probability density function (PDF) of the loss intervals. We compared the PDF of the packet loss processes to the corresponding Poisson processes with the same average event arrival rates. We observed that the packet loss processes are much burstier than the Poisson processes.

The measurements from NS-2, Dummynet, and the Internet all suggest that the sub-RTT packet loss process is very bursty.

Results in NS-2 Simulation Figure 3.1 shows the CDF of the loss interval in NS-2 simulations. The RTTs of the flows in simulation are random between 2ms to 200ms. From the figure, we observed that 80% of the packet losses cluster within short time periods smaller than 1% of the RTT.

We also plotted the PDF of the loss interval and compared it with the PDF of a Poisson process with the same arrival rate, as shown in Figure 3.1 (B) .

Figure 3.1 (C) zooms in to a small time scale of 0 to 2 RTT and uses log-scale in the Y-axle so that the Poisson process has a straight line in its PDF. Compared to the Poisson process, the loss process is much burstier – more than 10 times the packet losses occurred in the very small time interval.

Results in Emulation Network Figure 3.2 is the CDF of the loss interval in Dummynet emulations. The RTTs of the flows are fixed to 4 classes: 2ms, 10ms, 50ms, and 200ms. The loss interval CDF shows a similar pattern to the NS-2 results, except that the CDF starts from 0.1% of RTT due to the limited time resolution of our measurements in the Dummynet router.

Figure 3.2 (B) and (C) show the PDF of the loss interval. Again, the loss process is much burstier than the corresponding Poisson process.

Results in the Internet Figure 3.3 is the CDF with the Internet measurement.

The Internet measurement shows less burstiness in loss processes than we observed in simulation and emulation. This is due to the heterogeneity of the Internet, in terms

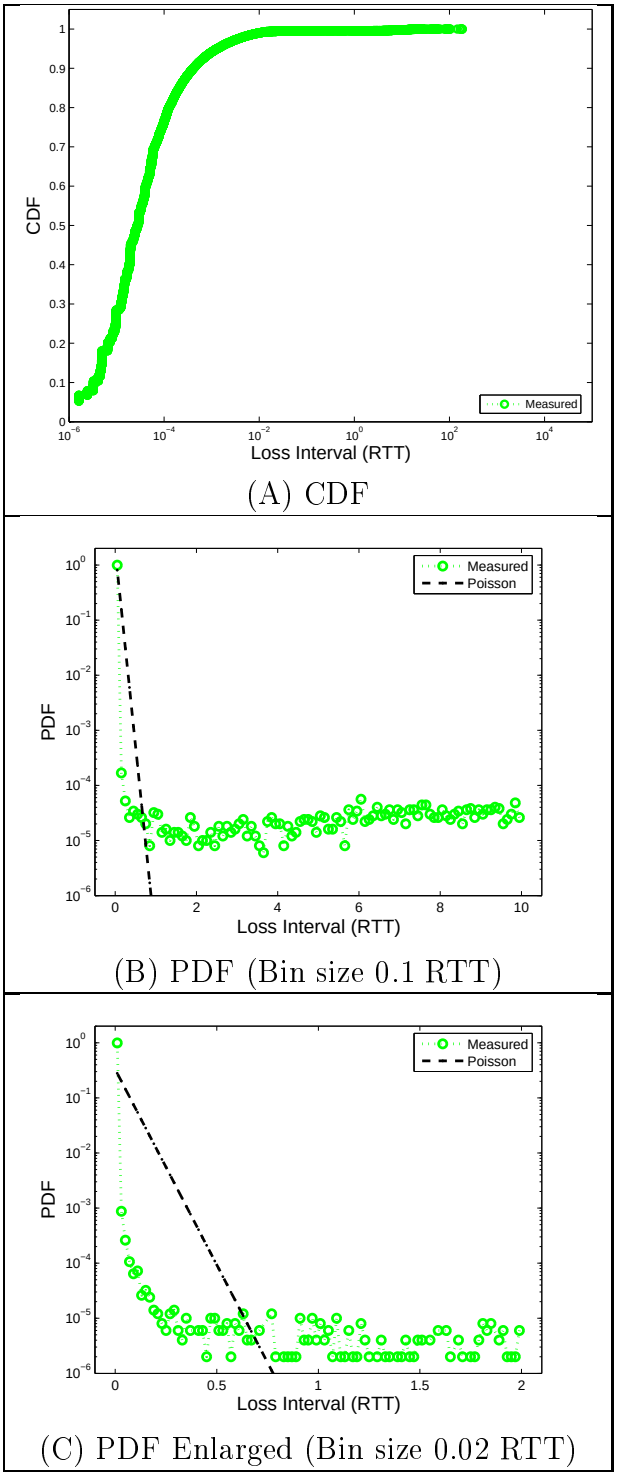
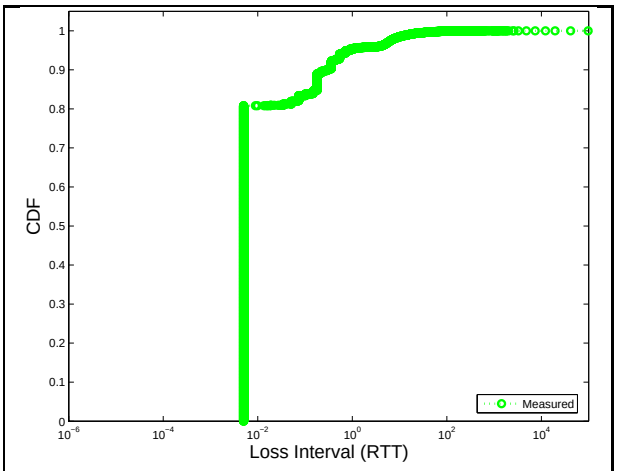
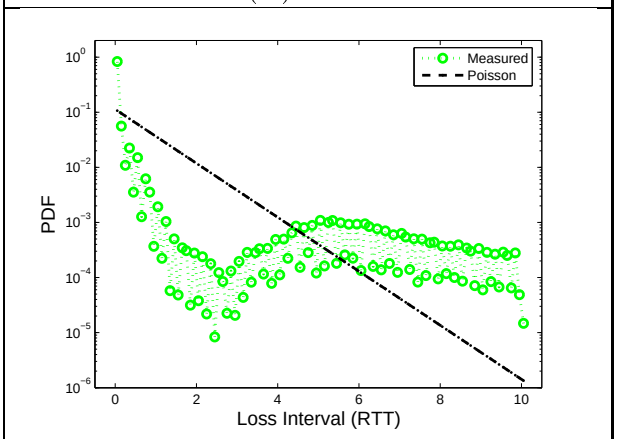


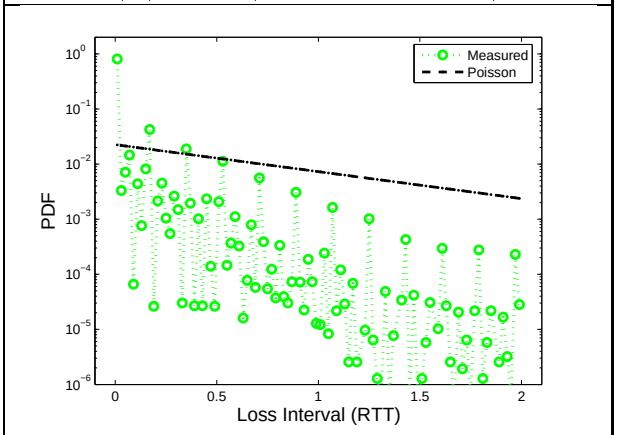
Figure 3.1: Loss intervals in NS-2 measurements.
Note that all the CDF figures in this chapter have X-axles in log-scale, and all the PDF figures in this thesis have Y-axles in log-scale.



(A) CDF

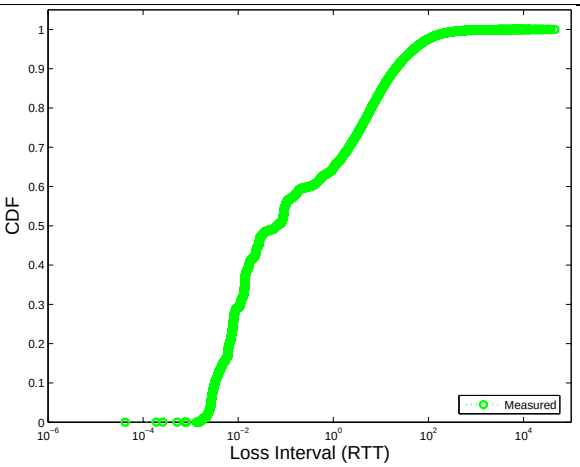


(B) PDF (Bin size 0.1 RTT)

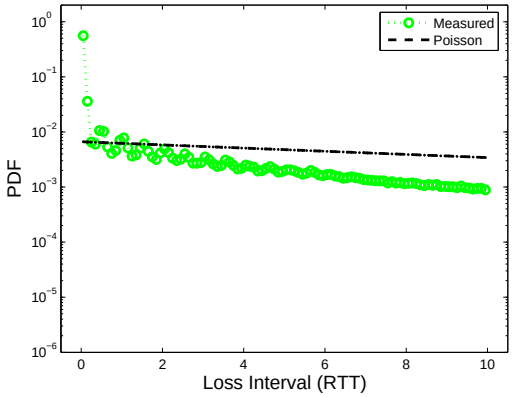


(C) PDF Enlarged (Bin size 0.02 RTT)

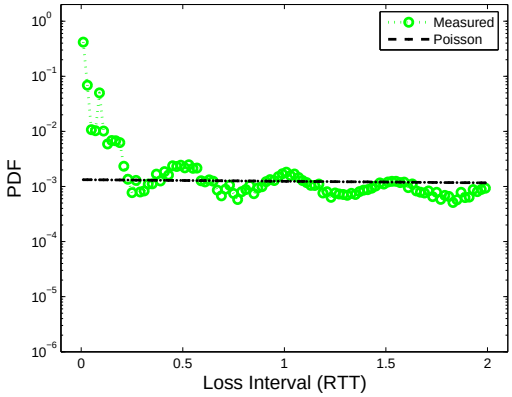
Figure 3.2: Loss intervals in Dummynet measurements.



(A) CDF



(B) PDF (Bin size 0.1 RTT)



(C) Loss intervals in PlanetLab measurements (Bin size 0.02 RTT)

Figure 3.3: Loss intervals in PlanetLab measurements.

of application types, traffic patterns, and queuing delay. In such an extremely heterogeneous environment, we observed that 60% of the packet losses cluster within short time periods of 1 RTT, and 40% of the packet losses cluster within time periods of 1% of RTT. This evidence is still very strong for sub-RTT burstiness in loss processes.

We plotted the PDF in Figure 3.3 (B)(C) and compared the Internet loss process against a Poisson process with the same arrival rate. We observed similar burstiness as in NS-2 and Dummynet. In the smallest interval region (left side), the measured loss process is far burstier than the Poisson process.

3.1.1.2 Possible Sources of sub-RTT Burstiness

As shown by the results of the NS-2 simulations, Dummynet emulations and the Internet measurements, packet loss is highly bursty in sub-RTT timescale. There are several possible sources that lead to such burstiness.

DropTail routers are considered the major source of packet loss burstiness [31]. A DropTail router serves as a FIFO queue, accepting incoming packets until the buffer is full. Working with DropTail routers, loss-based congestion control algorithms keep increasing the data rate when the router's buffer is not full. When the router's buffer is full and packets are dropped, the aggregate data rate is higher than the router's capacity and packet loss persists until the loss-based congestion control algorithms detect the loss of packets and reduce the data rate, usually one half of an RTT later. In between the first packet loss and the reduction of data rate, there is a peak of packet losses in the DropTail router. Some researchers propose introducing randomness in the router. For example, Floyd and Jacobson proposed to randomly drop the packets earlier before the buffer is overflowed [31]. However, these proposals suffer from difficult parameter settings problems.

Slow start of TCP flows is another source of packet loss burstiness. A TCP flow starts with a very small rate in burst (sending two packets back-to-back every round trip), and doubles its data rate if no loss is observed. This process can quickly increase the queue size in the bottleneck buffer in just a few round trips and produce a large number of continuous packet losses in the router. Some new congestion control

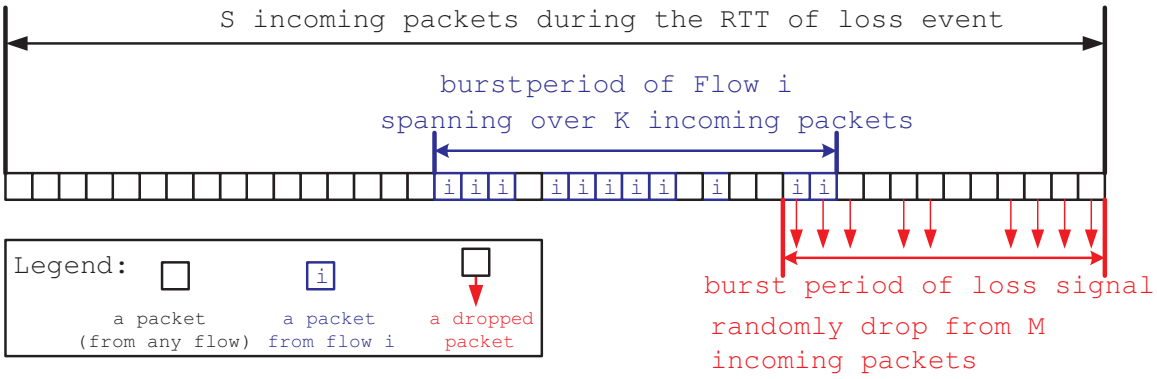


Figure 3.4: Congestion detection within one RTT: a flow uses its data packet process to sample the loss process. The loss synchronization rate is the probability that one of the w_i packets from flow i (distributed over K packets) happens to be one of the L dropped packets (distributed over M packets).

algorithms, such as QuickStart [46] and RCP [47] have been proposed to avoid such aggressive detection. These algorithms require changes in data packet formats, which are expensive for the existing infrastructure.

Hence, the sources of sub-RTT burstiness in packet loss processes will exist in the foreseeable future.

3.1.2 Modeling loss synchronization rate

Assuming that the loss process $l(t)$ only depends on the *aggregation* of transmission rates from all flows and is independent of the packet transmission process from an *individual* flow i in sub-RTT time scales, the signal sampling perspective leads to a simple model for sub-RTT time scale behavior, as shown in Figure 3.4.

The figure illustrates all packets going through the bottleneck router in the RTT of a congestion event. S is the number of these packets. These packets include packets that are accepted by the bottleneck and packets that are dropped by the bottleneck.

All the packets from an individual flow (flow i) are distributed in flow i's burst period, which spans over K incoming packets. Each of these K packets has a probability of $\frac{w_i}{K}$ to be from flow i and the total number of packets from flow i in this RTT is w_i on average.[3] Since i can be any of the N flows, we assume that the position of

[3]For both packet transmission process and loss process, we use Poisson arrival assumption to

the burst period of flow i is randomly distributed in the RTT. That is, the starting position of the burst period can be any of the S packets. If the burst period starts at the end of S, wrap-around is allowed.

We model the loss signal process as another on-off process, with the burst period spanning over M incoming packets, dropping L packets on average, with a dropping probability of $\frac{L}{M}$. If at least one of the w_i packets from flow i happens to be one of these L dropped packets, flow i detects the loss event and back off its congestion window. Otherwise, flow i is not aware of the loss event and continues to grow its congestion window.

From this perspective, the synchronization rate of flow i (λ_i) is the probability that one of the w_i packets happens to be one of the L dropped packets, as the position of flow i's burst period is randomly distributed in the RTT; that is $\lambda_i = P(\text{hit}_i)$ where hit_i is the event that flow i detects the loss signal.

Let the loss signal burst (M packets) and packet transmission process burst (K packets) intersect over k incoming packets ($\max\{0, M+K-S\} \leq k \leq \min\{M, K\}$). Conditioning on k, we have the probability of a packet from flow i getting dropped, given k packets are in the intersection:

$$P\left(\text{hit}_i | k\right) = 1 - \left(1 - \frac{L}{M}\frac{w_i}{K}\right)^k \tag{3.1}$$

and since the position of packet transmission process burst (K) is randomly dis-

simplify the descriptions. With Poisson arrival, the number of data packets is not always w_i. A more complicated computational model can be obtained with the assumption that the w_i and L packets are uniformly distributed over K and M incoming packet slots. We use Poisson model in our computations due to its simplicity and reasonable accuracy. However, we note that the model is not accurate if w_i or L is very small.

tributed in the RTT with modulo S, we have

$$P(k) = \left\{ \begin{array}{cc} 0 & \text{if } k > \bar{k} \text{ or } k < \underline{k} \\ \frac{2}{S} & \text{if } \underline{k} < k < \bar{k} \\ \frac{\max\{M,K\}-\bar{k}+1}{S} & \text{if } k = \bar{k} \text{ and } \underline{k} < \bar{k} \\ 1 - \frac{\max\{M,K\}+\bar{k}-2\underline{k}-1}{S} & \text{if } k = \underline{k} \text{ and } \underline{k} < \bar{k} \\ 1 & \text{if } k = \underline{k} = \bar{k} \end{array} \right\} \tag{3.2}$$

where $\underline{k} = \max\{0, M + K - S\}$ is the lower-bound of k and $\bar{k} = \min\{M, K\}$ is the upper-bound of k.

Hence,

$$\lambda_i = P(\text{hit}_i) = \sum_{k=\underline{k}}^{\bar{k}} P(\text{hit}_i|k) P(k) \tag{3.3}$$

There is no simple close form for the above formula. However, the formula reveal a good property of the loss synchronization rate: the dependency of loss synchronization rate on M and K are symmetric. Hence, changing the M and K have similar effects on loss synchronization rate.

We used MatLab to compute the values of λ based on (3.1), (3.2) and (3.3). Since the packet loss is due to congestion, the DropTail router's buffer must have been full. Hence, the number of packets going through the router in this RTT is approximately $S = cd + B + L$ (packets in flight in the path + packets in the buffer + packets that are dropped by the bottleneck). With N Reno flows in congestion avoidance state, at most N additional packets are transmitted in this RTT in comparison to the last RTT, in which no loss happens. Hence, at most N packets are dropped by the DropTail router. That is $1 \leq L \leq N$.[4]

Figure 3.5 shows the computational results of the model, with parameters $L = N = 32$, $cd + B + L = 2000$ and $w_i = \frac{2000}{32}$. This is roughly equivalent to the scenario of 32 Reno flows sharing a path of 200ms delay and a bottleneck with a capacity of 100Mbps and a buffer size of 400 packets.

[4]If the flows are not controlled by Reno, the number of loss packets may be larger. A general AIMD algorithm with additive parameter of α [48] will have $1 \leq L \leq \alpha N$.

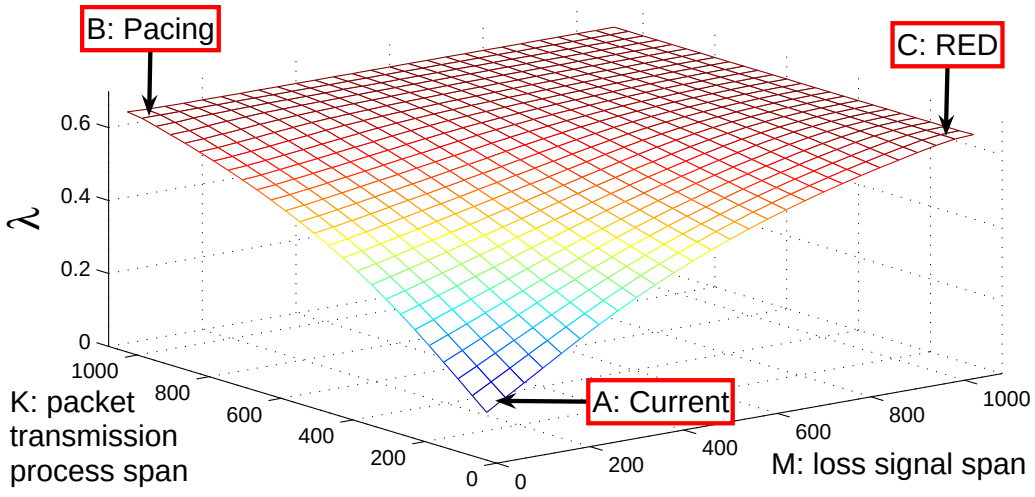


Figure 3.5: Synchronization rate: computational results from the model

In Figure 3.5, the synchronization rate hits its lowest point (A) when M and K are small, corresponding to the case in which both loss signal and data process are bursty in sub-RTT level. This is the current situation: we have TCP senders and DropTail routers, which send and drop packets in bursty patterns.

3.1.3 TCP Pacing and RED

As K increases in Figure 3.5, the loss synchronization rate increases. Its value hits a high point (B) when $K = cd + B + L$ (upper-left point in Figure 3.5), corresponding to the case in which the data packets of each flow spread out over the whole RTT. This is the situations with improvements in TCP sender, such as pacing [20, 21, 49].

Figures 3.6 and 3.7 intuitively illustrate the change in synchronization rates from point A to point B in the loss sampling perspective.

Figure 3.8 presents details of the loss signal process and the data packet processes in a simulation, with TCP and with a pacing improvement. In the simulation, we used a randomized version of pacing algorithm to reduce phase effects. The detailed algorithm can be found in Appendix 6.2.

A green dot $(t, i), i = 1 \cdots 16$ in the figure represents a packet from flow i going

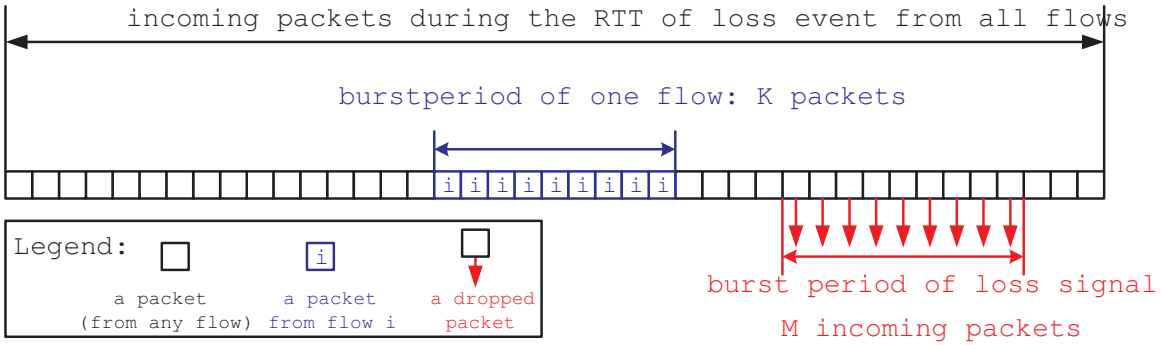


Figure 3.6: Packet loss with window-based implementations

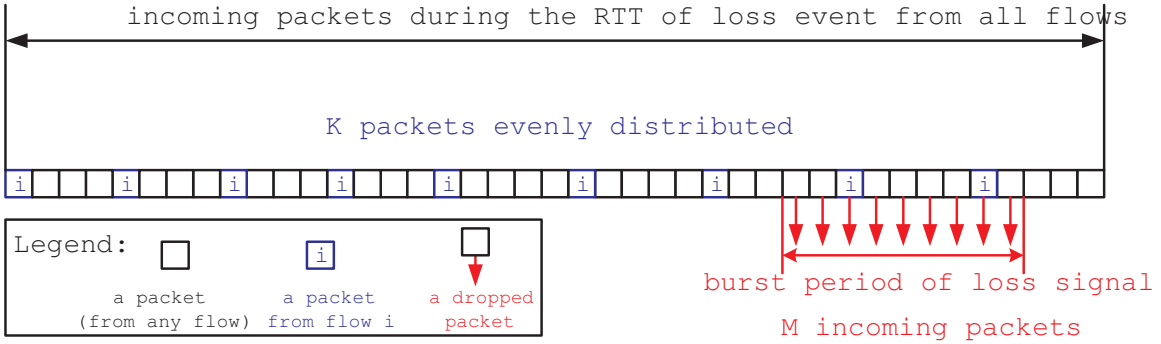
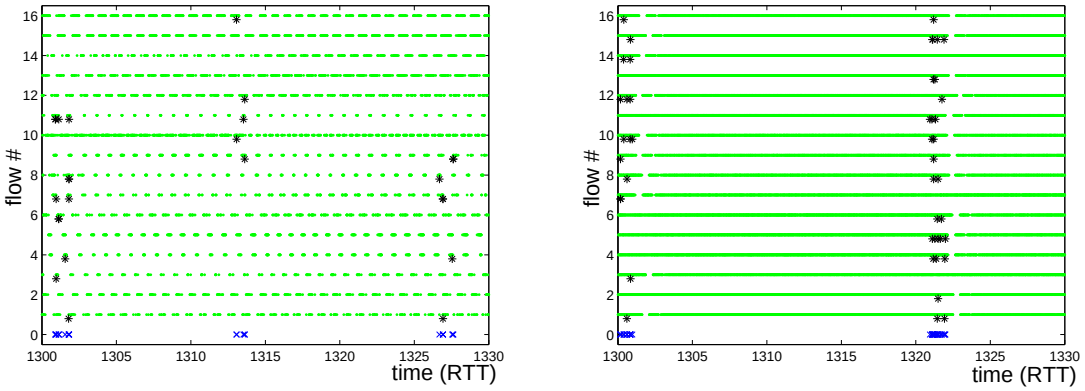


Figure 3.7: Packet loss with rate-based implementations



(a) TCP (b) TCP improved by pacing

Figure 3.8: Sampling effects of TCP and pacing (simulation results)

through the bottleneck at time t; a black star $(t, i), i = 1 \cdots 16$ represents a packet of flow i dropped at time t; a blue cross $(t, 0)$ at the bottom of the figures represents a packet (of any flow) dropped by the bottleneck. We collected 30 RTTs of the data after the flows ran for more than 1000 RTTs, so the flows were in congestion avoidance phase for a long time. In both cases, the bottleneck link was fully utilized and the aggregate throughputs in both cases were similar. Hence, when we compare these results, we see that the effect of sub-RTT level burstiness is still very significant.

In Figure 3.8(a), the packets are sent by TCP. The transmission processes of most flows clearly show a bursty on-off pattern. When some packets are lost in a burst, only a few flows (30% in this case), whose burst periods happen to cover the loss burst, detect the congestion signal.

In Figure 3.8(b), the packets are paced out equally so that they are evenly distributed throughout the whole RTT. When some packets are lost in a burst, most of the flows (70% in this case) experience the loss and thus detect the congestion signal.[5]

Symmetrically, as M increases, the loss synchronization rate increases, too. Its value hits a high point (C) when $M = cd + B + L$, corresponding to the case in which packet losses are spread out over the whole RTT. This is the situation with link algorithm improvements such as Random Early Detection (RED) [31]. Our simulation and trace analysis confirm that RED increases the loss synchronization rate among Reno flows to 0.5 to 0.6.

3.1.4 Validation

We measure the synchronization rate from our simulations and compare the results to the computation results based on equation (3.1), (3.2) and (3.3). The simulations have a setup with a bottleneck of 100Mbps and a round trip propagation delay of 200ms. The bottleneck buffer size is 1680 packets. Hence, $cd+B+L \approx 3340$. We vary the number of flows N from 2 to 32. In the computation, we assume $w_i = \frac{cd+B+L}{N}$

[5]Consequently, the length of loss-epochs is shorter in Figure 3.8(a) since less flows reduce their congestion windows in a loss event. This is consistent to the analysis in studies by Baccelli, et al [39].

and $L = N$. In the measurement, we take the first packet loss that is not part of any previous loss events as the beginning of a new loss event and consider all the subsequent packet losses within one round-trip time as in the same loss event. We average the loss synchronization rates of all flows and present the average values.

Figure 3.9 compares the computational results and the measurement from NS-2 simulations. Figure 3.9 (A)~(C) correspond to the three points in Figure 3.5.[6] This shows that our model can qualitatively estimate the loss synchronization rates.

3.1.5 Asymptotic results

Although the general formula (3.1), (3.2) and (3.3) are complicated, simple and interesting asymptotic results can be obtained for two special cases (point A and point B in Figure 3.5) with the additional assumption that the number of flows is large.

If TCP packet process is bursty and N is large, $\frac{w_i}{cd+B+L}$ is very small and $L >> w_i$ since $L \sim N$. (3.1) and (3.2) can be simplified into:

$$P(k) = \left\{ \begin{array}{cc} 0 & \text{if } k > w_i \\ \frac{2}{cd+B+L} & \text{if } 0 < k < w_i \\ \frac{L-w_i+1}{cd+B+L} & \text{if } k = w_i \\ 1 - \frac{L+w_i-1}{cd+B+L} & \text{if } k = 0 \end{array} \right\}$$

and

$$\begin{aligned} \lambda_i &= \frac{2}{cd+B+L}(w_i - 1) + \frac{L - w_i + 1}{cd+B+L} \\ &\approx \frac{L-1}{cd+B+L} \end{aligned} \tag{3.4}$$

In this case, flows with different congestion windows see similar synchronization rates since λ_i is almost independent of w_i.

When pacing is applied with DropTail routers, we have $K \approx cd + B + L$ and

[6] In Figure 3.5(C), the theoretic loss synchronization rate with RED is almost flat. We note that there is inaccuracy in the calculation of loss synchronization rate for the case where N=2. In this case, L=N=2 is very small and the Poisson assumption is inaccurate, as explained in Footnote 3.

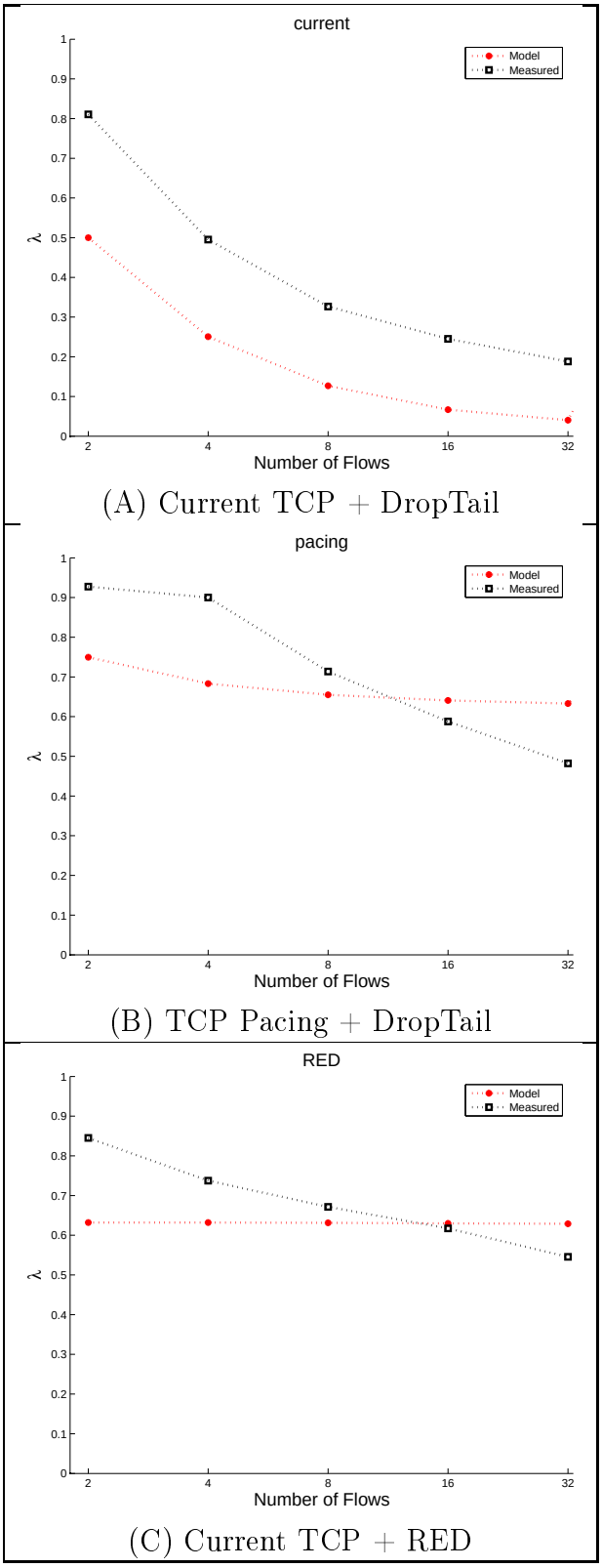


Figure 3.9: Synchronization rate with current TCP, TCP Pacing and RED

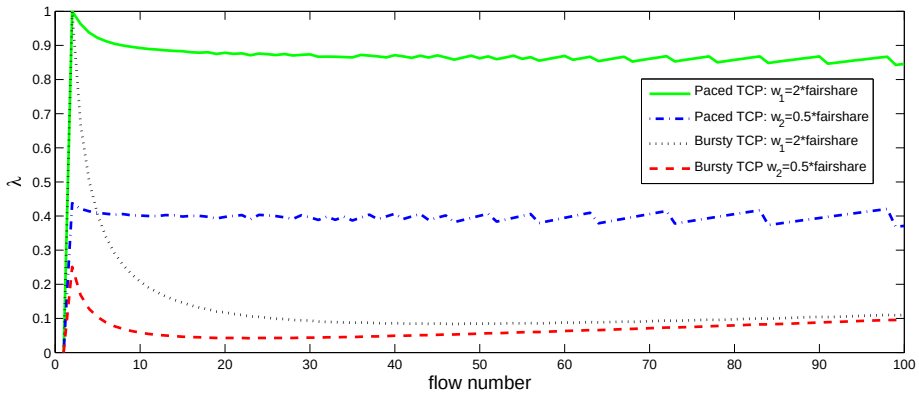


Figure 3.10: Synchronization rates of two flows with different window sizes, among N flows (N=2 to 100), with bursty TCP or paced TCP (MatLab results).

$M \approx L$. Hence,

$$P(k) = \left\{ \begin{array}{ll} 1 & \text{if } k = L \\ 0 & \text{else} \end{array} \right\}$$

and

$$\lambda_i = \left(1 - \left(1 - \frac{w_i}{cd + B + L}\right)^L\right) \tag{3.5}$$

If N is large, $\frac{w_i}{cd+B}$ will be very small and we have

$$\lambda_i \approx \frac{w_i L}{cd + B + L} \tag{3.6}$$

That is, the flows with larger congestion windows see higher synchronization rates.

Figure 3.10 shows the synchronization rates of two flows (among N flows) with different congestion window sizes (w_1 and w_2). $w_1 = 2\frac{cd+B}{N}$ is double the fair share window size and $w_2 = \frac{cd+B}{2N}$ is half of the fair share window size. All other parameters, except flow number and window sizes of flow 1 and flow 2, are the same as in Figure 3.5. With bursty TCPs, flow 1 and flow 2 have similar loss synchronization rates, and hence, see similar loss event rates, as the number of flows increases. With paced TCPs, flow 1 always sees higher loss event rates than flow 2. As we will show in Section 3.2.2, this asymptotic result has very interesting implication on the fairness of MIMD (Multiplicative-Increment-Multiplicative-Decrement) algorithms.

3.2 Implications on Performance of Loss-based TCP

Our model points out three important implications for loss-based TCP flows with a DropTail router:

1. The current implementation has a low synchronization rate due to the *sub-RTT burstiness* introduced by *ack-clocking*;

2. Asymptotically, the loss synchronization rates of flows with different congestion windows tend to be the same, due to the *sub-RTT burstiness* introduced by *ack-clocking*;

3. TCP pacing will see a higher synchronization rate since its data packet arrival process is smooth and is able to detect loss more efficiently.

These predictions have realistic impacts in the system performance.

3.2.1 Fairness convergence

Fairness convergence is a metric that is of interest to the cluster computation industry. In cluster computation, the data transfer time scale is usually measured in seconds. In these scenarios, rate fluctuations in one or two RTT are acceptable, and hence, the traditional short-term fairness definition is not suitable in this case. In this time scale, we are more interested in how fast the TCP flows can share the bottleneck, both efficiently and fairly, in term of average rates over the convergence period.

3.2.1.1 Definition of Fairness Convergence Time

To quantify the fairness convergence, we introduce the notion of fairness convergence time. Fairness convergence time measures *how fast* the TCP flows converge to their fair shares from start up. We give our formal definition of fairness convergence time as the time taken by the slowest flow to reach the fairshare rate as

$$F = \min \left\{ t \left| \forall \tau > t, \min_i \left\{ \bar{x}_i \left(\tau \right) \right\} > 0.8 x_i^* \right. \right\} \tag{3.7}$$

where $x_i^* = \frac{c}{N}$ in our homogeneous setup is the fairshare rate for flow i and $\bar{x}_i(\tau)$ is the average throughput for flow i during the first τ seconds, defined as

$$\bar{x}_i(\tau) = \frac{1}{\tau}\int_0^\tau x_i(u)\,du$$

in which τ is the averaging interval.[7]

This metric measures how long a user has to participate in the data transfer until he or she can enjoy a sense of fairness (by getting 80% of his/her fair share bandwidth). The metric has a small value only if all flows quickly converge to, and maintain, the desired equilibrium in which they share the bottleneck both *efficiently* and *fairly*. The metric has a large value if the bottleneck is not efficiently used (underutilized), or if the flows are sharing the bottleneck unfairly, or if the flows fail to maintain the desired equilibrium in long run.

The fairness convergence time also provides an upper bound for the data transfer latency of parallel flows in cluster applications. If each of the parallel flows needs to transfer a data chunk of D bits, the completion time of all flows as a whole will be at most $F + \frac{D}{0.8\frac{c}{N}}$, since the definition guarantees that each TCP flow achieves 80% of the fair share bandwidth on average at or after time F.

3.2.1.2 Loss Synchronization Rate and Fairness Convergence

Baccelli and Hong point out that the short term fairness highly depends on the *loss synchronization rate* among all TCP flows [39]. With similar derivation, a lower bound of fairness convergence time for TCP Reno is

$$F \geq \max\left\{0, \frac{\log 0.2 - \log\left(1 - \frac{1}{(2-\lambda)}\right)}{\log\left(1-\frac{\lambda}{2}\right)}\frac{\lambda\,(cd+B)}{2N}\right\} \tag{3.8}$$

assuming that the synchronization rate is the same for all flows.

The detailed derivation of (3.8) can be found in Wei, et al [50]. The lower bound

[7] If τ is infinitely large, $\bar{x}_i(\infty)$ is the asymptotic average rate, which is proved to be x_i^* in long-term fairness for most TCPs.

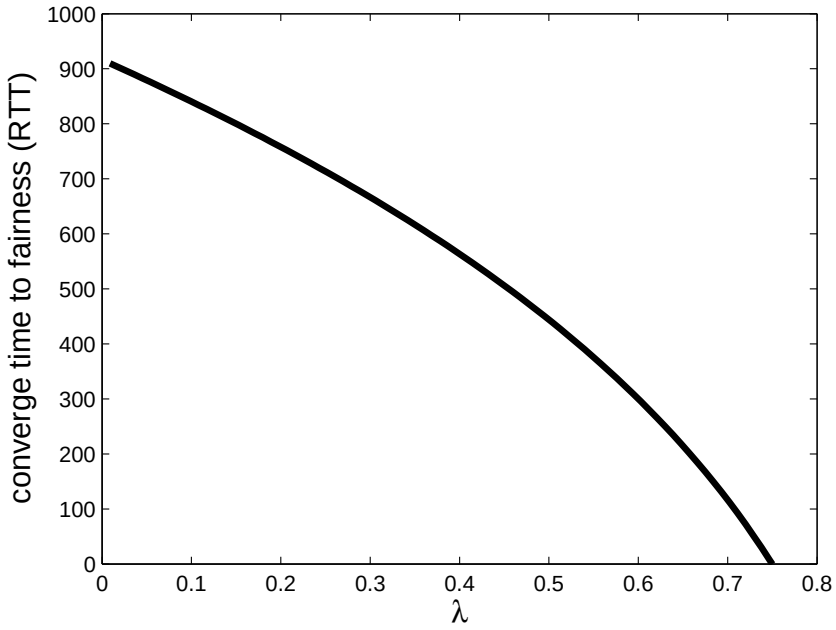


Figure 3.11: Relation between fairness convergence time F and synchronization rate λ (MatLab results)

for TCP-Reno in (3.8) can also be extended to general AIMD TCP algorithms [48], HS-TCP [8], and S-TCP [9]. The derivation is based on the AIMD model used by Baccelli and Hong in [39]. Similar conclusions can be reached with an extended model proposed by Shorten, et al [51]. Note that $\frac{cd+B}{N}$ is a TCP flow's average window size upon a loss event, which depends on the network condition and user pattern which are controlled by TCP. Loss synchronization rate λ is the parameter that we can control.

To intuitively illustrate the relation between fairness convergence time and loss synchronization rate, Figure 3.11 shows the computational results of F as a function of λ, according to (3.8) with $\frac{cd+B}{N} = 1000$ packets.

Figure 3.11 clearly shows that the loss synchronization rate has a significant impact on the fairness convergence. As we discussed in Section 3.1, loss synchronization rate can be controlled by sub-RTT level TCP behavior in the source and by the packet dropping behavior on the link. One can control and increase the loss synchronization rate to achieve better short-term fairness.

Reno	HS-TCP	S-TCP
0.2042152591	0.2496234482	0.2331014195

Table 3.1: Average loss synchronization rates of TCP with a DropTail router

3.2.1.3 Fairness convergence with bursty TCP and DropTail Routers

Using the definition of fairness convergence time, we examined the fairness convergence time of TCP-Reno (Reno), HighSpeed-TCP (HS-TCP [8]) and Scalable-TCP (S-TCP [33]).

According to the analysis in Section 3.1, bursty TCP have low loss synchronization rates when they share a DropTail router. Hence, we expect that the fairness convergence time under TCP and DropTail routers/switches is very long. Our simulation results confirmed our expectation, as shown in Figure 3.12.

Figure 3.12 presents the fairness convergence times with parallel Reno, HS-TCP, or S-TCP flows sharing a 200ms path with a bottleneck capacity of 100Mbps. The results are from NS-2 simulations. All the convergence time measurements are presented in the unit of RTTs. Figure 3.12(a) shows that Reno takes more than 1500 RTTs to converge to its fair share. More interestingly, such slow convergence is *neither improved by increasing bottleneck buffer size, nor by increasing the number of parallel flows.*

Figures 3.12(b) and (c) further show that the fairness convergence is not improved by the recent new loss-based TCP proposals such as HS-TCP and S-TCP.[8]

In the simulation, we measured the loss synchronization rate. Table 3.1 shows the measurements with Reno, HS-TCP and S-TCP flows, averaged over all loss events in the simulations. The measurement results confirm the correlation between long fairness convergence time and low loss synchronization rate.

[8]With HS-TCP or S-TCP, the fairness convergence is actually worse, due to their congestion control dynamics. We only use Reno, HS-TCP and S-TCP as examples throughout this paper as their control structures are cleaner for understandings. Li, et al also show that many other high speed TCP proposals experience long convergence time [34].

As explained in Section 4.2.5, the noise in our simulations in previous sections are heavy-tail on-off traffic with a fixed sending rate in *on* period. Usually, the *on* period lasts for more than one RTT. Hence, the noise serves as some paced flows that cuts the bursty Scalable-TCP into smaller burst. This helps Scalable-TCP to converge with the cases in our previous sections.

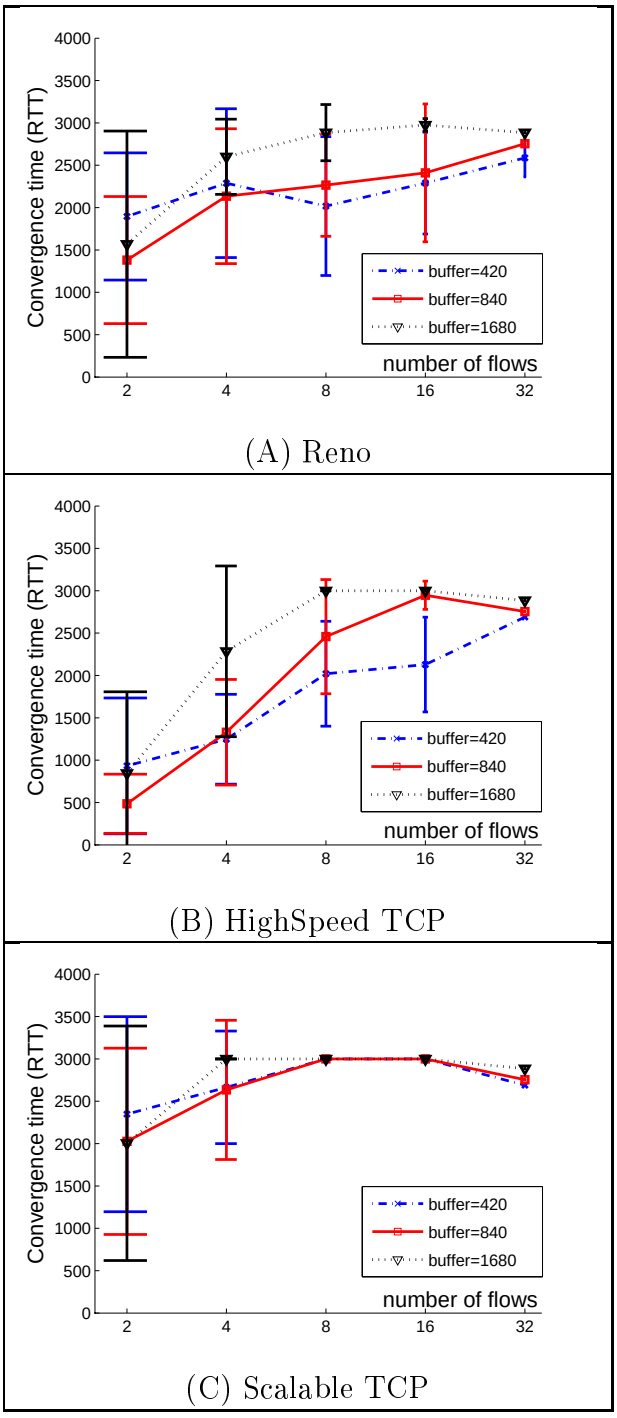


Figure 3.12: Convergence time of different TCPs in simulations with different number of flows and different buffer sizes (in packets).

The average loss synchronization rates of these TCP flows are around 0.2 to 0.25. Such low loss rates can substantially affect the short-term fairness of TCP, as depicted in Figure 3.11. The low loss synchronization rate also shows that there are opportunities to improve TCP short-term fairness. If we can move the synchronization rate λ from 0.2 closer to 1, we can significantly reduce the convergence time and improve the TCP fairness experienced by the real applications.

3.2.2 Convergence of MIMD algorithms

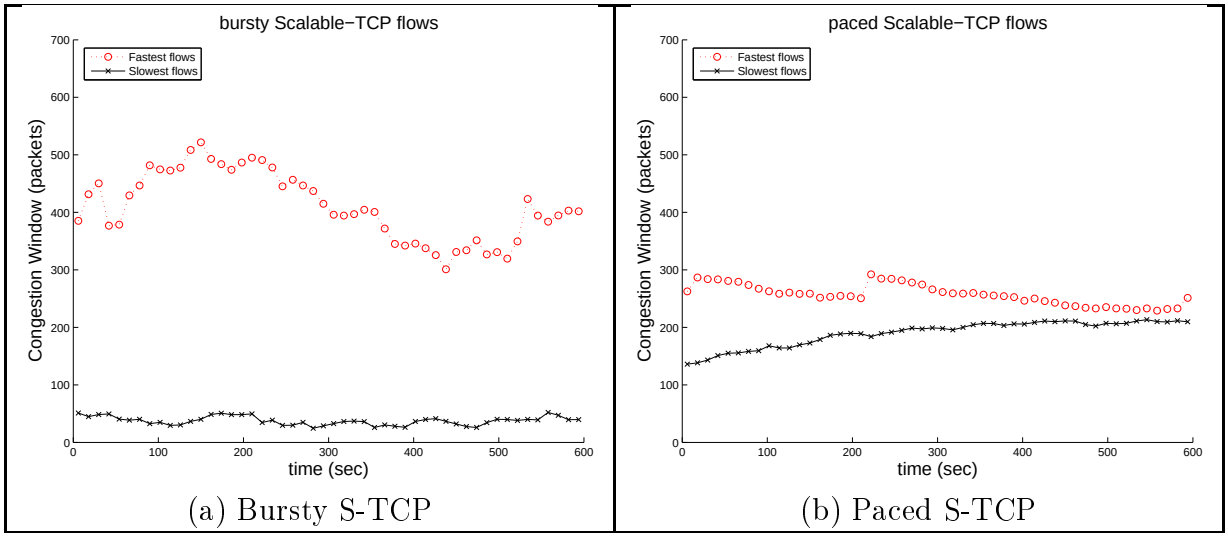
Multiplicative-Increment-Multiplicative-Decrement (MIMD) algorithms are a class of control algorithms which increase and decrease the congestion window by ratio. When MIMD algorithm do not observe congestion, they increase the congestion window by a small percentage. When MIMD algorithms observe congestion, they decrease the congestion window by a large percentage.

Chiu and Jain prove, with a static model, that two MIMD flows with different window sizes cannot converge to a fairness point [32]. This static model assumes that all MIMD flows observe the same congestion event. This assumption is equivalent to $\lambda_i = \lambda$ in which λ is a constant independent of window size w_i of the flow.

However, Kelly proves with the fluid model that Scalable-TCP, an MIMD algorithm, can converge to fairness [33] . The assumption is equivalent to $\lambda_i \propto w_i$.

Our asymptotic results in Section 3.1.5 explain the different conclusions from [32, 9]. As shown in equation (3.4) and illustrated in Figure 3.10, with bursty TCP, λ_i is proportional to window size w_i only when the number of flows N are very small. As N increases, λ_i among flows with different window sizes quickly converges to a very similar value. Hence, the fluid model prediction by Kelly [33] is more accurate when N is very small, and the static model in [32] is more accurate when N is large.

This result has an interesting implication in the fairness convergence of Scalable TCP [9], an MIMD algorithm. Two flows usually cannot converge to fairness with bursty TCP, as pointed out by Leith and Shorten [40]. This effect is particularly significant when there is no cross traffic to pace out the TCP bursts.



(a) Bursty S-TCP (b) Paced S-TCP

Figure 3.13: Convergence of S-TCP: congestion window trajectories of the fastest flow and the slowest flow

On the other hand, the asymptotic result in equation (3.6) suggests that the synchronization rate is always proportional to the window size if pacing is deployed. In this case, Scalable-TCP can converge.

Figure 3.13 shows the congestion window trajectories of the fastest and slowest flows in a case study. In this case, 8 Scalable-TCP flows share a 100Mbps bottleneck link, without noise traffic. Each point of the congestion window size is an average value over 10 seconds. Clearly, Scalable-TCP does not converge in Figure 3.13(a), as reported by several literatures [52, 53]. With pacing, Scalable-TCP converges. Scenarios with different number of flows show similar effects.

Figure 3.14 is a summary of Scalable TCP fairness with different number of flows. We run N Scalable-TCP flows for 600 seconds and calculate, in each case, the ratio between the throughput of the smallest flow and the fair share throughput $\frac{c}{N}$. The larger the ratio, the better fairness among the flows. As predicted by the model, bursty Scalable TCP flows are fair only when the number of flows is small. As N increases, the fairness quickly degrades. With pacing, Scalable TCP's fairness is much improved.

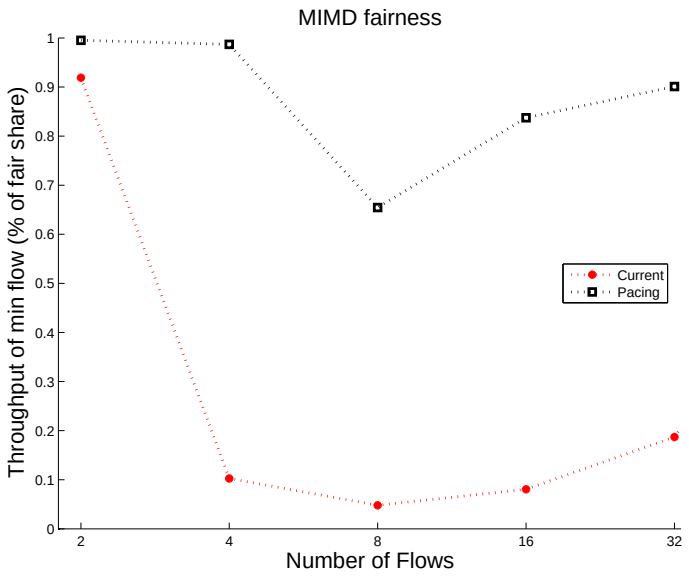


Figure 3.14: MIMD fairness

3.2.3 Performance of TCP Pacing

The performance of TCP pacing has been a controversial topic since the introduction of TCP pacing in late 1990s. On one hand, simulation results presented by Kulik, et al show that TCP pacing can significantly improve the throughput of TCP flows in networks with large capacity, long delay and small buffer [20]. Simulations by Hong show that TCP pacing does improve TCP performance in both efficiency and fairness [21]. On the other hand, Aggarwal, et al show that TCP pacing might actually have lower average throughput in many cases [35].

Our model shows that TCP pacing eliminate the sub-RTT burstiness and increases the loss synchronization rate. As shown in [39], this increment in the loss synchronization rate has two-sided effects on TCP performance. On one hand, the increased loss synchronization rate improves fairness; on the other hand, the increased loss synchronization rate decreases the aggregate throughput of TCP Reno. These two-sided effects explain the different conclusions in the past discussions.

3.2.3.1 Aggregate Throughput

A very important observation presented in other literature is that the aggregate throughput of paced TCP may be lower than the bursty TCP even in isolated scenarios, due to synchronization effects[35]. This is especially true with large numbers of Reno flows when working with small buffers.

Baccelli and Hong give an explanation with synchronization rate [39]. A direct application of equation (7) in [39] shows that Reno's throughput in the worst case (fully synchronized flows sharing a bottleneck with an infinitely small buffer) is 75% of the capacity.[9] Hence, the throughput loss due to synchronization can be up to 25%.

However, such throughput degradation is largely alleviated by the new congestion control algorithms. We can calculate an upper bound of throughput loss due to synchronization with different loss-based congestion control algorithms similar to the work by Baccelli, et al [39].

Assume that buffer size B is infinitely small and all N flows are fully synchronized. In this worst case, each flow's behavior is exactly the same, equivalent to a single TCP flow using a bottleneck with a capacity of $\frac{c}{N}$ and a buffer size of $\frac{B}{N}$. Hence, we can estimate the aggregate throughput of N synchronized flows by the throughput of a single TCP flow. Also, since TCP oscillates in every loss epoch with the same pattern, we only need to calculate the aggregate throughput loss in one loss epoch.

HS-TCP can be approximated by a general AIMD around the equilibrium. The congestion window at the end of a loss epoch is $\bar{w} = \frac{cd+B}{N}$ for the single flow. The congestion window in the beginning of the loss epoch is $\underline{w} = \left[1 - \beta\left(\frac{cd+B}{N}\right)\right]\frac{cd+B}{N}$ where $\beta\left(\frac{cd+B}{N}\right)$ is the multiplicative decrement parameter for a window size of $\frac{cd+B}{N}$. Assuming the additive parameter α is a constant in the loss epoch, the average rate in the loss epoch is approximately $\frac{\bar{w}+\underline{w}}{2d} = \frac{\left(2-\beta\left(\frac{cd+B}{N}\right)\right)}{2}\frac{c+\frac{B}{d}}{N}$. Assuming $B \to 0$, the average rate in the loss epoch is $\frac{2-\beta\left(\frac{cd+B}{N}\right)}{2}\frac{c}{N}$. Comparing to the full utilization of $\frac{c}{N}$, the loss of aggregate throughput is $\frac{2-\beta\left(\frac{cd+B}{N}\right)}{2}$.

[9] Let $p = 1$ in (7) of [39], we have $E\left(X^{(i)}\right) = \frac{C}{2N}$. This is the throughput after rate halving. Hence, the average throughput over the whole congestion epoch is $\frac{E\left(X^{(i)}\right)+\frac{C}{N}}{2} = \frac{3}{4}\frac{C}{N}$.

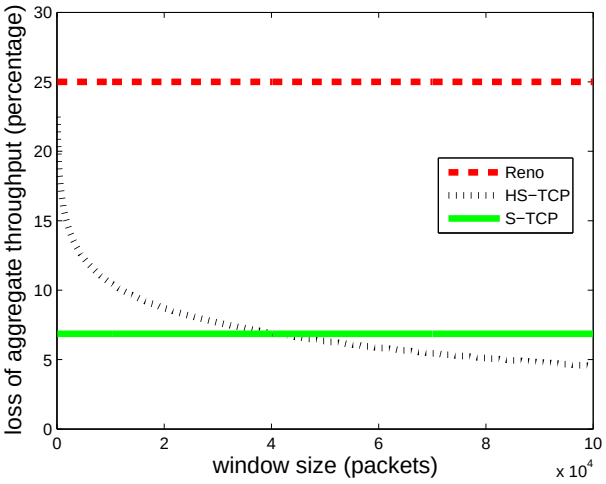


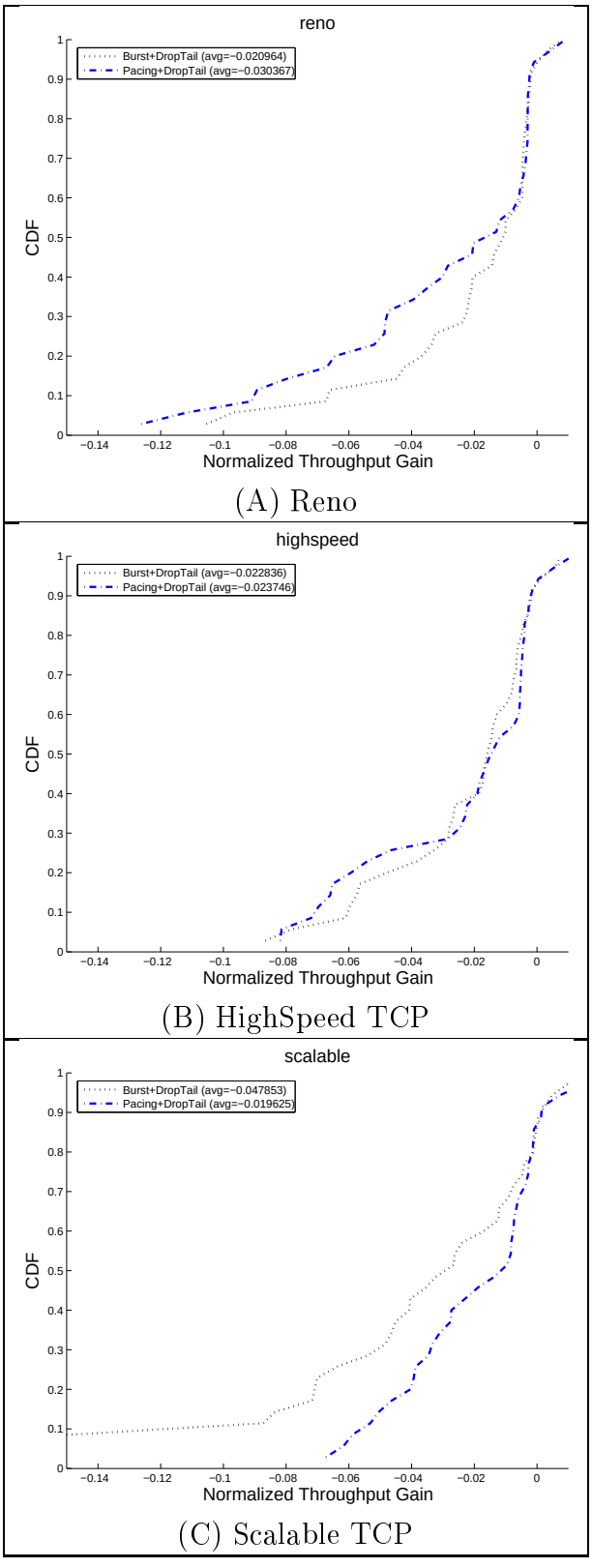
Figure 3.15: Synchronization throughput loss of different congestion control algorithm (MatLab results) (BDP = 10440 packets)

S-TCP is an MIMD algorithm. Similar to HS-TCP, the congestion window at the end of a loss epoch is $\bar{w} = \frac{cd+B}{N}$ and the congestion window at the beginning of the loss epoch is $\underline{w} = (1-\beta_S)\frac{cd+B}{N}$ where $\beta_S = \frac{1}{8}$ is the multiplicative decrement parameter in S-TCP. S-TCP multiplies its congestion window by $(1+\alpha_S)$ every RTT where $\alpha_S = 0.01$. Hence, the number of RTTs in the congestion epoch is $T_S = -\frac{\log(1-\beta_S)}{\log(1+\alpha_S)}$. The average throughput in one loss epoch is

$$\frac{\sum_{i=0}^{T_s-1} \underline{w}\,(1+\alpha_S)^i}{T_s d} = \frac{(1-\beta_S)\,c}{T_s N}\left[\frac{(1+\alpha_s)^{T_s}-1}{\alpha_s}\right]$$

and the loss of aggregate throughput due to synchronization is a constant which equals $1 - \frac{(1-\beta_S)\left[(1+\alpha_s)^{T_s}-1\right]}{\alpha_S T_s}$.

Figure 3.15 is the calculation results for these congestion control algorithms, with different buffer sizes, under a 1Gbps link with 120ms round trip propagation delay and the standard packet size (MTU=1500). We can see that all the new congestion control algorithms have much smaller throughput loss when the loss signals are synchronized. Our simulation results confirmed the expectation. Figure 3.16 present the statistic results of aggregate throughputs for paced TCP and bursty TCP. In the figure, we present the *Normalized Throughput Gain* for each experiment to illustrate the



(A) Reno

(B) HighSpeed TCP

(C) Scalable TCP

Figure 3.16: Normalized Throughput Gain of isolated bursty TCP or paced TCP in simulations

difference between achieved throughput and fairshare throughput. The *Normalized Throughput Gain* is defined as the difference between the achieved throughput and the fairshare throughput, normalized by the fairshare throughput:

$$\text{Normalized Throughput Gain} = \frac{\text{Achieved Throughput} - \text{Fairshare Throughput}}{\text{Fairshare Throughput}}$$

The achieved throughput is the measured throughput averaged over all the participant flows. The fairshare throughput is the theoretic throughput that a flow should be able to receive if all flows share the bottleneck capacity equally. If the achieved throughput is the same as the fairshare, the Normalized Throughput Gain will be zero. If the achieved throughput is lower than the fairshare, the Normalized Throughput Gain will be a negative number. From Figure 3.16 (a)-(c), we observe that there is loss of throughput for Paced TCP Reno, due to the synchronization. However, the loss of throughput is significantly reduced with HS-TCP and becomes unnoticeable with S-TCP, as we predicted. [10]

3.2.3.2 Fairness convergence

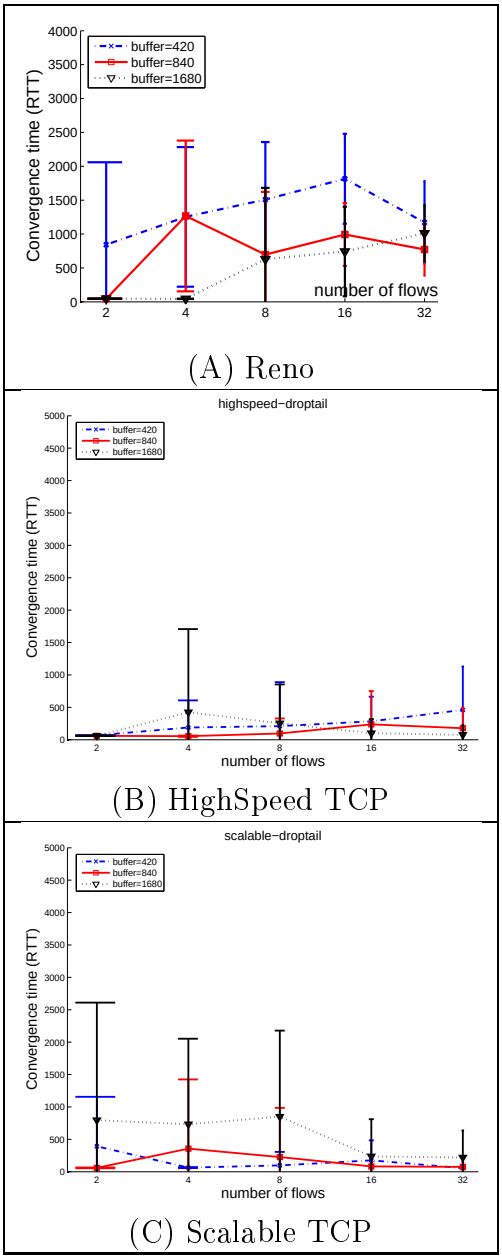
As TCP pacing increases loss synchronization rate, according to equation (3.8) and Figure 3.11, it improves the fairness convergence.

We repeat the same simulations in Figure 3.12 with a pacing extension (the detailed pacing algorithm can be found in Appendix 6.2) and present the results in Figure 3.17.

Comparing Figure 3.12 and Figure 3.17, the paced TCP flows have much faster convergence to fairness.

We summarize the fairness convergence time over all NS-2 simulations in Figure 3.18 . Overall, pacing reduce the fairness convergence time by 2.4 times.

[10] In Figure 3.16 (c), a few cases with bursty Scalable TCP have average throughputs much smaller than fair share. This can be explained because they do not converge to fairness at all.

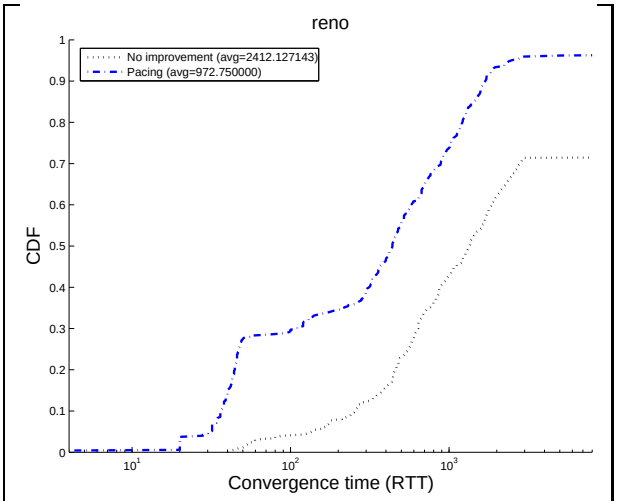


(A) Reno

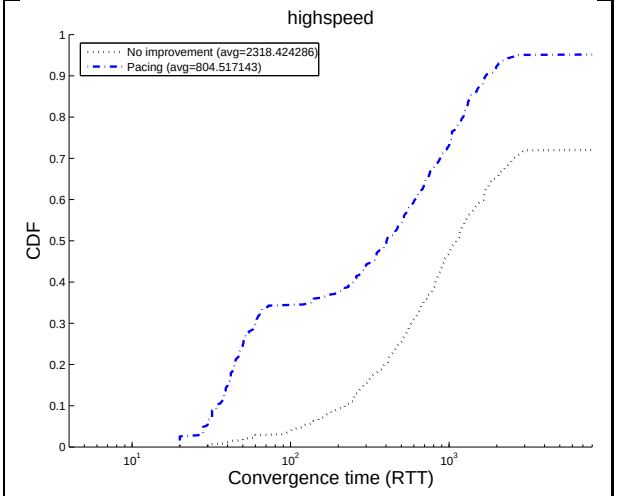
(B) HighSpeed TCP

(C) Scalable TCP

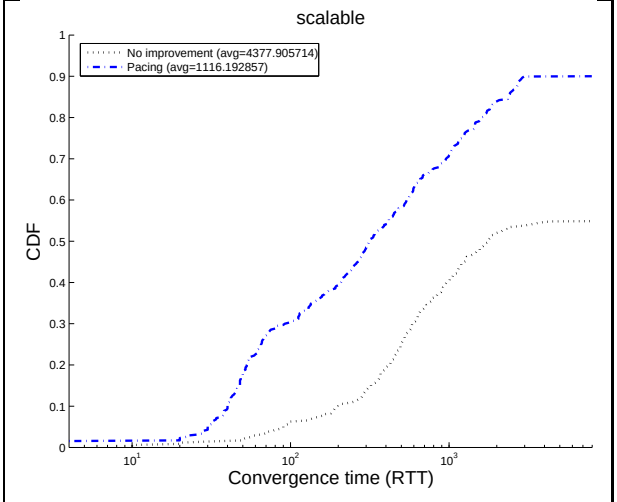
Figure 3.17: Convergence time with TCP Pacing in simulations



(A) Reno



(B) HighSpeed TCP



(C) Scalable TCP

Figure 3.18: Summary of convergence time of Reno, HS-TCP and S-TCP in simulations

3.2.4 Competition between paced TCP and bursty TCP

Aggarwal, et al also report that paced TCP tends to lose to bursty TCP in terms of average throughput, in co-existing cases [35].

The behavior of co-existing paced TCPs and bursty TCPs is very complicated since the bursty TCPs' data processes are cut into smaller bursts by the paced TCPs' data processes, and the paced TCPs' data processes are "squeezed" into small bursts by the bursty TCPs' data processes.

Qualitatively, we can expect that the loss synchronization rates of the bursty TCP will be increased and the loss synchronization rates of the paced TCP will be decreased, though in terms of absolute values, paced TCP flows still see higher loss synchronization rates than bursty TCP flows. This leads to the two effects:

1. The aggregate throughput achieved by the paced TCP flows will be smaller than the bursty TCP flows, due to the fact that paced TCPs have higher probability to detect a packet loss event and reduce their congestion windows.

2. The fairness convergence of the bursty TCP will be improved and the convergence of the paced TCP will be degraded, compared to isolated cases.

3.2.4.1 Aggregate Throughput

A paced TCP evenly distributes its data packets in one RTT and is more likely to detect a packet loss in a congestion event than the bursty TCP, which clusters its data packets within a short period. This makes the paced TCP flows lose to bursty TCP flows in co-existing scenarios. Fundamentally, this is caused by the burstiness of loss signal process, and cannot be corrected unless there are additional link-level mechanisms.

In general, if two classes of flows with different synchronization rates co-exist, the class with the higher synchronization rate will have smaller aggregate throughput. However, also from the model, we can see that paced TCP loses to bursty TCP only when the loss signal is bursty in sub-RTT level. In the next section, we propose a

link algorithm, *persistent ECN*, which can ensure that both paced TCP and bursty TCP will detect the same loss signal and get similar throughput.

Figure 3.19 presents the statistic results of aggregate throughputs, with the same network scenarios as in Figure 3.16. Instead of running the simulations with isolated paced flows or isolated bursty flows , we mixed them in one simulation, with half of the flows paced and the other half bursty. In such co-existing cases, most of the bursty TCP Reno flows get more than their fairshare bandwidth (with a positive Normalized Throughput Gain) and most of the paced TCP Reno flows get less than their fairshare bandwidth.

3.2.4.2 Fairness Convergence

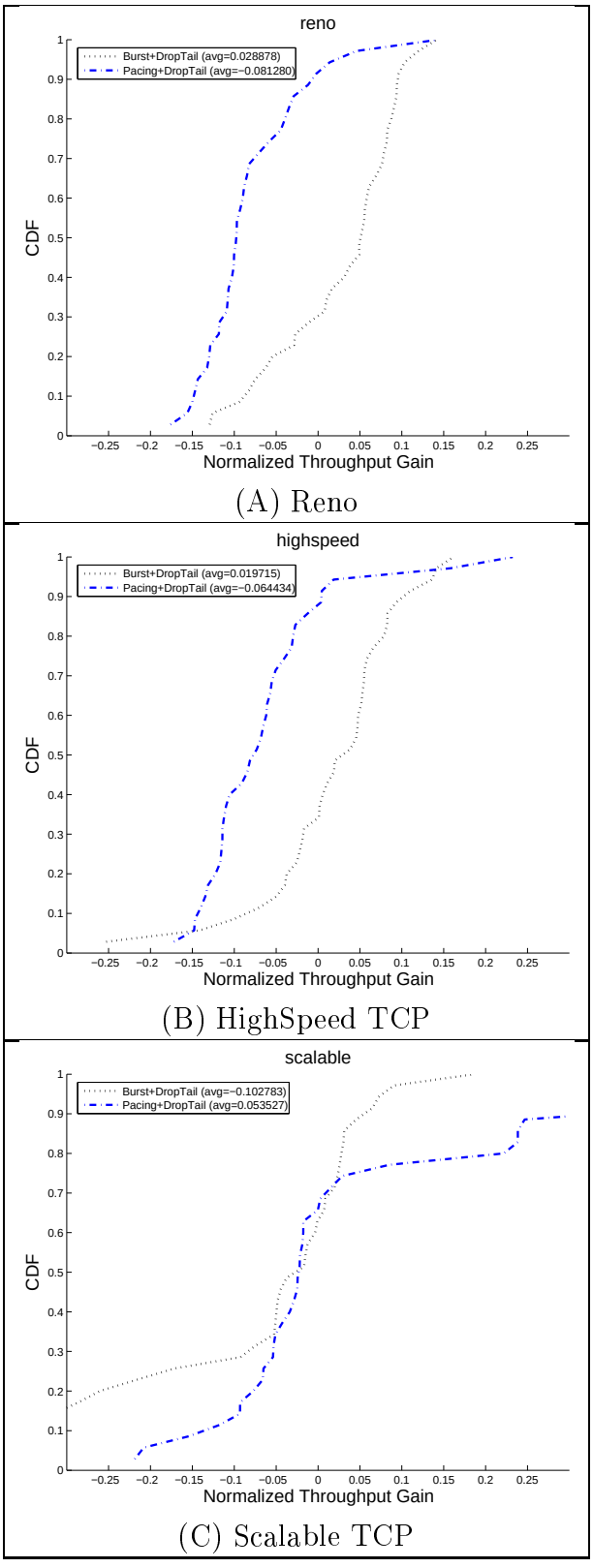
We repeated the same simulations in Figure 3.18, but with half of the flows using paced TCP and the other half using bursty TCP in each scenario. The results are presented in Figure 3.20 . The results are consistent with our expectations on fairness convergence time. Paced TCP in mixed environments has a larger convergence time than in isolated environments; bursty TCP in mixed environments has a smaller convergence time than in the isolated environments.

3.3 Algorithms

Our model shows that we can control the loss synchronization rate by controlling the pattern of data process and the pattern of loss process. This understand suggests that the use of TCP pacing or RED can increase the loss synchronization rates and improve the fairness of loss-based congestion control algorithms.

However, both pacing and RED have drawbacks. For TCP pacing, as we discussed in Section 3.2.4, flows with TCP pacing loses to flows without TCP pacing when they compete for bottleneck bandwidth. For RED, it is difficult to tune the parameters to achieve both stability and efficiency.

Figure 3.5 also shows that the highest achievable synchronization rate is still far less than 1 (0.65 in this particular case) with TCP pacing or RED. This is because

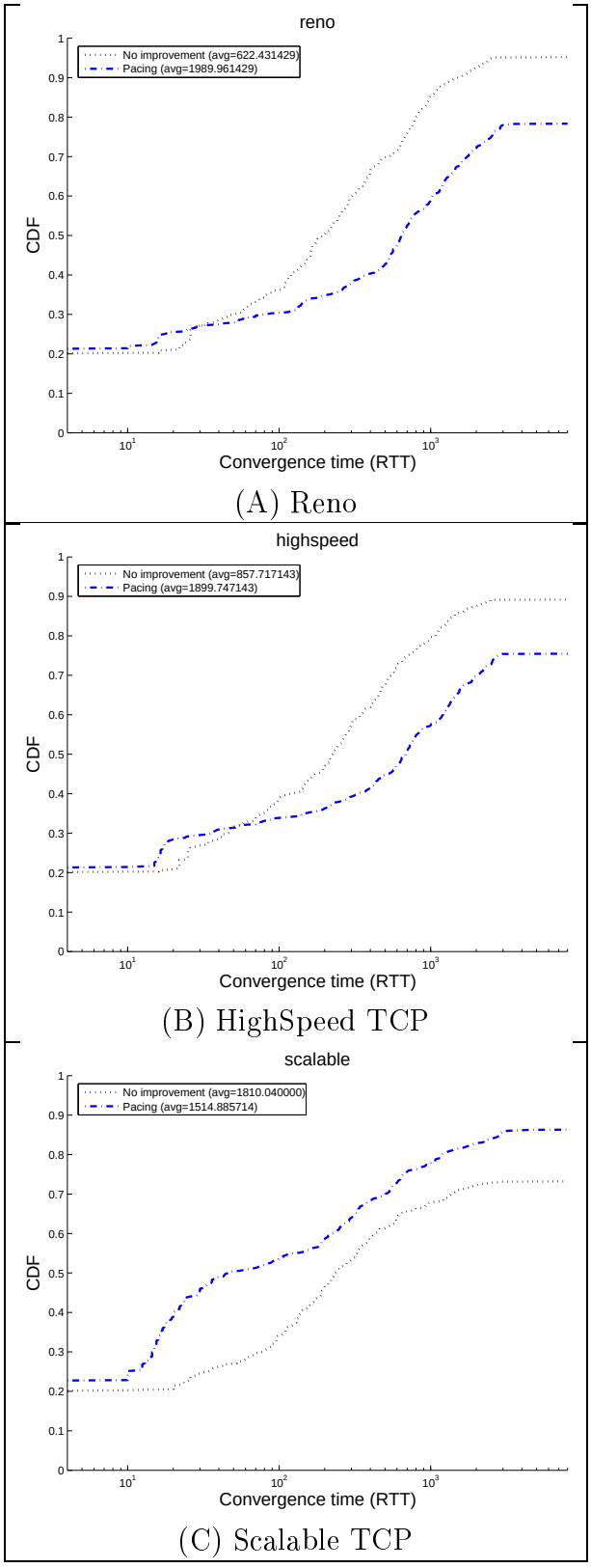


(A) Reno

(B) HighSpeed TCP

(C) Scalable TCP

Figure 3.19: Normalized Throughput Gain with co-existing pacing TCPs and bursty TCP in simulations



(A) Reno

(B) HighSpeed TCP

(C) Scalable TCP

Figure 3.20: Convergence time with co-existing pacing TCPs and bursty TCPs in simulations

the number of congestion signals (a.k.a. number of lost packets) in each loss event is too small to let each flow observe at least one signal. To further increase the loss synchronization rate, we need to maintain the congestion signal persistently for a whole RTT so that every flow is able to detect the congestion signal.

3.3.1 Persistent ECN algorithm

The above observations lead to a new algorithm: *Persistent ECN*. The goal of this algorithm is to provide a persistent congestion signal to the sources in each congestion event and achieve a very high loss synchronization rate (close to 1). There are two challenges to achieve this goal.

The first challenge is how to provide persistent congestion signal. Dropping packets persistently for the whole RTT of loss event can achieve such a goal. However, if packets in one RTT are all dropped, TCP senders will be forced to timeout and the link utilization will be significantly low. We used ECN to provide a persistent congestion signal. As TCP senders treat all ECN signals received in one RTT as the same congestion signal, the senders only reduce their congestion windows once for each loss event.

The second challenge is how to detect the end of a congestion event. We used the reduction of queue length in the router as an indication of the end of a congestion event. If the queue length is reduced significantly, we assume that the TCP senders have responded to the congestion event and the congestion event ends.

The detailed algorithm is described in Algorithm 3. The algorithm keeps two thresholds for the queue length, upper-watermark and lower-watermark. Lower-watermark is half of the value of upper-watermark. The algorithm monitors the queue length and enters *marking* state when the queue length is above the upper-watermark. In *marking* state, the algorithm marks the congestion bit in every packet. The algorithm exits *marking* state when the queue length decreases below the lower-watermark. Since the lower-watermark is half of the upper-watermark, the algorithm exits *marking* state only when the queue length has been decreased, as a result of the

Algorithm 3 Persistent ECN

A link (router) keeps two state variables:
Marking switch: $o \in \{0, 1\}$, where $o = 0$ (initially) means the link is not congested and $o = 1$ means a congestion event has happened recently;
Queue length: $q(t)$.
The link also has two constant parameters: upper water mark $\bar{Q}$ and lower water mark $\underline{Q}$. By default, $\underline{Q} = \frac{1}{2}\bar{Q}$ and $\bar{Q} = B$, where B is the buffer size.
For each packet p that arrives at the link:

1. if $q(t) \geq \bar{Q}$: $o \leftarrow 1$
2. if $o = 1$: mark the packet with congestion signal

For each packet p that leaves the link:

1. if $q(t) \leq \underline{Q}$: $o \leftarrow 0$

senders' response to the congestion signal. This algorithm can be viewed as an extension to the DropTail algorithm, as DropTail is a special case with upper-watermark and lower-watermark both equal to the buffer size. We implement this algorithm in NS-2 by modifying the DropTail queue. In the implementation, the upper-watermark is equal to the full buffer size. As this algorithm introduces a more deterministic behavior into the network, we use it with randomized pacing algorithm to eliminate phase effects.

3.3.2 Loss synchronization rate with different algorithms

We ran simulations with the same scenarios in Table 3.1 under a randomized pacing algorithm, the adaptive RED algorithm, and Persistent ECN algorithm. Table 3.2 shows the measured loss synchronization rates with these improvements in simulations. For readers' convenience, we also present the same results from Table 3.1 (without improvement) for comparison. With pacing and RED, TCP-Reno can achieve a loss synchronization rate of 0.5 to 0.6, more than two times the loss synchronization rate without these improvements. Pacing also helps HS-TCP and S-TCP to achieve a loss synchronization rate of 0.5. The loss synchronization rates of S-TCP and HS-

	Reno	HS-TCP	S-TCP
No improvement	0.2042152591	0.2496234482	0.2331014195
Pacing	0.5011622009	0.5011320266	0.5665284797
RED	0.6022723651	0.3971728460	0.3032633715
Persistent ECN	0.9663141265	0.8071350395	0.8064983921

Table 3.2: Average loss synchronization rates with different improvements

TCP with RED are also increased, but by a smaller margin. The detailed reasons for this difference are under investigation. Currently, we suspect this is due to the different dynamics when S-TCP and HS-TCP interact with RED. With Persistent ECN, the loss synchronization rate is as high as 0.8 to 0.9. As predicted in Section 3.2.1.2, such high synchronization rates can lead to a very fast fairness convergence.

3.4 Performance in Simulation

Since Section 3.2.1.2 predicts that a high loss synchronization rate can improve TCP fairness convergence, and Section 3.1 shows that loss synchronization rates can be increased by pacing, RED and Persistent ECN algorithms, we now apply these algorithms to parallel flow applications and evaluate fairness convergence time with these improvement solutions.

3.4.1 Fairness convergence and finishing time of parallel flows

A total of 350 simulation scenarios were run. Each scenario ran for at least 100 loss epochs and was repeated for at least 10 times with different random seeds. In the individual case analysis, we presented both average values and standard deviations. In the general evaluations of all scenarios, we presented CDF as summaries.

3.4.1.1 Case studies on short-term fairness

We repeated the same simulations in Figure 3.12 (bursty TCP) and Figure 3.17 (paced TCP) with RED and Persistent ECN. The results are shown in Figure 3.21 and Figure 3.22, respectively. Comparing Figure 3.12 to Figure 3.17, 3.21 and

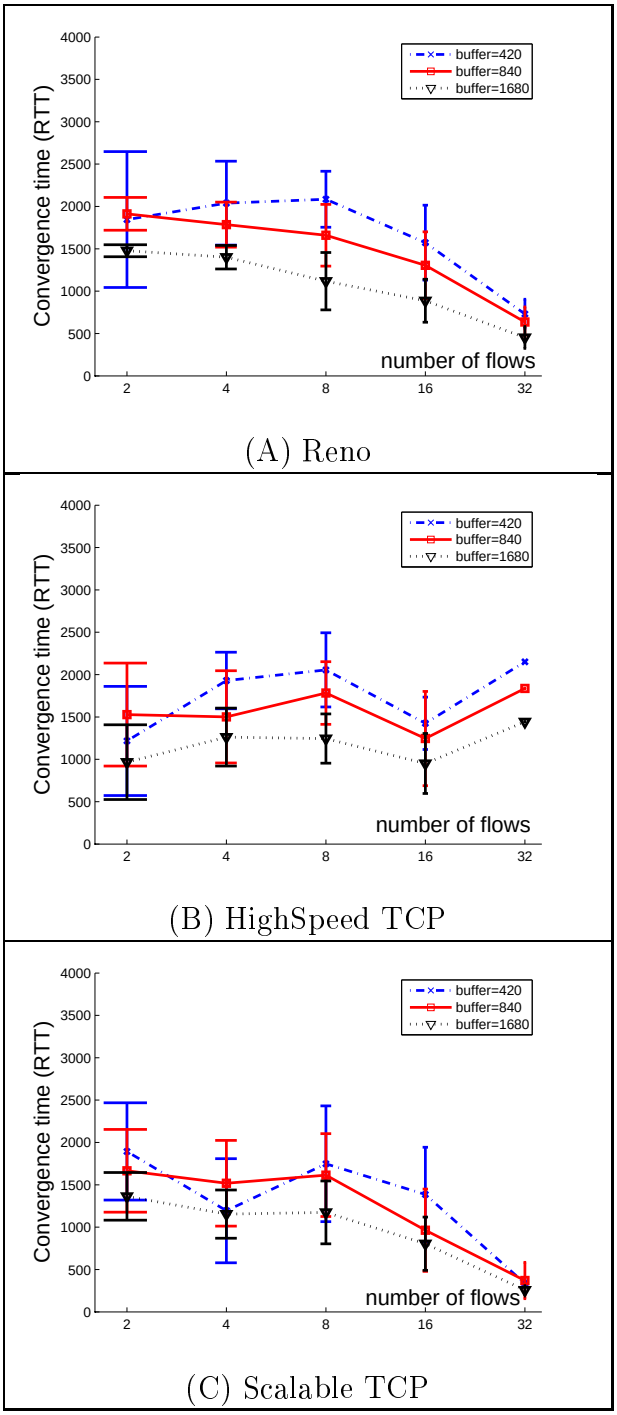


Figure 3.21: Convergence time of Reno, HS-TCP and S-TCP with RED in simulations

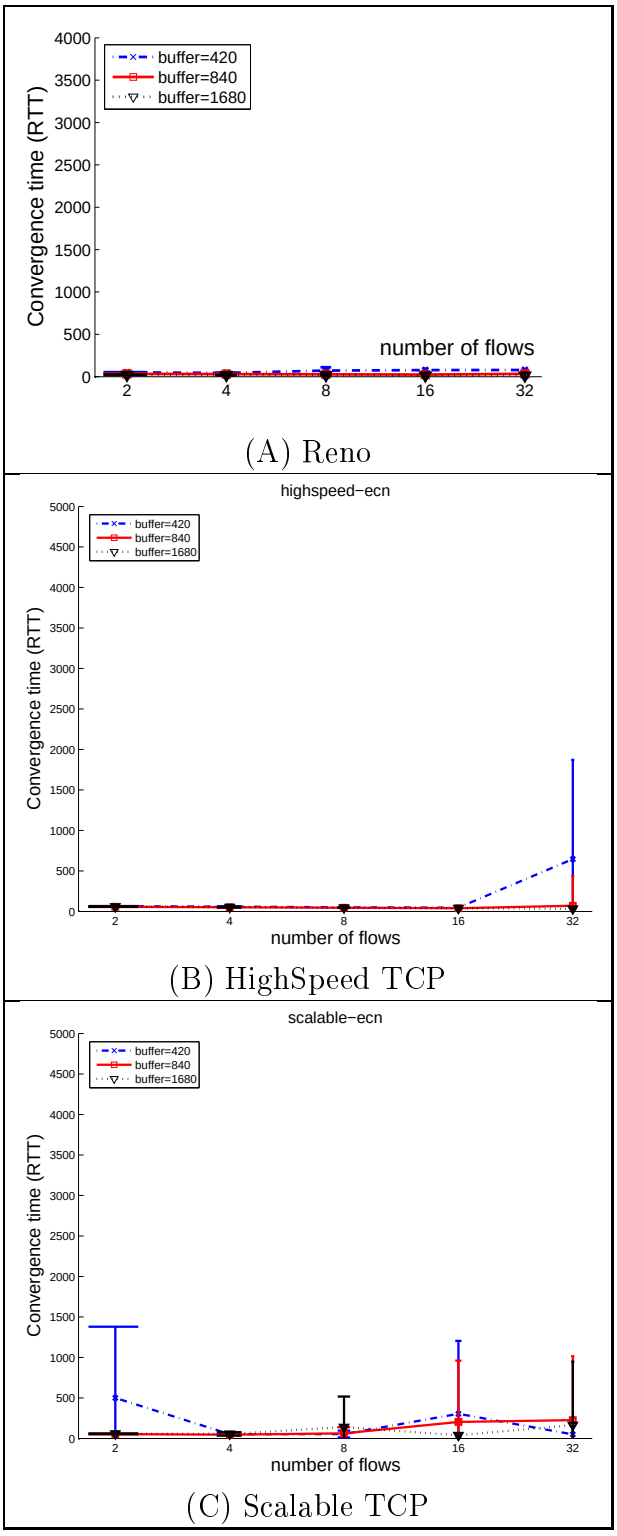


Figure 3.22: Convergence time of Reno, HS-TCP and S-TCP with Persistent ECN in simulations

3.22, one can see that both pacing and RED improve the TCP fairness convergence. Additionally, pacing has better convergence with HS-TCP and S-TCP, while RED has lower synchronization rates with these two TCP algorithms. With Persistent ECN, the convergence time is further significantly reduced. These results agree with the observations of loss synchronization rates in Table 3.2. With the same TCP algorithm, the higher the loss synchronization rate, the faster the convergence.

3.4.1.2 Summaries of short-term fairness

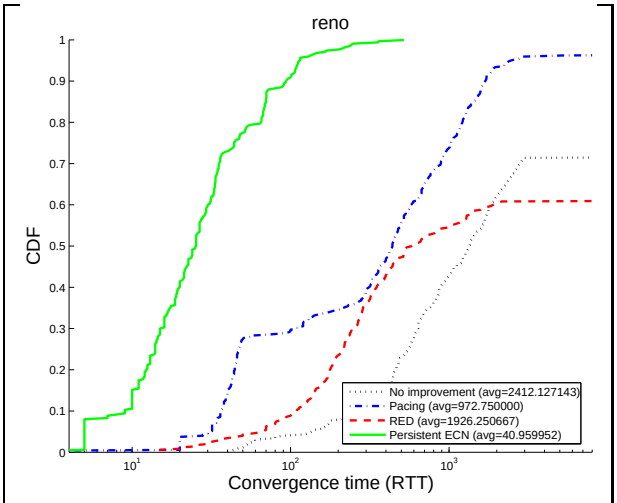
To have a global image of the parallel flow performance with different improvement algorithms, we summarized all the 350 scenarios with different improvements into CDF graphs. Figure 3.23 presents the CDF summary of the convergence time of the original TCP performances and three improvements. Note that the X-axle is in log scale. A constant gap represents a constant ratio of difference.

On average, pacing reduces the Reno convergence time by 2.4 times, RED reduces the Reno convergence time by 1.25 times and Persistent ECN reduces the Reno convergence time by 30 times. It is interesting to note that about 40% of the RED scenarios cannot converge in 1000 RTTs. In most of these cases, we found that RED could not fully utilize the bottleneck capacity. We suspect that the under-utilization is due to window oscillations with RED. We still need to investigate the details in these cases.

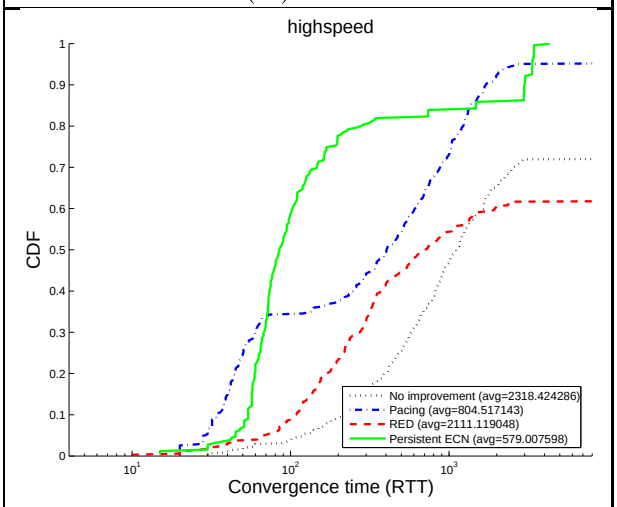
Similar observations can be found in (b) and (c) with HS-TCP and S-TCP.

3.4.1.3 Results on data transfer latency

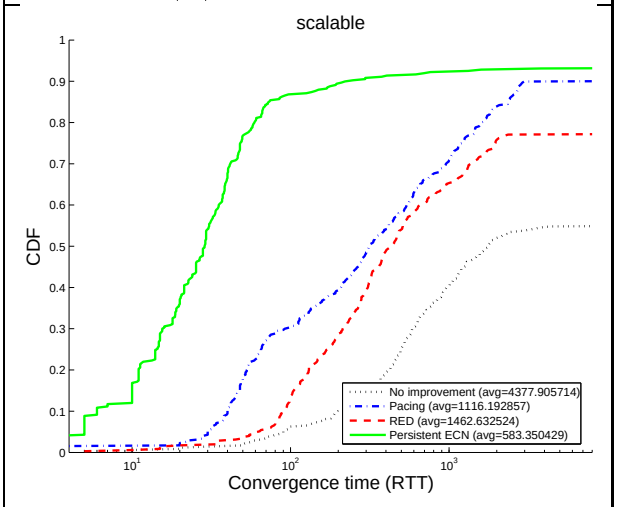
We ran simulations with parallel FTP flows that transferred a fixed amount of data. The scenario is very similar to the applications of Grid-FTP[54], GFS[55] and etc. The simulated application sends a total of 64MB of data in the same topology shown in Figure 4.13, with a different numbers of flows, different RTTs and different buffer sizes. We measured the completion time of the whole data transfer. In the 100Mbps network, the theoretic lower-bound of completion time of a 64MB transfer is 5.39 seconds. The bound is tight if the network is fully utilized in all time. We compared



(A) Reno

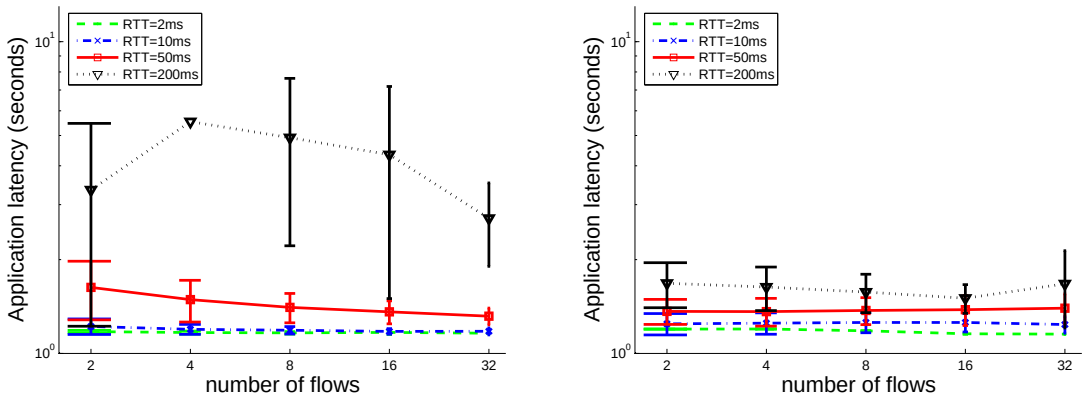


(B) HighSpeed TCP



(C) Scalable TCP

Figure 3.23: Summary of convergence time of Reno, HS-TCP and S-TCP in simulations



(a) No improvement (b) Persistent ECN

Figure 3.24: Data transfer latency (normalized by theoretic lower-bound) with parallel flows sending a total of 64MB data Both X and Y axles are in log scale.
Note that the error bar with 4 flows in (a) is too large and cannot be displayed in the figure

the completion time of TCP and the completion time of Persistent ECN, as in Figure 3.24. Due to space limit, only results on Reno TCP are presented. The results are normalized by theoretic lower-bound.

In the region of small latencies (2ms and 10ms), even though TCP's fairness convergence is long in the unit of RTT, the real time spent in the convergence is not long since the RTT is small. So, TCP still works well. When the latency is large, TCP's performance becomes unpredictable. With 200ms, TCP spend up to 10 times of the theoretic completion time.

On the other hand, Persistent ECN scales well with latency. The worst-observed latency is within 2 times of the theoretic one. This worst case happens with large delay (200ms) and a small number of flows (2 flows). In this case, slow-start takes a long time even though Persistent ECN solves the fairness problem.

These results show that Persistent ECN is a promising mechanism to improve fairness convergence of TCP flows and shorten application latency.

3.4.2 Aggregate throughput with persistent ECN

As discussed in Section 3.2.3, high synchronization rates might lead to low throughputs with TCP Reno when the buffer size in the bottleneck router is not as large as the bandwidth delay product (BDP), even though the throughput loss is bounded. With persistent ECN, we observe the same effect.

Figure 3.25 present the statistic results of aggregate throughputs for paced TCP and bursty TCP, with persistent ECN. Indeed, comparing to Figure 3.16, the aggregate throughput of Reno flows is even lower with persistent ECN. The CDF exhibits three steps. These steps correspond to the three different buffer sizes we used in the simulations (1/4 BDP, 1/2 BDP and BDP worth of buffer size). In most of the cases, the loss of average throughput is within 15%. [11]

With the new congestion control algorithms, the synchronization effect on aggregate throughput will be much less significant. On the other hand, as we have shown in the previous sections, the synchronization rate's effect on fairness is much more significant. Hence, we argue that the balance of the trade-off should move toward increasing synchronization rate.

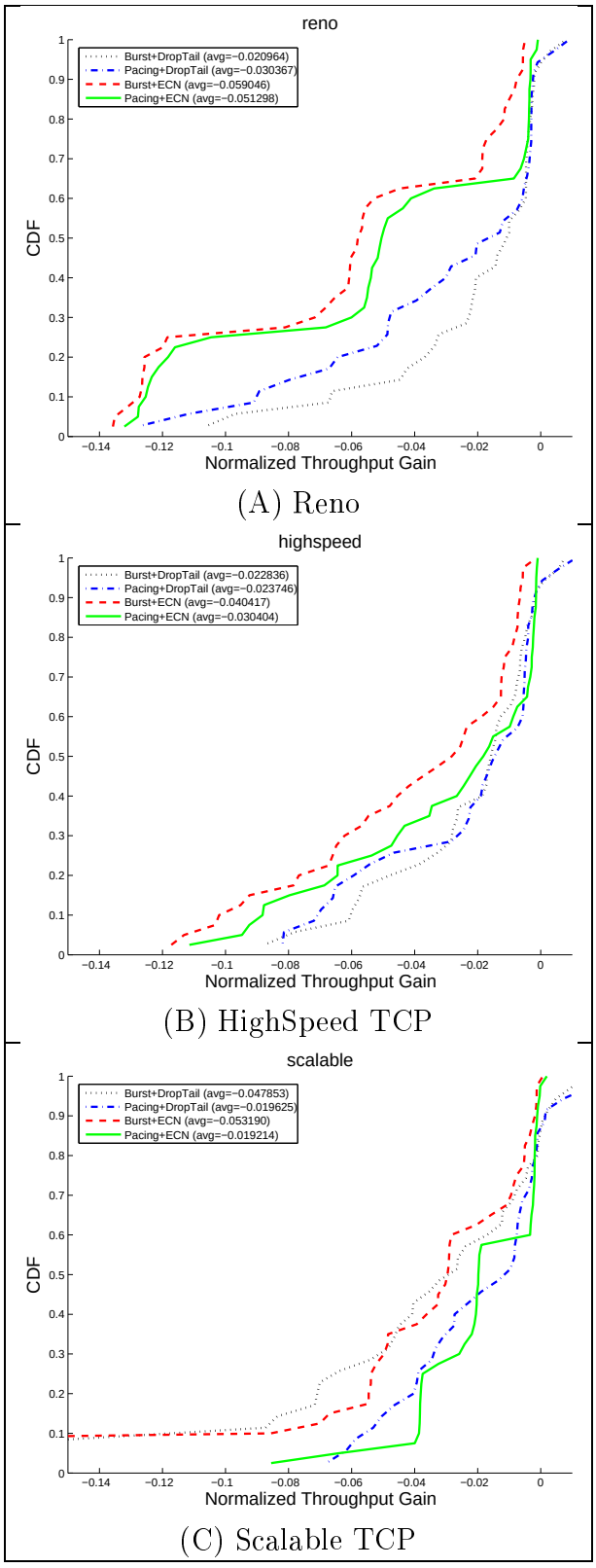
3.4.3 Aggregate throughput with co-existing bursty TCP and paced TCP under persistent ECN

Section 3.2.4 shows that the paced TCP flows lose to bursty TCP flows with a Drop-Tail router, in terms of aggregate throughput.

However, with persistent ECN, such unfriendliness disappears. This is because that the persistent ECN algorithm eliminates the sub-RTT burstiness in loss signal. When the loss signal is persistent throughout the congestion event, both paced TCP and bursty TCP will see the same loss signal and get similar throughput.

Figure 3.26 presents the statistic results of aggregate throughputs under persistent ECN, with the same simulation scenarios as in Figure 3.19.

[11] For the few bursty Scalable Cases where the average throughput is far smaller than fair share, that is because they do not converge to fairness at all.

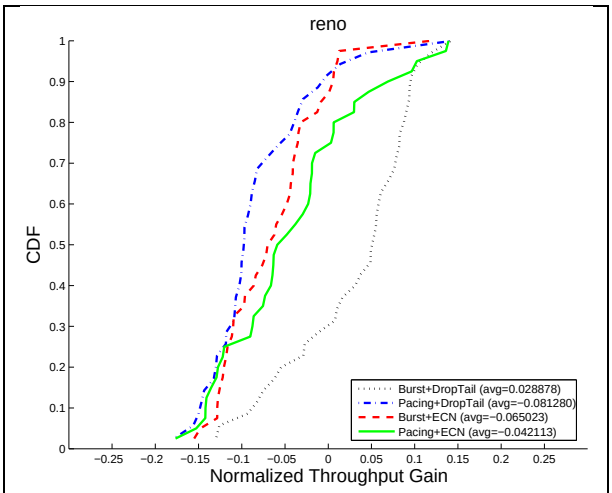


(A) Reno

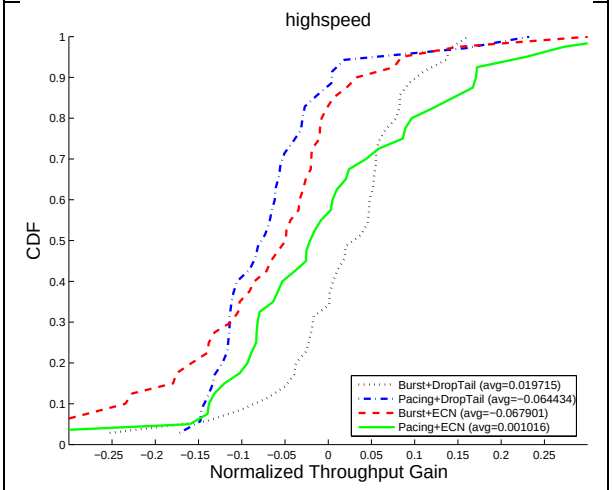
(B) HighSpeed TCP

(C) Scalable TCP

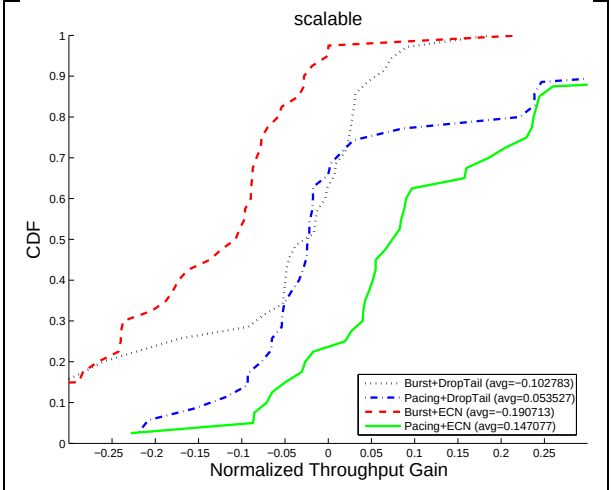
Figure 3.25: Normalized Throughput Gain with isolated pacing TCPs or bursty TCP in simulations



(A) Reno



(B) HighSpeed TCP



(C) Scalable TCP

Figure 3.26: Normalized Throughput Gain with co-existing pacing TCPs and bursty TCP in simulations

With persistent ECN, paced TCPs have higher aggregate throughput than bursty TCPs in all cases; due to two reasons: first, paced TCPs and bursty TCPs see the same congestion loss signals; second, bursty TCPs may generate additional bursty loss in sub-RTT timescale.

Hence, the persistent ECN algorithm can also be used as a link algorithm that encourages the deployment of TCP pacing.

Chapter 4

Research Tools

Our research focuses on the microscopic behavior of TCP. This focus required experimental tools more realistic and more accurate than those that existed. Hence, we developed two new tools for our research: one is a simulation module able to import the Linux source code and run real TCP congestion control implementations on the network simulator NS-2 [43]. The other is a loss measurement system that runs on PlanetLab [45] and measures loss pattern in the Internet. In addition, we also use an in-house testbed with a Dummynet emulation router [44] and Linux hosts to validate our simulation and measurement results.

4.1 A testbed with emulation router and Linux hosts

We use an in-house testbed with an emulation router and Linux hosts to validate our results in simulations and measurements.

4.1.1 Introduction to Dummynet

Dummynet is an emulation tool built in the FreeBSD system [44]. It can emulate a router, with pipes of different buffer size, different propagation delay, and different bandwidth. The testbed in our study is based on Dummynet of FreeBSD 5.2. We modified the maximum buffer size to 3000 and system clock resolution to 1 ms in the FreeBSD kernel to enhance its performance as a router emulator.

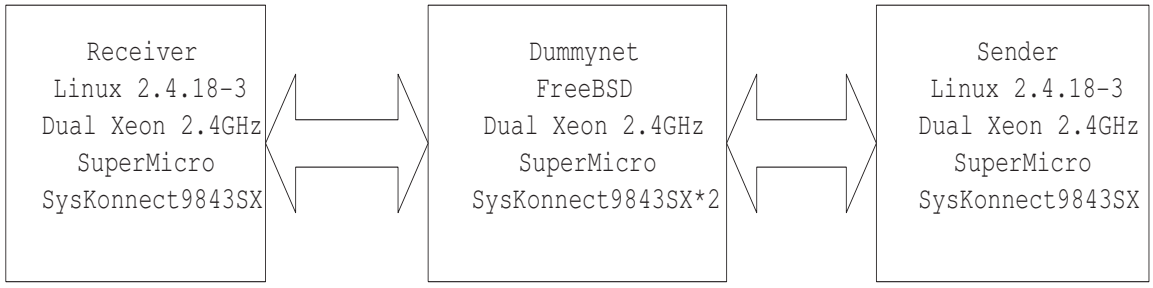


Figure 4.1: Dummynet testbed

4.1.2 Topology

The Dummynet testbed includes three machines: one sender, one receiver and one Dummynet router that emulates a WAN, as shown in Figure 4.1. The machines are equipped with Intel E1000 Gigabit Ethernet cards. The bottleneck buffer size of the Dummynet router is set to be 2000 packets. The default capacity is set to be 100Mbps. The Dummynet router processes packets with a time resolution of 1ms. Hence, the per-packet processing delay has a random fluctuation of 1ms. Four pipes with different delays (2ms, 10ms, 50ms, and 200ms) and zero random loss rate are set up on the Dummynet router for different destination ports. We use TCP flows from the single sender to different ports in the single receiver to emulate multiple flows traveling different paths with different delay and sharing the same bottleneck. The Linux sender and receiver are configured with send buffer size and receive buffer size equal to two times of bandwidth-delay-product to ensure that the send buffer and receive buffer are not the bottleneck. Iperf is used to generate TCP traffic.

4.1.3 Measurement

To collect the timing of each packet loss, we instrument the Dummynet router to record the time when each packet is dropped. The timestamp resolution of the Dummynet records is 1ms. At the Linux machines, we periodically record the congestion window and slow-start threshold of each connection from */proc/net/tcp* file. In the */proc/net/tcp* file, each line describes the information of one TCP connection. The congestion window size is the second last (16-th) column in each line. The slow start

threshold is the last (17-th) column in each line. A slow start threshold equal to -1 means the slow start threshold has not been changed and has an infinitely large value. The record frequency is 0.5 second.

4.2 *NS-2 TCP-Linux*: an extensible TCP simulation module in *NS-2*

To provide realistic TCP performance results, we developed a new module for TCP simulation in NS-2, called *NS-2 TCP-Linux*. *NS-2 TCP-Linux* is a new NS-2 TCP implementation which embeds the source codes of TCP congestion control modules in the Linux kernel. In comparison to the existing NS-2 TCP implementations, *NS-2 TCP-Linux* predicts the TCP performance more accurately with similar, or even faster, simulation speed and better extensibility to emerging new TCP congestion control algorithms in the future. *NS-2 TCP-Linux* was the major tool used in our research. In addition to helping us in performance analysis, *NS-2 TCP-Linux* has also helped the Linux kernel community to debug and test new congestion control algorithms.

4.2.1 An introduction to TCP implementation in NS-2

NS-2 is an open-source packet level network simulator widely used in network performance analysis [43]. It provides fairly accurate results for simple devices (e.g. DropTail link). The NS-2 simulations are fully controllable and repeatable. The TCP modules in NS-2 were originated from source codes of BSD kernel. Over the years, NS-2 TCP modules have contributed tremendously to the understanding and analysis of TCP behaviors, and led to the development of several new congestion control algorithms. The TCP implementation in NS-2 currently include TCP-Reno [5], NewReno [6], Sack [56], Fack [7], HighSpeed TCP [8], etc. However, as the major operating systems evolved gradually over the years, the TCP modules in NS2 have deviated significantly from mainstream operating systems such as FreeBSD and Linux.

The use of NS-2 has become less popular in the congestion control community, due to difficulties in the following three aspects:

- Extensibility: As NS-2's code structure deviates from mainstream operating systems, it becomes harder and harder to implement the NS-2 correspondent of a Linux algorithm. For new algorithm designers, it is a huge burden to implement the same congestion control algorithm in both NS-2 and a real system such as Linux.

- Validity of NS-2 results: As more improvements are implemented in Linux, the performance predicted by NS-2 simulation deviates more from the Linux performance and FreeBSD performance. As reported in recent literature as [57], the difference in performance can be significant in some scenarios.

- Simulation speeds: NS-2 users often suffer from long simulation time when they simulate scenarios with high-speed long-distance networks. In some examples, the simulator might take up to 20 hours to finish a 200-second simulation [58]. This is particularly troublesome to network simulations since many runs of the a scenario with different random seeds are usually required to eliminate artifacts of deterministic behaviors, such as phase effects.

Indeed, there is a trend that designers of new congestion control algorithms are less inclined to use NS-2 to evaluate their algorithms. For example, the TCP congestion control community has sparked many new congestion control algorithms for high-speed long-distance networks, but many of them [52, 10, 11] are first implemented in Linux and evaluated in emulation testbeds such as the Dummynet testbed described in Section 4.1. This new approach allows new algorithms to be evaluated in a real operating system and easily deployed with Linux releases. However, due to limitations of the emulation router, these evaluations are usually limited to very simple topologies and very small scale in terms of numbers of flows, delays, and bottleneck capacities. This approach is also riskier as the new algorithms can be deployed without thorough tests in complicated scenarios. For example, recent research [59, 60] points out some

interesting observations on TCP behaviors that only exist in scenarios with multiple bottleneck links, which are hard to reproduce by Dummynet testbeds with dumbbell topologies. Hence, we believe that NS-2 is a critical component in the spectrum of tools to evaluate protocol performance with its flexibility in topology and scale. *NS-2 TCP-Linux* is designed to serve this purpose by improving the current NS-2 implementations.

Corresponding to the difficulties, *NS-2 TCP-Linux* has three design goals:

- Enhance extensibility by allowing users to import congestion control algorithms directly from Linux source codes;
- Provide simulation results that are close to the performance of Linux;
- Maintain the simulation speed at least as fast as the current TCP modules.

To improve the accuracy of the simulation result, we redesigned the loss recovery module (ScoreBoard). We also improved the scheduler to speed up the simulation. [1]

Our efforts and results show that these three goals can be achieved at the same time, when the NS-2 TCP module is carefully redesigned.

4.2.2 An introduction to Linux TCP

Both NS-2 TCP and Linux TCP [61, 62, 63] implementations follow the relevant RFCs. However, there are a few major differences between the existing NS-2 implementation and Linux implementation. Some of them are listed below:

1. SACK [56] processing: current NS-2 TCP (*Sack1* and *Fack*) times out when a retransmitted packet is lost again. Linux SACK processing may still recover if a retransmitted packet is lost;
2. Rate halving [64]: Linux has a complicated rate halving process which gradually reduces the congestion window to half of the slow start threshold, and then

[1] Note that these two improvements can be used in a more general context. Scoreboard1 can be used in other TCP implementations and the improved scheduler can be used in any NS2 simulation.

returns to the slow start threshold after recovery; NS-2 has a simplified rate-halving algorithm;

3. Delayed Ack: the Linux receiver disables delayed ack in the first few packets to avoid delaying slow start;[2]

4. Duplicated SACK (D-Sack [65]): current NS-2 TCP does not use D-SACK information to infer the degree of packet reordering in the path; Linux has a process to detect D-SACK and adjust duplicated ACK threshold.

All these different issues in implementation lead to differences in throughput predicted by NS-2 and throughput achieved by Linux. More importantly, the code structure of NS-2 is very different from the code structure in Linux. It is a burden to port an algorithm between Linux and NS-2. From version 2.6.13, the Linux kernel has introduced the concept of *congestion control modules* [61]. A common interface is defined for congestion control algorithms, and algorithm designers can implement their own congestion control algorithms as Linux modules easily.

With this interface, all the state variables for a TCP connection are stored in a structure called *tcp_sk*. The interface also defines a congestion window operation structure called *icsk_ca_ops*, for third party to write new congestion control algorithm as a kernel module in Linux. The *icsk_ca_ops* structure is a set of function pointers. The structure has three required function call pointers:

- *cong_avoid* function: This function is called when an ack is received. The implementation of this function is expected to change the congestion window in this function during the normal situation (without loss recovery). In TCP Reno, this means slow start and congestion avoidance.

- *ssthresh* function: This function is called when a loss event occurs. It is expected to return the slow start threshold after a loss event. The returned value shall be half of *snd_cwnd* in TCP Reno.

[2]This is not RFC-conforming but follows Nagle's advice.

- *min_ cwnd* function: This function is called when a fast retransmission occurs, after *ssthresh* function. It is expected to return the value of the congestion window after a loss event. In Reno, the returned value shall be *snd_ ssthresh*.

Linux kernel performs tasks of acknowledgment processing, SACK processing, loss detection and retransmission. It calls the congestion control module when the congestion window or the slow start threshold needs to be changed (e.g. upon the arrival of a new acknowledgment or a loss is detected). In these cases, Linux kernel calls the corresponding function pointers in *icsk_ ca_ ops* structure. The address of *tcp_ sk* is passed to the congestion control module as a parameter so that the congestion control module has its flexibility to read or change other TCP states as well.

A congestion control module are required to implement the above three functions and control the congestion window and the slow start threshold. A very simple algorithm (TCP Reno) is shown in Figure 4.2 as an example of an implementation for this interface.

More sophisticate congestion control schemes require more operations such as obtaining high resolution RTT samples [66]. These *advanced* functions are introduced in Appendix 6.1.

Until the version of 2.6.16-3 appeared, Linux incorporated nine congestion control algorithms in the official release version. At least three new implementations are anticipated in the community. We believe that NS-2 will benefit from a new TCP module which conforms to the Linux *congestion control module* interface. The benefits are two-fold: First, the research community can use NS-2 to analyze Linux algorithms without implementing the NS-2 version of a Linux algorithm. This leads to improved productivity and accuracy. Second, NS-2 is a tool the Linux community can use to debug and test their new congestion control algorithms. This leads to more reliable and better-understood implementations. Hence, we designed *TCP-Linux* in the spirit of bridging the gap between the implementation community developing the Linux system and the analysis community using the NS-2 as a tool.

```
/* Create a constant record
* for this congestion control
* algorithm for the interface */
struct tcp_congestion_ops
    simple_reno = {
.name       = "simple_reno",
.ssthresh   = nr_ssthresh,
.cong_avoid= nr_cong_avoid,
.min_cwnd   = nr_min_cwnd
};

/* This function returns the
 * slow-start threshold after
 * a loss.
 */
u32 nr_ssthresh(struct tcp_sk *tp)
{
  return max(tp->snd_cwnd/2,2);
}

/* This function returns the
 * congestion window after a
 * loss -- it is called AFTER
 * the function ssthresh (above)
 */
u32 nr_min_cwnd(struct tcp_sk *tp)
{
  return    tp->snd_ssthresh;
}
```

```
/* This function increases
 * congestion window for
 * each acknowledgment
 */
void nr_cong_avoid
 (struct tcp_sk *tp, ...)
{
  if (tp->snd_cwnd <
          tp->snd_ssthresh)
  {
//slow start
    tp->snd_cwnd++;
  } else {

//congestion avoidance
    if (tp->snd_cwnd_cnt <
                tp->snd_cwnd)
    {
    // not enough for 1 pkt,
    // we increase the fraction.
        tp->snd_cwnd_cnt++;
    } else {
    // we can increase cwnd
    // by 1 pkt now.
        tp->snd_cwnd++;
        tp->snd_cwnd_cnt = 0;
    }
  }
}
```

Figure 4.2: A very simple implementation (Reno) of the congestion control interface

Name	Meanings	Equivalence in NS-2 *TCPAgent*
snd_ssthresh	the slow start threshold	ssthresh_
snd_cwnd	integer part of the congestion window	$\lfloor$cwnd_$\rfloor$
snd_cwnd_cnt	fraction of the congestion window	$\lfloor$cwnd_ $*$ cwnd_$\rfloor$%$\lfloor$cwnd_$\rfloor$
icsk_ca_priv	a 512-bit array to hold per-flow states for a congestion control algorithm	n/a
icsk_ca_ops	a pointer to the congestion control algorithm interface	n/a

Table 4.1: Important variables in *tcp_sk*

4.2.3 Design of *NS-2 TCP-Linux*

The design of *NS-2 TCP-Linux* shares the same congestion control interface with Linux. It allows users to easily port the source code from a congestion control implementation from Linux to NS-2.

4.2.3.1 Interface

NS-2 TCP-Linux follows the same interface of congestion control module in Linux 2.6. It uses the same congestion control module structure as Linux 2.6 (struct *tcp_congestion_ops*) . [3]

However, the simulation module only supports a subset of fields in the *tcp_sk* structure and synchronizes these fields with the variables in NS-2 TCP; the most important fields and their corresponding variables in NS-2 are listed in Table 4.1. Appendix 6.1 provides details of all Linux parameters supported by *NS-2 TCP-Linux*.

By sharing the same interface as congestion control module in Linux 2.6, *NS-2 TCP-Linux* is able to use the source code of congestion control modules from Linux kernel with minor changes, ensuring the extensibility of *NS-2 TCP-Linux*.

[3]The meaning of *min_cwnd* function in *TCP-Linux* is, however, slightly different from Linux. Linux has a complicated rate-halving process and *min_cwnd* is used as the lower bound of the congestion window in a rate-halving process after a loss event. In NS-2, *TCP-Linux* has a simplified version of rate-halving, and the congestion window can be set to *min_cwnd* directly.

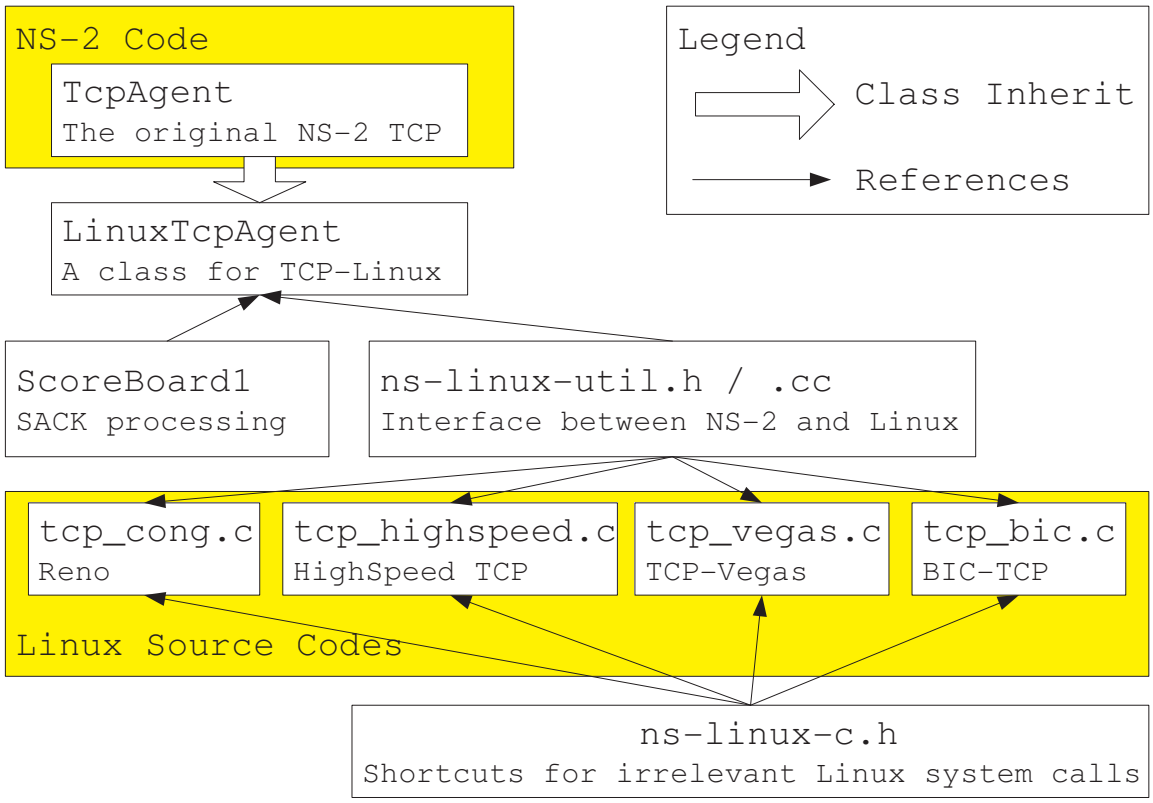


Figure 4.3: Code structure of *TCP-Linux*
The boxes in the shaded areas are components from existing source codes in NS-2 or in Linux kernel. The four white boxes outside the shaded areas four components in *TCP-Linux* implementation.

4.2.3.2 Code architecture

TCP-Linux implements a simplified version of the Linux kernel packet processing tasks and conforms to the congestion control module interface. The code structure has four major components, as presented in Figure 4.3:

- LinuxTCPAgent: The NS-2 module for *TCP-Linux* (tcp-linux.h and .cc)
- Scoreboard1: The scoreboard design (scoreboard1.h and .cc) which manages loss detection and packet retransmission
- The interface between C++ codes in NS-2 and C codes in Linux (ns-linux-util.h and .cc)
- Shortcuts for Linux system calls (ns-linux-c.h and .c)

The interface between C++ and C is a set of data structure declarations. The shortcuts for Linux system calls is a set of macros that redefine many Linux system calls not relevant to congestion control. These two components serve as a highly simplified environment for the embedded Linux source codes. A simplified TCP control block (tcp_sock structure in Linux) serves as the data structure for communication between Linux congestion control modules and LinuxTCPAgent in NS-2.

With the interface and shortcuts, users of *TCP-Linux* can easily include new congestion control algorithms from the Linux source codes [67].

LinuxTCPAgent is the main NS-2 component of *TCP-Linux*. It has two major functions:

1. Simulate Linux acknowledgment packet processing;

2. Provide user interface, trace and measurement support for NS-2.

Different from other TCP modules in NS-2, LinuxTCPAgent loosely follows the design of the Linux acknowledgment processing process (tcp_ack function in Linux), including RTT sampling routine, SACK processing routine, fast retransmission routine and transmission timeout routine. We made the following simplifications to tailor the Linux implementation to NS-2:

- Eliminate the difference between fast path and slow path: NS-2 simulation does not process actual data packets. There is no difference in fast path and slow path in NS-2. However, the Linux congestion control module interface allows a congestion control algorithm to use fast path and slow path as a hint for congestion level; we use a very simple algorithm to simulate this hint.

- The rate halving process is greatly simplified: In Linux, the congestion window is not reduced immediately when the sender observes a packet loss. Instead, the congestion window is reduced by half a packet every round trip time during the FastRetransmission state, until it hits the minimum congestion window threshold (usually, the threshold equals the slow start threshold or half of the

slow start threshold). If the sender recovers from FastRetransmission state before the congestion window hits the minimum threshold, the sender has to first complete the window reduction and then return to normal state. In *NS-2 TCP-Linux*, the implementation does rate halving more explicitly. The sender halves the congestion window immediately after a loss. However, it keeps track of the number of packets in flight, and allows at most one packet to be sent out for each ack, as long as the number of packets in flight keeps reducing. We did not notice any major difference in the results when comparing the results of *TCP-Linux* simulation and Linux experiments.

- The loss recovery does not have an "undo" function: In Linux, when a false retransmission is detected (e.g. due to D-SACK), Linux can undo the congestion window reduction due to the false retransmission. This function is not implemented in *TCP-Linux*. There might be an impact to the simulation results when the scenario includes a network device which reorders data packets. We do not have scenarios with packet reordering device. For general purpose usages, we need to investigate this impact further .

The LinuxTCPAgent updates the NS-2 traced variables at the end of each ack processing routine and support congestion control algorithm selection as a command line option, making it very easy to use *NS-2 TCP-Linux*. For any existing TCP simulation scripts, users only need to add one command to use *NS-2 TCP-Linux* [67].

4.2.3.3 Scoreboard1: improving the accuracy by better loss recovery

Scoreboard1 is a new scoreboard implementation that combines the advantage of Scoreboard-RQ in NS2 and the Linux SACK processing routine (sacktag_write_queue). Similar to Linux SACK processing, each packet in the retransmission queue is in one of the four possible states: InFlight, Lost, Retrans or SACKed. The state transition diagram is shown in Figure 4.4.

A packet that is sent for the first time, but not acknowledged and not considered lost, is in InFlight state. It enters SACKed state if it is selectively acknowledged

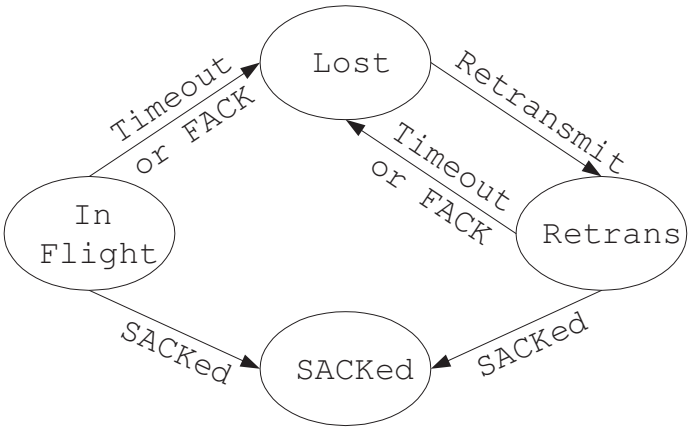


Figure 4.4: State machine of each packet

by a SACK, or enters Lost state if a retransmission timeout occurs, or the furthest SACKed packet is more than 3 packets ahead of it.

A packet in Lost state will be retransmitted and enter Retrans state.

When a packet is retransmitted, it is assigned with a sequence number *snd_nxt* (similar to Scoreboard in NS-2 Fack) which records the packet sequence number of the next data packet that is going to be sent. Additionally, it is also assigned with a *retrx_id* which records the number of packets that is retransmitted in this loss recovery phase, as shown in the first two nodes in Figure 4.5. The (*snd_nxt*, *retrx_id*) pair helps detect if a retransmitted packet is lost. The retransmitted packet can be considered lost if: [4]

1. Another packet is selectively or accumulatively acknowledged (SACKed or ACKed), and the acknowledged packet's sequence number is higher than *snd_nxt*+3; or

2. Another retransmitted packet is selectively or accumulatively acknowledged, and the acknowledged packet's retrx_id is higher than *retrx_id*+3.

With the definition of per-packet state, Scoreboard1 can keep an explicit counter for the number of packets in flight (which equals to the sum of number of packets in

[4]Strictly speaking, the Linux implementation also includes the third case: when a packet is acknowledged, and its transmission timestamp is higher than an unacknowledged packet's transmission timestamp plus RTO, the unacknowledged packet is considered to be lost. This is called head timeout. This case is not included in the current implementation of *TCP-Linux* and it might affect the Linux performance when the packets in flight is smaller than 3.

InFlight state and the number of packets in Retrans state).

To improve the speed of SACK processing, Scoreboard1 incorporates the one-pass traversing scheme from Scoreboard-RQ. Scoreboard1 organizes all the packets in a linked list, as shown in Figure 4.5. Each node of the linked list can be either

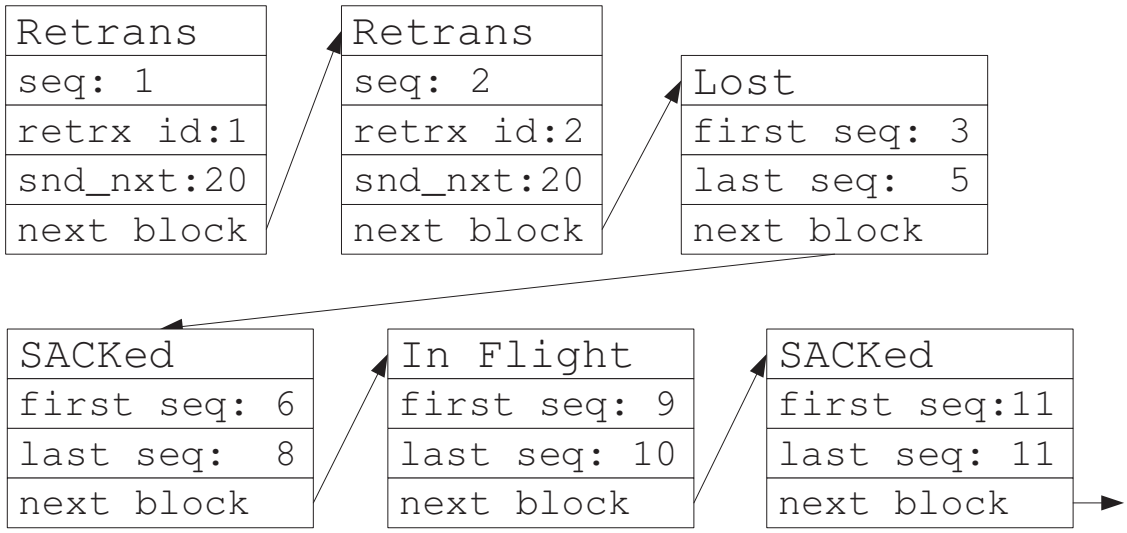


Figure 4.5: SACK queue data structure

a single packet in retransmitted state, or a block of packets in other states. This linked list allows ScoreBoard1 to traverse the retransmission queue only once every acknowledgment, regardless of the number of SACK blocks in the acknowledgment.

The retransmission queue update process is a simplified version of tcp_clean_rtx_queue and tcp_sacktag_write_queue in Linux, without D-SACK processing and timestamp processing.

4.2.3.4 SNOOPy Queue Scheduler: Speed up the simulation with a better scheduler

The current NS-2 (Version 2.29) scheduler uses a calendar queue [68] to store simulation events. A calendar queue is similar to a hash table with dynamic bucket size and uses the time of the event as the key. Intuitively, a bucket in a calendar queue corresponds to a "day" in a real calendar. Each bucket has a linked list to store multiple events, just like multiple notes can be written in each day on a real calendar.

The whole bucket array corresponds to a "year". Events in the same "day" but in different "years" share the same bucket in increasing order. When a new event is inserted, the event's destination bucket can be calculated in O(1) by the hash key, and the event is inserted into the in-order position of the destination bucket via a linear search along the linked list. To de-queue the next event, the calendar queue traverses the bucket array to find the bucket with the earliest event. The size of the array may be doubled if the number of events grows larger, or halved if the number of events grows smaller, to keep the average length of all the linked lists within a constant range. On average, the calender queue can insert an event and de-queue an event in O(1).

The efficiency of the calendar queue depends on the width of each bucket. If the width of a bucket is too large ("a long day"), many events may be put into one bucket and the calendar queue degrades into a single linked list which requires a linear search when a new event is inserted. If the width of a bucket is too small ("a short year"), most of the events in the buckets are of different years and repeated linear searches over the bucket array are necessary to de-queue the next event. Dynamic Calendar Queue [69] suggests that the bucket size should be dynamically set to the average interval in the fullest bucket. The NS-2 calendar queue takes the suggestion in setting the bucket width.[5]

However, this suggestion on bucket width works perfectly only if the events are evenly distributed in the calendar queue. If events that span over many "years" happen to be in the fullest bucket, while most of the events in the whole calendar queue are clustered within several "seconds", NS-2 will set the bucket width to be very large, on the order of "year". In this case, most of the events (clustered within seconds) will go into a few buckets. The calendar queue hence degrades into a few linked lists and long linear searches occur in event insertions.

Unfortunately, such uneven event distribution is very common in NS-2 simulation. Users usually set an "end time" before the simulation start. This end time corresponds

[5]Different from [69], NS-2 does not adapt the bucket width until the number of events is too large or too small. This difference further degrades performance when the bucket width is set to an inefficient value.

to an event far in the future in the calendar queue. If this "end time" event happens to be in the fullest bucket when NS-2 sets its bucket width, the simulation speed slows down significantly.

To correct this problem, we added an average interval estimation into the calendar queue scheduler. We used the average interval of each pair of *de-queued* events, averaged over a whole queue size of de-queued events, as an estimation of the bucket width. If the event departure interval is similar over time, this width results in the O(1) in both de-queue and en-queue operation.

To address the possible change of event departure patterns, we also implemented SNOOPy Calendar Queue [70], which dynamically adjusts the bucket width by balancing the linear search cost in the event insertion operations and the event dequeueing operation.

With these two improvements, the scheduler performed consistently in terms of simulation speed.

4.2.4 Validation of *NS-2 TCP-Linux*

We examined simulation results with *TCP-Linux* according to our three goals: extensibility, accuracy and performance. To validate the accuracy of the simulation results of *TCP-Linux*, we compare them with Linux results from Dummynet testbed. To evaluate the performance, we compare the simulation time and memory usage of *TCP-Linux* and *TCP-Sack1*, the best TCP implementation in NS-2.[6] We also present a real example of how *TCP-Linux* can help the Linux community debug and test congestion control implementations.

The setup of our NS-2 scenario is shown in Figure 4.6. There is one FTP flow running from the sender to the receiver for 900 seconds. We recorded the congestion window every 0.5 second.

The setup of our Dummynet experiment is shown in Figure 4.7. In the experi-

[6]We also tried other existing implementation in NS-2. *Reno* and *NewReno* have much more timeout than Sack1, leading to even poorer accuracy. *Fack* and *Sack-RH* run much more slowly due to the inefficient implementation in Scoreboard.

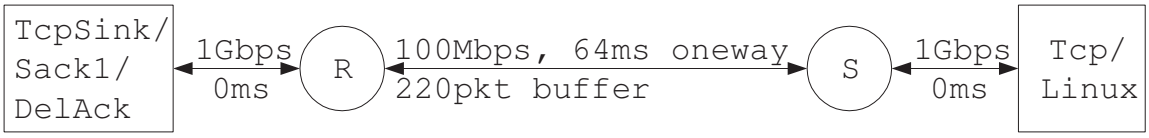


Figure 4.6: Setup of NS-2 Simulation

ments, the application is Iperf with a large enough buffer. We read the /proc/net/tcp file every 0.5 second to get the congestion window value of the Iperf flow and compare the congestion window trajectories with the simulation results.

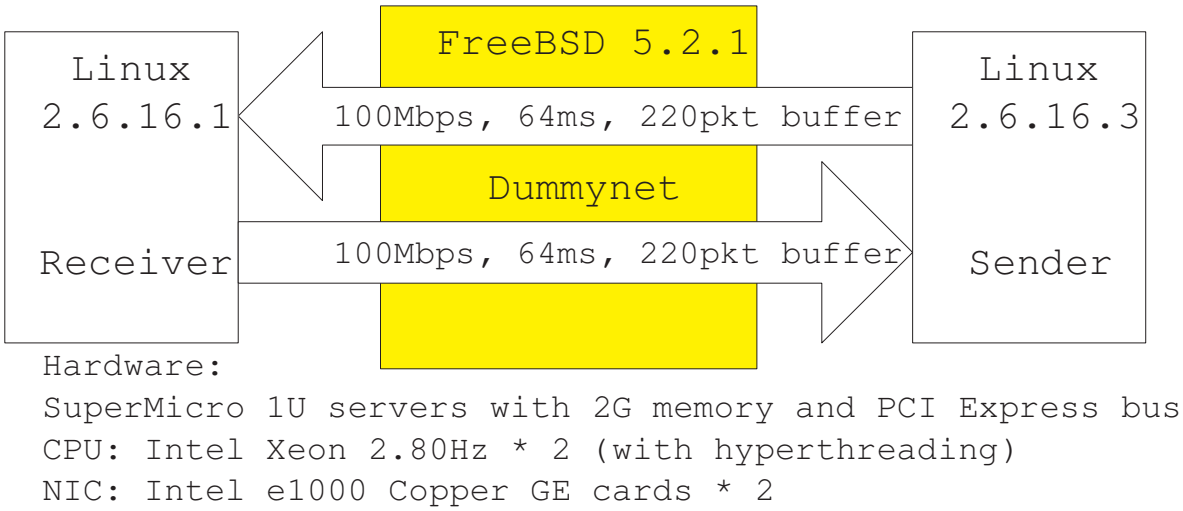


Figure 4.7: Setup of Dummynet Experiments

4.2.4.1 Extensibility

We incorporated all the nine different congestion control algorithms in Linux 2.6.16-3 into *NS-2 TCP-Linux*. Six of them are not in the current NS-2 release (2.29). Table 4.2shows the results with these six different congestion control algorithms. To make the figures readable, we re-scaled the time axles in the figures to include only six congestion epochs in each figure.

From Table 4.2, we can see that the congestion window trajectories predicted by *NS-2 TCP-Linux* are very similar to the results from Dummynet testbed. The two cases which have the most significant differences are TCP-Hybla, and TCP-Cubic. TCP-Hybla measures the round trip delay to set its additive increment (AI)

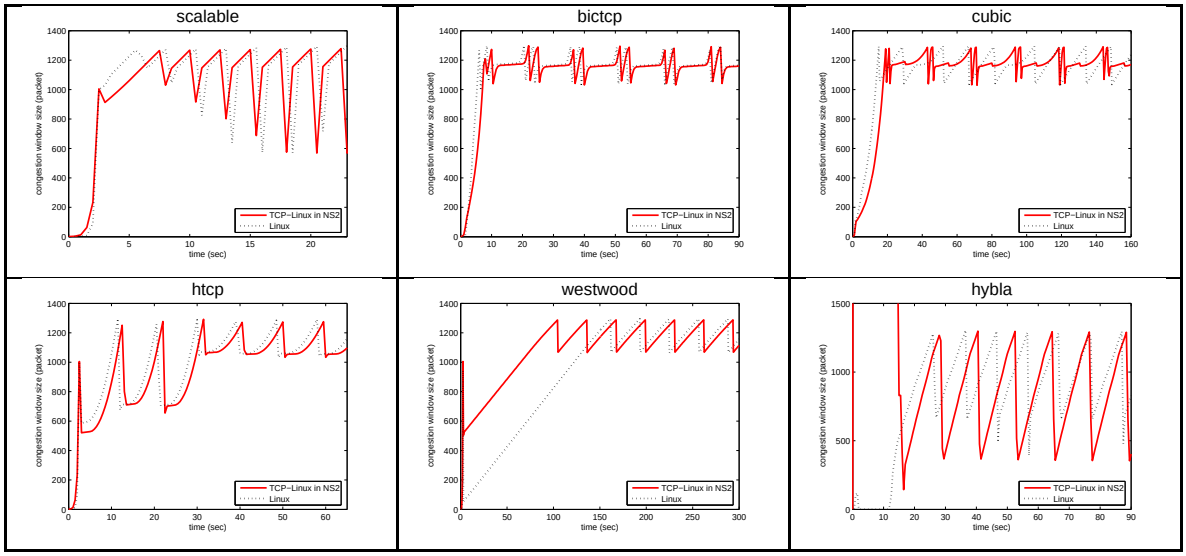


Table 4.2: Congestion window trajectory of different congestion control algorithms

parameters. Due to noise, the Linux host measures higher delay in the Dummynet testbed than in the NS-2 simulation. The higher delay leads to higher AI parameter and shorter length of congestion epoch in the Linux result. Also, TCP-Hybla sets a large congestion window in the start-up phase of a flow. The high congestion window leads to packet loss, but Linux quickly gets a timeout and *TCP-Linux* predicts multiple fast-recoveries before timeout. This results in the difference of the congestion window sizes at the start-up phase (though the rates predicted by *NS-2 TCP-Linux* are very similar to the Linux results). For TCP-Cubic, there are some difference in both the congestion widow trajectory and the length of congestion epoch. We have not understood this case yet. Further investigation is necessary.[7]

4.2.4.2 Accuracy

To compare the accuracy of *NS-2 TCP-Linux* and existing TCP implementations in NS-2, we compared the results by Dummynet testbed, the simulation results by *TCP-Linux,* and the simulation results by *NS-2 TCP-Sack1* or *NS-2 TCP-Vegas*, as shown in Table 4.3 .

In general, simulation results with *TCP-Linux* were much closer to the Linux

[7]As reported by the group that designs TCP-Cubic, the difference in TCP-Cubic was due to a bug in the Linux source code and a bug in *NS-2 TCP-Linux* at the time we ran these experiments.

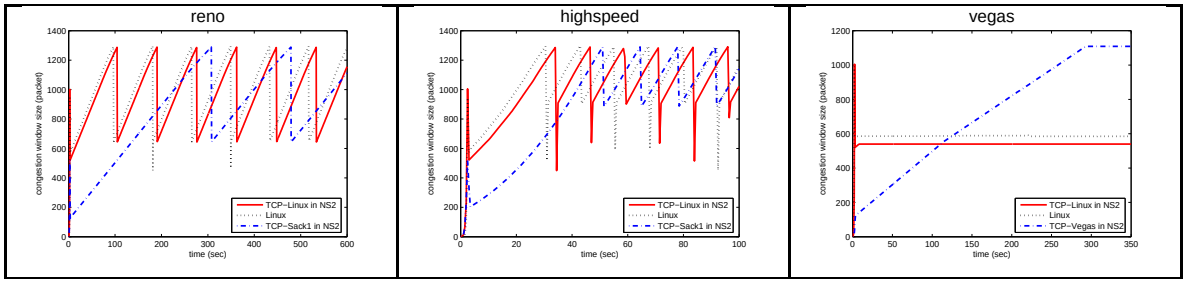


Table 4.3: Congestion window trajectory of Reno, Highspeed TCP and Vegas

results, especially for Reno and Vegas. In the case of Reno, the huge difference between *TCP-Sack1* and Linux is mainly due to the appropriate byte counting [71] in Linux Reno. The Vegas case is even more interesting. Both *TCP-Linux* and Linux results have smaller congestion windows than the *TCP-Vegas* results. After careful investigation, the combination of delayed ACK and integer operation is found to be the source of the problem. With delayed ACK, two packets are sent into the network in a micro-burst. The second data packet is buffered until the first packet is processed by the bottleneck router. Hence, the second packet is delayed for one packet worth of time even without congestion. Unfortunately, the Vegas sender can only get the acknowledgment of the second packet, due to delayed ACK. This second packet's RTT includes most of the queueing delay introduced by the first packet in the micro-burst.[8] Such a queueing delay is equivalent to *almost* one packet in the queue. With integer operation in Linux implementation, Vegas sees one packet in the queue. Since Vegas sets its α parameter to be 1, it stops increasing its congestion window when it sees this one packet worth of delay and results in low throughput. However, TCP/Vegas in NS-2 uses high-resolution float numbers to calculate the available and expected bandwidths and converts the results to integers only at the last step of comparison, avoiding this problem.[9]

We also ran simulations with different per-packet loss rates in the bottleneck and compared the throughput with the Linux throughput in Dummynet experiments.

[8] There is still a small gap between the two packets in a burst. The gap depends on the edge link capacity, which is 10 times of the bottleneck link in our simulations and experiments.

[9] This problem has been accepted by the Linux networking group and the default value for α has been changed to 2 in the new releases of Linux.

Figure 4.8 shows the throughput of a single TCP Reno flow (ran for 600 second) under different per-packet loss rates. Each experiment or simulation is repeated 10 times and we present the average and error-bar in the figure.

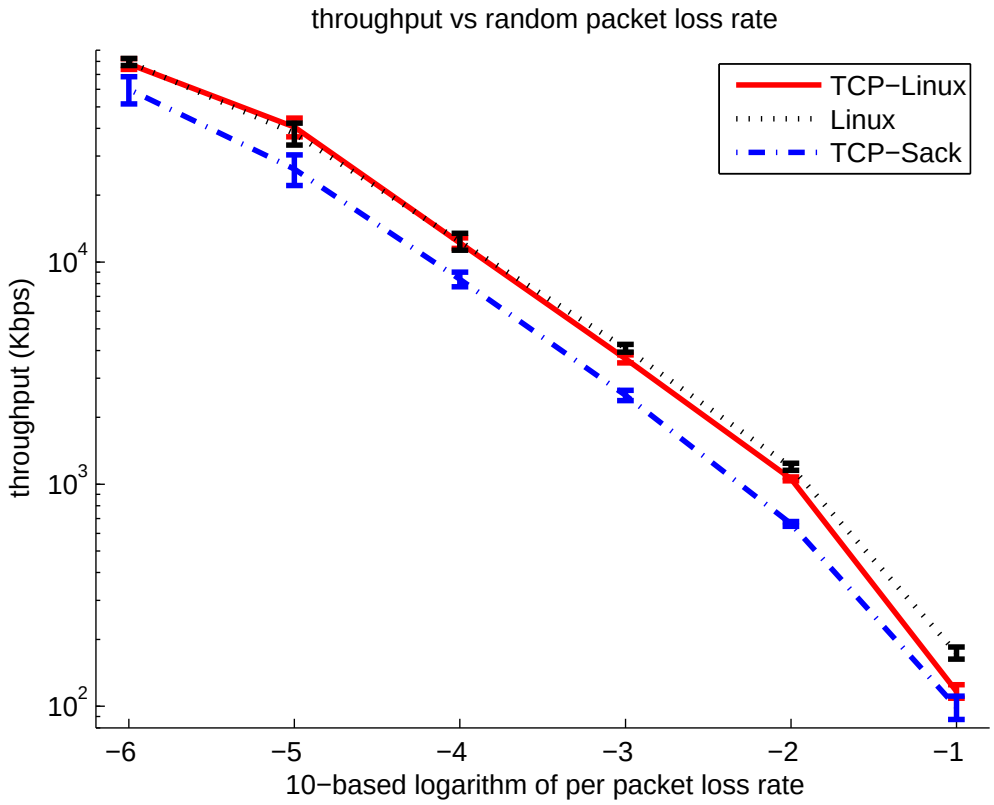


Figure 4.8: Throughput under different random loss rate (log-log scale)

As shown in Figure 4.8, the results with *TCP-Sack1* in NS-2 have a constant gap from the Linux performance. This gap, in log scale, implies a constant ratio in throughputs. This is mainly due to two problems. First, *TCP-Sack1* uses Scoreboard-RQ to process SACK. This module cannot detect the loss of retransmitted packets, and forces the TCP flow to time out when a retransmitted packet is lost. This leads to poor performance when packet loss is heavy. Second, Linux has incorporated appropriate byte counting [71], but TCP/Sack1 has not implemented this feature, leading to slower growth in the congestion window when packet loss is light.

The simulation results of *TCP-Linux* are very similar to the experiment results of Linux, except in the case when the per packet loss rate is 10%. In this case, the Linux

receiver almost disabled delayed acknowledgment, which leads to better performance than the simulation, where the delayed acknowledgment function in *TCPSink* in NS-2 is not adaptive.

4.2.4.3 Simulation performance

We run simulations with Reno and HighSpeed TCP with different numbers of flows (from 2 to 128), different round trip propagation delays (from 4ms to 256ms) and different capacities (from 1Mbps to 1Gbps) to test the speed and memory usage. We also compared the simulation performance of *TCP-Linux* with the performance of *TCP-Sack1* in NS-2. Each case simulates the scenario for 200 seconds. All the simulations were run on a 1U server, with two Intel Xeon 2.66GHz CPUs, a cache size of 512KB, though only one of the CPUs can be used for each simulation. We make sure that only one simulation was run at a time, without any other application. We present here two of our dozens of figures. Figure 4.9 shows the simulation times of HighSpeed TCP with different bottleneck capacities. Figure 4.10 shows the simulation times of HighSpeed TCP with different numbers of flows.

Both figures show that the speed of *TCP-Linux* is very similar to the speed of *TCP-Sack1* module in most scenarios. However, *TCP-Sack1* does not perform consistently well and might have a much longer simulation time when the capacity is high, or the number of flows is large. *NS-2 TCP-Linux* has a very consistent simulation speed with its improved event scheduler.[10]

To measure the memory usage of the simulation, we measured the simulator's memory usage in the middle point of the simulation period. The memory usage of *TCP-Linux* was almost the same as *TCP-Sack1* in most scenarios. The only difference we observed was the case with Reno and with two flows, as shown in Figure 4.11 . In this case, *TCP-Linux* used about 1MByte more than *TCP-Sack1*.

Based on these simulation results, we believe that *TCP-Linux* can be a good alternative, or even a replacement, for the existing NS-2 TCP modules, given its

[10] We also ran *NS-2 TCP-Sack1* with our improved scheduler. With the improved scheduler, *TCP-Sack1* performed exactly the same as *TCP-Linux*.

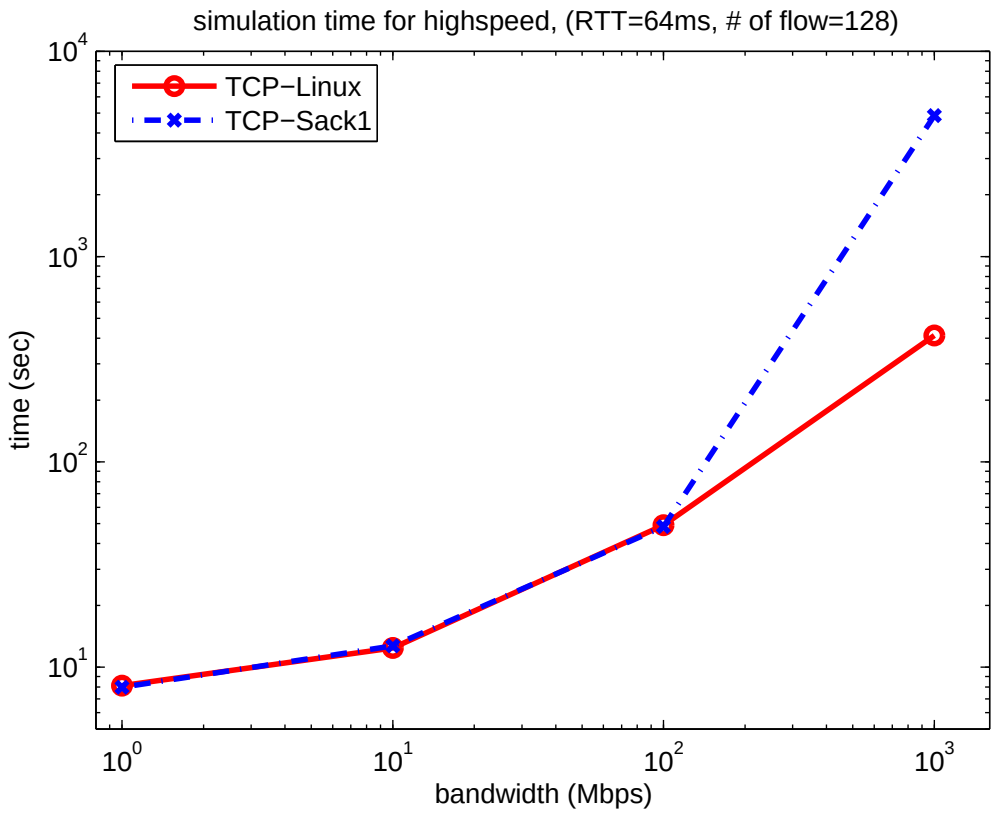


Figure 4.9: Simulation time of different bottleneck bandwidth (log-log scale)

similar performance in terms of speed and memory usage, and its advantages in terms of adaptability and accuracy.

4.2.4.4 An example: identifying a potential bug in Linux HighSpeed TCP implementation

Figure 4.12 shows an example of how *TCP-Linux* can help the Linux community test and debug the implementation of new congestion control algorithms.This figure illustrates a potential bug in HighSpeed TCP implementation in Linux 2.6.16-3. In the rare situation when $snd\_cwnd\_cnt\_ == snd\_cwnd\_$, $snd\_cwnd\_$ is increased by one, *before* $snd\_cwnd\_cnt\_$ is decreased by snd_cwnd_. This leads to a value of -1 for $snd\_cwnd\_cnt\_$. Since $snd\_cwnd\_cnt\_$ is an unsigned variable, the negative

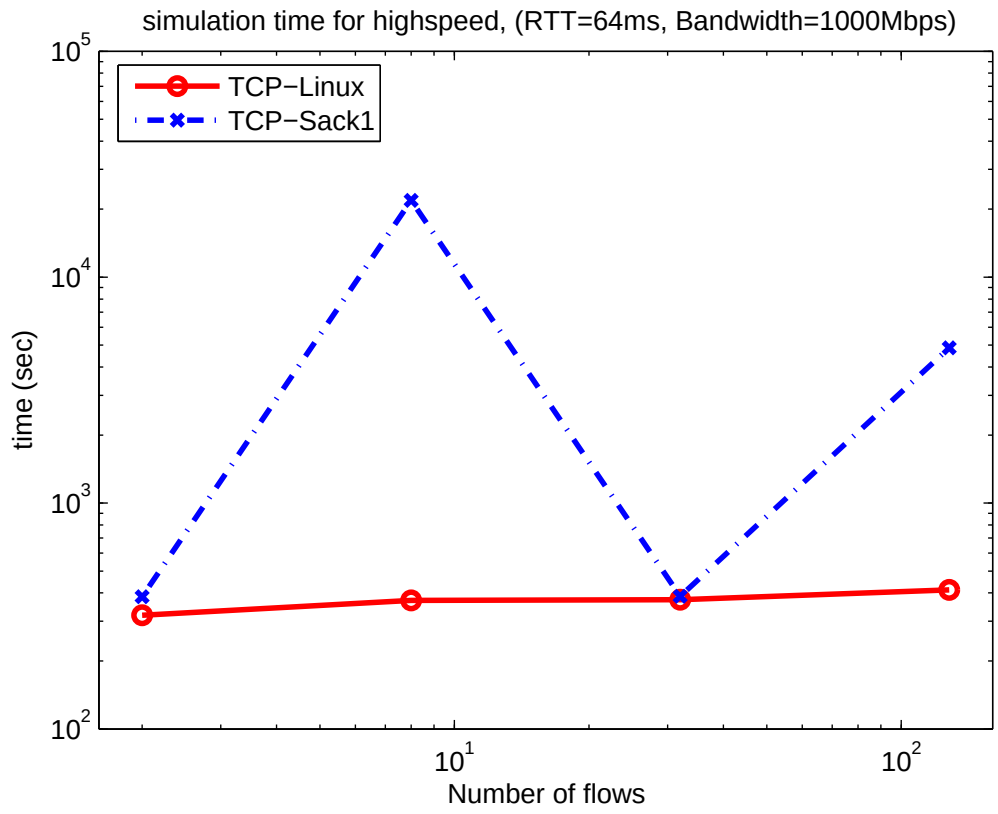


Figure 4.10: Simulation time of different number of flows (log-log scale)

value causes overflow, results in an infinitely large *snd_ cwnd_ cnt_*.[11]

Motivated by this example, we strongly believe that *TCP-Linux* can help the implementation community debug, test, and understand the new congestion control algorithms and close the gap between the implementation and analysis communities.

4.2.5 Usages in research

In our research, *NS-2 TCP-Linux* is used to simulate different TCP variants (TCP-Reno, HighSpeed TCP and Scalable TCP) and study their performance under different packet loss patterns.

We used a dumb-bell topology with a set of senders and a set of receivers sharing a bottleneck, as shown in Figure 4.13.

[11] The Linux networking group has accepted this bug report and the bug hass been corrected in the newer Linux releases.

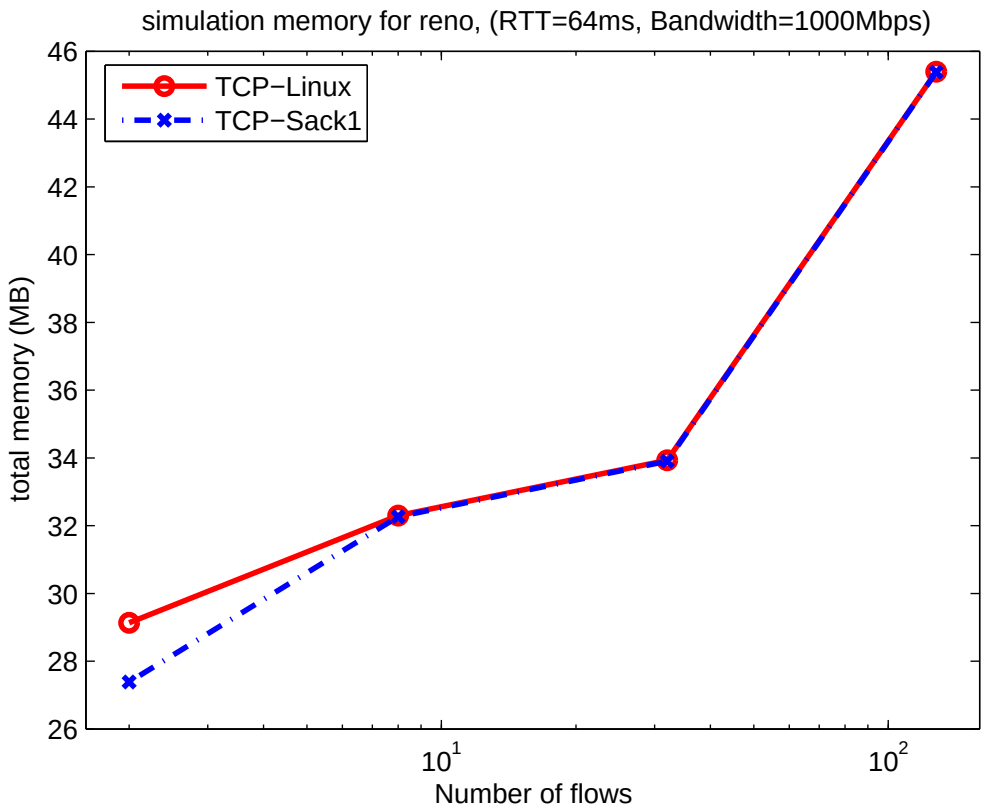


Figure 4.11: Memory usage of different number of flows (x-axle in log scale)

The bottleneck is a 100Mbps link. We ran simulations with different RTT (2ms to 200ms in different scenarios) and different buffer sizes (1/8 bandwidth-delay-product to 2 bandwidth-delay-product in different scenarios). Different link algorithms, Drop-Tail, RED and Persistent ECN, are run in the simulations. For scenarios with RED, the RED parameters are set to be self-adaptive. The bottleneck is shared by a set of parallel TCP flows and 50 UDP noise flows, generating exponential on-off traffic with the aggregate rate of 10% of the bottleneck capacity. The number of flows varies from 2 to 32 flows in different scenarios.

The TCP implementations in *NS-2 TCP-Linux* come with SACK [56], FACK [7] and rate halving [64]. Reno implementation also includes ABC [71].

In each simulation, we repeat the scenario for at least 10 times with different random seeds and present both the average values and the error-bars.

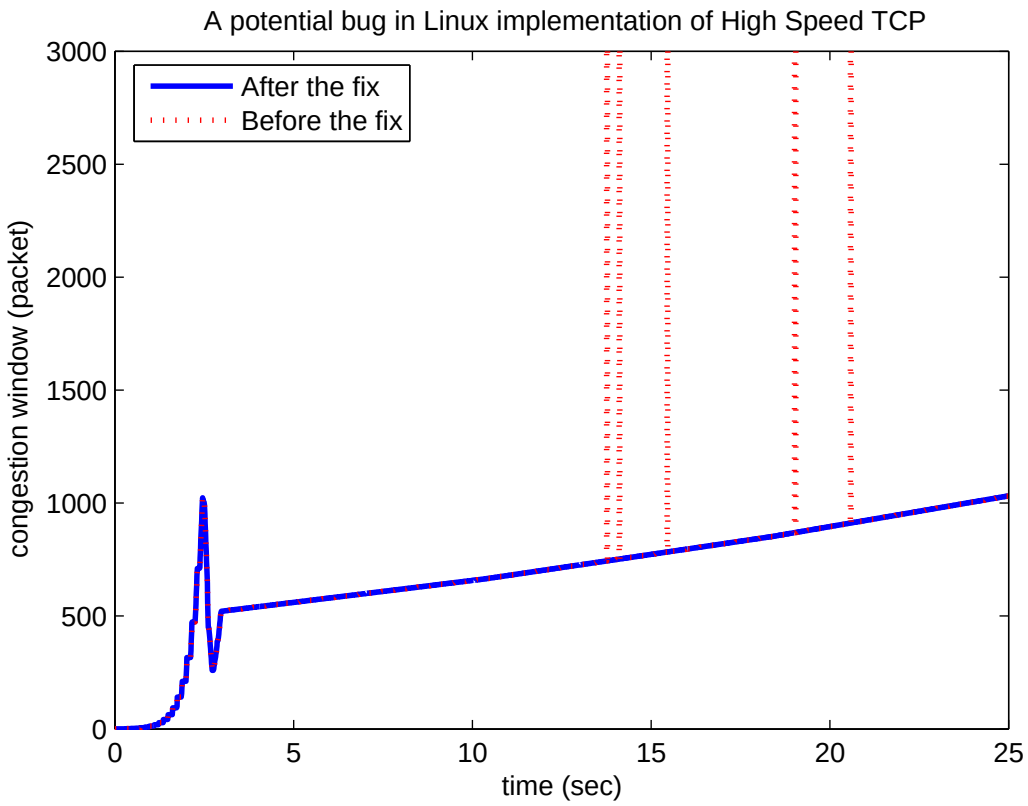


Figure 4.12: A potential bug in Linux implementation of HighSpeed TCP

4.3 A packet level measurement tool in PlanetLab

To measure the Internet behavior, we have designed a distributed measurement system to periodically measure loss and delay pattern between sites in PlanetLab.This measurement system is different from existing loss measurement techniques.

The most frequently used technique for loss measurement is to analyze traces of TCP flows and detect packet retransmissions in TCP. Each packet retransmission can be treated as the inference of a lost packet in the network. This approach is widely used because it has the advantage of requiring measurement at one end only [41]. However, there are a few disadvantages. First, a packet retransmission only infers a possible packet loss. Whether there is really a packet loss in the network depends on the accuracy of the TCP retransmission algorithm. If TCP times out prematurely, this approach overestimates the packet loss rate. Second, and more importantly, this approach measures the packet loss rate observed by TCP flows. As we will see in

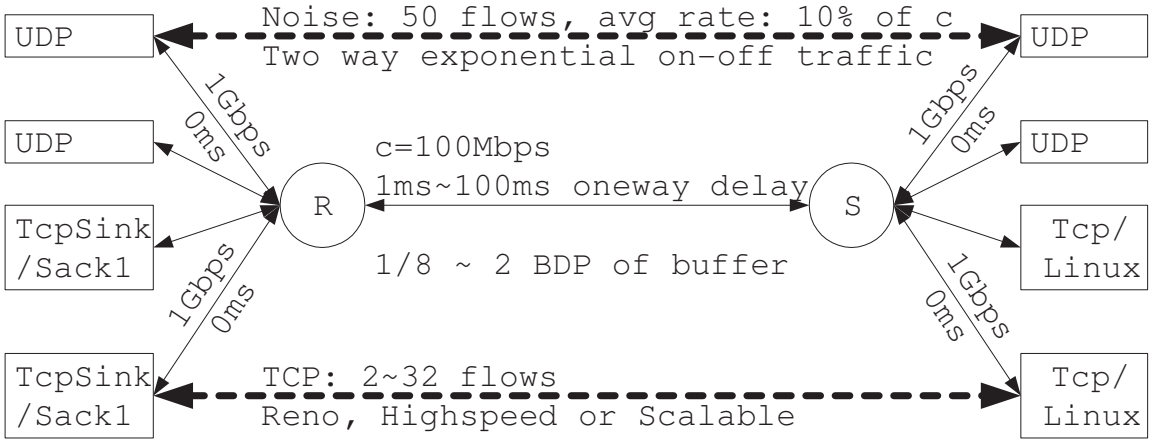


Figure 4.13: Setup of NS-2 simulations

Chapter 4, the packet loss rate observed by TCP flows is not necessarily the same as the packet loss rate in the path, due to TCP burstiness.

In our tool, we used UDP packets to probe the network and measure the loss information at the other end. We designed our tool in a way that the UDP packet transmission pattern is tunable. We used different transmission patterns to simulate bursty packet sequences (e.g. generated by TCP) or smooth packet sequences (e.g. generated by TFRC [72], paced TCP) .

4.3.1 An introduction to PlanetLab

PlanetLab [45] is an open platform for developing, deploying and accessing planetary-scale service. It is an overlay network with more than 600 nodes geographically distributed around the world. PlanetLab users can run services on a set of nodes with full control on the end hosts, making it possible to measure packet loss at two end hosts.

4.3.2 Design of the measurement system

The system has a server and a client at each site in the experiments. There is also a light-weight central monitor running on one site (in our experiments, the central monitor is run on WAN-in-Lab at Caltech). The central monitor periodically performs

maintenance tasks of the experiments. These tasks include: installing and upgrading software at each remote site, monitoring the health of each remote site, and collecting measurement data from all the sites into a centralized database. Each remote site runs a server, which accepts new measurement requests and UDP packet sequences. Each remote site periodically starts a client, which randomly picks a site and initiates new measurement request. The client is the sender of UDP packet sequences and the recorder of the experiment results.

In each measurement, server and client use a TCP connection to exchange control messages and measurement results. After a simple handshake, the client sends UDP packets to the server in a specific pattern. The server measures loss information and periodically returns statistics to the client via TCP. The server keeps a window of 128 packets so that packet reordering within 128 packets can be detected. These reordered packets will not be considered lost packets.[12]

4.3.2.1 Message formats

There are four different packets in the system. We carefully designed the packet formats so that we could filter unreliable results that were probably due to abnormal network behavior, for example, packet corruption. Each packet starts with an 8-byte identifier, filled with 8 English characters. This identifier indicates the packet type. The servers and clients verify the identifier and the validity of the contents. For UDP packets, the servers and clients also use checksum to detect possible packet corruption.

StartPacket StartPacket is the first packet the server sends to the client to confirm the acceptance of the measurement request. Before sending this packet, the server assigns a unique measurement request ID and the StartPacket carries this request ID back to the client. This request ID will be in all UDP packets sent from the client to the server. Table 4.4 illustrates the format of StartPacket.

[12]This is much more robust than the normal TCP, which can only detect packet reordering within 3 packets.

STARTUDP	<request id>
8 bytes	8 bytes

Table 4.4: StartPacket format

STOP_UDP	<error code>
8 bytes	8 bytes

Table 4.5: StopPacket format

StopPacket StopPacket is the packet that the server or the client sends to indicate the termination of the experiments. Whenever an error is detected, either server or client will send a stop packet, with an error code to inform the other side of the reason of the termination. A error code of zero indicates normal termination. Table 4.5 illustrates the format of StopPacket.

UDPPacket UDPPackets are the packets sent from the client to the server, which measures the delay and loss in the forward path. As illustrated in Table 4.6, each UDP packet carries a request ID for the measurement, a unique sequence number, the timestamp of sending event, and variable length of random bytes that fills the packet to a specific length. A UDPPacket also comes with a checksum that covers the whole packet.

ReportPacket A ReportPacket is the packet sent from the server to the client to report the statistics of packet loss and measure round-trip time (RTT). As illustrated in Table 4.7, a ReportPacket carries the timestamp of the UDPPacket upon whose arrival the ReportPacket is sent. When the client receives this ReportPacket, it can estimate the round-trip time by the difference of the receiving time and the value in

UDP_TEST	<request id>	<seq #>
8 bytes	8 bytes	8 bytes
<timestamp>	<checksum>	<random bytes>
16 bytes	8 bytes	variable

Table 4.6: UDPPacket format

L_REPORT	<timestamp>	<ack seq #>
8 bytes	16 bytes	8 bytes
<# of loss in this report>	<loss seq numbers>	
8 bytes	8*<# of loss> bytes	

Table 4.7: ReportPacket format

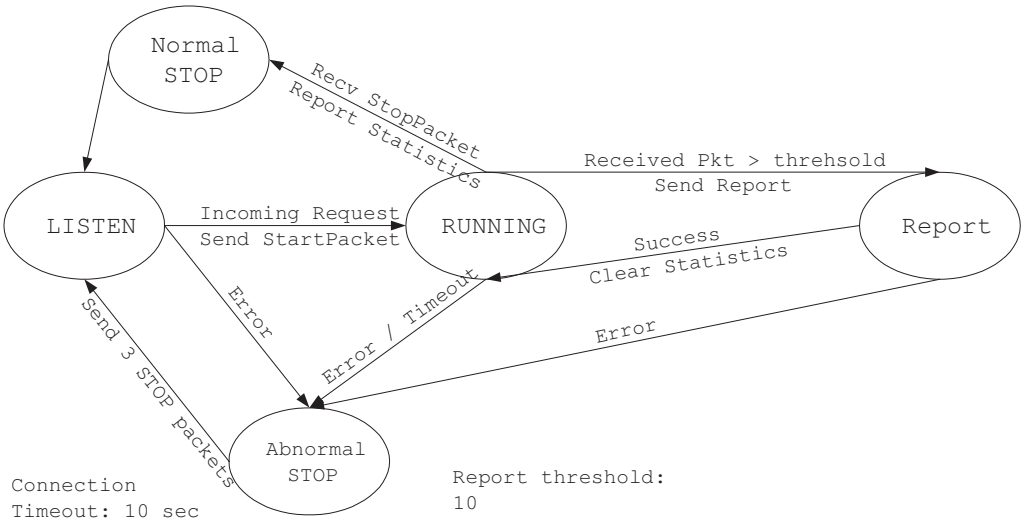


Figure 4.14: State machine of a measurement server in PlanetLab

<timestamp> field.[13]

A ReportPacket also carries the highest sequence number of the UDPPacket it receives. This serves as a positive acknowledgment. Finally, a ReportPacket carries the number of observed loss packets, followed by the list of sequence numbers of these lost packets. The client will record these lost packets and their sending times to estimate the loss pattern in the path.

4.3.2.2 Design of measurement servers

The server listens to a static port and accepts, at most, 10 incoming measurement requests. The design of the server is shown in the state machine in Figure 4.14. For each request, the server assigns a unique ID and returns the ID in a StartPacket. The

[13]As ReportPackets are transported by TCP, the RTT calculation is not very accurate if the ReportPacket is lost. We only use this RTT information to get a rough estimation of the RTT and we use the minimum observed RTT as our estimate. This estimation ignores the queueing delay in the path and underestimates the RTT. Hence, it underestimates the sub-RTT burstiness.

server then enters the RUNNING state. In the RUNNING state, it listens to the port for any UDP packets that match the ID for the request. When the server receives more than 10 packets, or it detects more than 128 packet loss, it sends a report packet to the client via TCP connection. The server stays in the RUNNING state until it receives a StopPacket from the client. When the server receives a StopPacket, it sends the last report and waits for a normal TCP exit. At any time if there is abnormal behavior, the server enters the AbnormalStop state, sends a StopPacket to the client, and closes the connection.

Abnormal behavior is defined as one of the following symptoms:

- A read or write error in TCP socket;
- A corrupted UDP packet;
- The measurement has been idle for more than 10 seconds.

For these abnormal behaviors, the client will receive a StopPacket with an error message, and the result in this measurement will be discarded.

4.3.2.3 Design of measurement clients

The design of the client is shown in Figure 4.15.

It sends a measurement request to a server which is randomly chosen from a list, and waits for the StartPacket. Once the StartPacket arrives, it enters the Operation state, in which it sends UDP packets in a specific pattern. Upon the receipt of a ReportPacket, it calculates the RTT estimation and records the delay and loss information into a file. When the measurement period is over, the client exits the Operation state by issuing a StopPacket with error code zero and waits for the last report. The client stops upon timeout or the arrival of the report.

Each site runs the client randomly every 30 minutes. Each measurement period lasts 5 minutes. The client sends UDP at a rate less than 1 packet per ms. Each UDP packet is no greater than 400 bytes .[14] Hence, the average throughput over the mea-

[14]The maximum packet size of 400 bytes is to guarantee that the packet will not be fragmented by a router with a small MTU of 500.

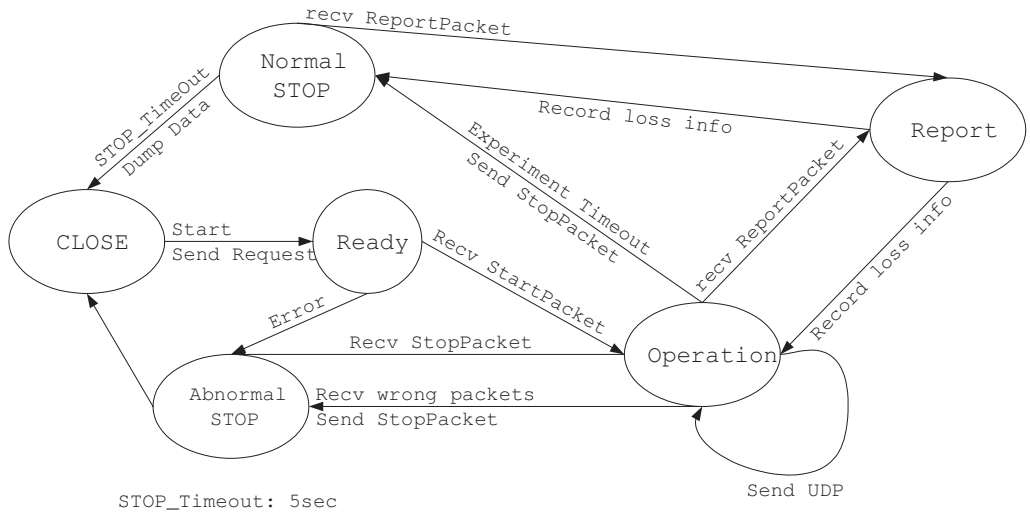


Figure 4.15: State machine of a measurement client in PlanetLab

surement period is no greater than 3.2Mbps (with 400 bytes per packet). The average throughput is only 384Kbps with 48-byte packets. We usually ran experiments with packet sizes of 48 bytes or 400 bytes and compared the results to make sure that our measurement traffic did not inject additional congestion. Before each measurement, we ran ping and traceroute to record the RTT and the routing paths. The RTT reported by ping will be used to compare the delay measurement and validate the results. The routing path information can be used to detect route changes and path intersections. Currently, we only use it to calculate hop count of a path.

4.3.3 Deployment and data pre-processing

We deploy this measurement system in 26 sites in the PlanetLab . In each measurement, the client in a site randomly picks a server. Hence, 650 directional paths are measured. As listed in Table 4.8, our sites are geographically located on different continents. Among them, 6 are in California, 11 are in other parts of United States, 3 are in Canada and the rest are in Asia, Europe, and South America. The RTTs of these paths have a range from 2ms to more than 200ms.[15]

Our system uses a central monitor to maintain the sites in the experiments. The

[15]The highest measured RTT is more than 300ms, depending on the time of the day.

Node	***Location***
planetlab2.cs.ucla.edu	Los Angeles, CA
planetlab2.postel.org	Marina Del Rey, CA
planet2.cs.ucsb.edu	Santa Barbara, CA
planetlab11.millennium.berkeley.edu	Berkeley, CA
planetlab1.nycm.internet2.planet-lab.org	Marina del Rey,CA
planetlab2.kscy.internet2.planet-lab.org	Marina del Rey,CA
planetlab3.cs.uoregon.edu	Eugene, OR
planetlab1.cs.ubc.ca	Vancouver, Canada
kupl1.ittc.ku.edu	Lawrence, KS
planetlab2.cs.uiuc.edu	Urbana, IL
planetlab2.tamu.edu	College Station, TX
planet.cc.gt.atl.ga.us	Atlanta, GA
planetlab2.uc.edu	Cincinnati, Ohio
planetlab-2.eecs.cwru.edu	Cleveland, OH
planetlab1.cs.duke.edu	Durham, NC
planetlab-10.cs.princeton.edu	Princeton, NJ
planetlab1.cs.cornell.edu	Ithaca, NY
planetlab2.isi.jhu.edu	Baltimore, MD
crt3.planetlab.umontreal.ca	Montereal, Canada
planet2.toronto.canet4.nodes.planet-lab.org	Toronto, Canada
planet1.cs.huji.ac.il	Jerusalem, Israel
thu1.6planetlab.edu.cn	Beijing, China
lzu1.6planetlab.edu.cn	Lanzhou, China
planetlab2.iis.sinica.edu.tw	Taipei, China
planetlab1.cesnet.cz	Czech Republic
planetlab1.larc.usp.br	Brazil

Table 4.8: PlanetLab sites in measurement

monitor ensures that the client and server at each site run normally. It performs tasks such as upgrade, data collection, and cleanup.

From October 2006 to December 2006, we periodically initiated constant bit rate (CBR) flows between two randomly selected sites. For each experiment, two runs of measurements were conducted: one with a packet size of 48 bytes and the other with a packet size of 400 bytes. We compared these two results and validated the measurement only if the two traces exhibited similar loss patterns. Hence, the effect of traffic load from our own measurement CBR flows was negligible in our measurement results. Each run lasted for 5 minutes. In analysis, we normalized the loss interval by the RTT of the path.

Chapter 5

Conclusions and Future works

This thesis studies the microscopic behavior of TCP congestion control. We find that the burstiness at the microscopic level has huge impacts on the stability, efficiency, fairness, and convergence of congestion control algorithms.

We categorize burstiness into two types: micro-burst and sub-RTT burstiness.

Micro-burst leads to quick convergence of queueing delay. This effect makes delay-based congestion control algorithms much more stable than the fluid model predicts. In particular, homogeneous TCP-Vegas flows or FAST flows are globally stable regardless of capacity and delay. This new prediction agrees with the experimental results and are in sharp contrast to the fluid model prediction. With this observation, we can design new algorithms that achieve both responsiveness and stability.

Sub-RTT level burstiness leads to on-off patterns in the data packet transmission process. With DropTail routers producing bursty loss processes in sub-RTT timescale, sub-RTT level burstiness directly affects the *loss synchronization rate*, the probability that one flow sees a packet loss during a congestion event. Loss synchronization rate affects the efficiency, fairness, and convergence of loss-based congestion control algorithms. With all these understandings, we can explain controversial problems such as MIMD fairness, competition between paced TCP and bursty TCP, and slow convergence of TCP.

We developed two new tools in the research process. The *NS-2 TCP-Linux* is a simulation module that runs Linux source codes directly. It is extensible for new congestion control implementations, runs faster, and produces more accurate results.

Since its introduction, *NS-2 TCP-Linux* has helped the Linux community identify several implementation defects and parameter tuning problems. We expect that this tool will be a bridge between the Internet engineering community and research community. The PlanetLab-based Internet loss measurement system uses UDP packets to measure packet loss rates along different paths, with different packet arrival patterns and different packet sizes. Different from the traditional TCP-based loss monitor tools, the measurement results from this system are not affected by the bursty sampling process of TCP.

The work in this thesis is the beginning, rather than the conclusion, of many new questions. Future work is moving in several directions:

5.1 Packet Level Model for Delay-based Congestion Control Algorithm

The packet level model used in this thesis has new predictions sharply different than results from the commonly-used models. These new predictions agree with the experimental results. However, the packet level model is only accurate in a single flow scenario and can only be extended to multiple homogeneous flows. We are still in search of a good way to extend the model to cases with heterogeneous flows. One possible direction is to use a two-level on-off processes, instead of one-level on-off process, to model the packet level dynamics.

5.2 Application of the model for loss synchronization rate

We have applied our understanding to explain several interesting and long-standing problems. There are many other problems which we believe our findings can resolve. These problems include:

- Friendliness between TFRC and TCP. TFRC claims to be friendly to TCP, but

however, experimental results show that TFRC is too friendly and usually loses to TCP, although the design of TFRC follows the TCP response function as accurately as possible at the macroscopic level. We expect that the main reason for the loss of TFRC is due to the difference at the microscopic level. Specifically, the data packet transmission pattern of TFRC is similar to the pattern of paced TCP, which leads to a higher loss synchronization rate than the one observed by TCP sharing the same bottleneck. This different synchronization rate leads to lower throughput for TFRC, when competing to TCP.

- Effects and extensions of RED. According to our model, RED is supposed to help the fairness convergence of MIMD protocols. We will study this effect to test our predictions. We also expect to use our research to help the parameter tunings in RED.

- Loss measurement methodology. Most existing loss measurement tools use TCP-based probing flows. This approach is easy to deploy (control is only required on one side) but prone to the effect of burstiness in TCP. We need to study further the implication of the loss measurement results from these TCP-based tools and compare the accuracy against our UDP-based tools.

5.3 Improvement of new algorithms

The Persistent ECN algorithm is deterministic and subject to phase effects. We plan to investigate the performance of the algorithm under a variety of application loads and patterns, and investigate the possibility of randomizing the algorithm. The future algorithm we expect will be a hybrid of persistent ECN and RED, which will provide a persistent random signal in a congestion event.

5.4 Extension to *NS-2 TCP-Linux*

The current status of this NS-2 module has its limitation. It might not be able to simulate the Linux performance well in the case where the packet reordering in the path is severe, or packet loss rate is extremely high. In future work, we plan to include D-SACK [65] processing and the congestion window reduction undo function from Linux. We are also considering developing a Delayed Ack module for NS-2 that performs similar to Linux. Finally, we need to incorporate the ECN and F-RTO functions in Linux to NS-2.

Furthermore, *TCP-Linux* provides a very good platform for testing different TCP congestion control protocols with the flexible scenarios in NS-2. This can be a good foundation towards a benchmark suite implementation for TCP congestion control algorithms. We do plan to enhance our benchmark suites and summarize a set of NS-2 scenarios for the benchmark.

Finally, we plan to extend our simulation framework to include a more detailed model for distributed applications. Currently, we only use parallel flows as the application. We plan to simulate more complicated scenarios such as a complete graph topology in MapReduce [73].

Chapter 6

Appendix

6.1 Complete list of control variables and functions ported by NS-2 TCP-Linux

6.1.1 Control variables:

6.1.1.1 Local variables for each connection:

snd_nxt: The next sequence that the flow is going to send.

snd_una: The next sequence that the flow is waiting for acknowledgment

mss_cache: The size of a packet

srtt: 8 times of the smooth RTT

rx_opt.rcv_tsecr: The timestamp echoed by the acknowledgment

rx_opt.saw_tstamp: Whether there is a timestamp in the acknowledgment

snd_ssthresh: the slow start threshold

snd_cwnd: the congestion window

snd_cwnd_cnt: the congestion window counter, since congesition window is in unit of packet, when the congestion window increment is smaller than one, snd_cwnd_cnt is increased instead. Whenever snd_cwnd_cnt is larger or equal to snd_cwnd, the snd_cwnd_cnt shall be decreased by snd_cwnd and snd_cwnd shall be increased by 1.

snd_cwnd_clamp: the upperbound of the congestion window

snd_cwnd_stamp: the last time that the congestion window is changed

bytes_acked: the number of bytes that are acknowledged in this acknowledgment

icsk_ca_state: the state of congestion control: Normal (OPEN), Loss Recovery, or time out.

icsk_priv: a sixteen 32bit integer array for private data of congestion control algorithm

6.1.1.2 Global variables:

Besides the tcp_sk structure, there are several global variables which are important for congestion control algorithms:

tcp_time_stamp: the current time in ms

sys_ctl_abc: whether and how the congestion control algorithm shall do Appropriate byte Counting.

tcp_max_burst: the maximum back-to-back burst that a TCP flow can send into the network.

6.1.2 Function interfaces:

6.1.2.1 Required functions:

- *cong_avoid* function: This function is called when an ack is received. The implementation of this function is ecpected to change the congestion window in this function during normal situation (without loss recovery). In Reno, this means slow start and congestion avoidance.

- *ssthresh* function: This function is called when a loss event happens. It is expected to return the slow start threshold after a loss event. The returned value shall be half of *snd_cwnd* in Reno.

- *min_cwnd* function: This function is called when a fast retransmission happens, after *ssthresh* function. It is expected to return the value of the congestion window after a loss event. In Reno, the returned value shall be *snd_ssthresh*.

6.1.2.2 Other optional function calls include:

RTT sample function (rtt_sample): called when a RTT sample is detected, a congestion control algorithm shall implement this function if it has special requirement on RTT sampling;

State change function (set_state): called when the congestion control state is changed (among Open (normal state) state, Loss Recovery state, and Loss (timeout) state).

Congestion event function (cwnd_event): called when there is a special event that might be interesting for a congestion control algorithm. The possible congestion events include: TX_START, CWND_RESTART, COMPLETE_CWR, FRTO, LOSS, FAST_ACK and SLOW_ACK.

The congestion window after loss (undo_cwnd): called when the flow exists loss recovery.

Packet Acked function (pkts_acked): called when a packet is acknowledged.

Initialization function (init): called when the congestion control algorithm is loaded. Any private data in icsk_priv shall be intialized here;

Destroy function (release): called when the congestion control algorithm is removed, private data shall be deleted here.

Algorithm 4 Randomized Pacing

For each flow i, given $w_i(t)$ from congestion control algorithm; t is the current system time.
$w_i^{'}(t) = \max\{w_i(t) + \alpha, \min\{2w_i(t), ssthresh(t)\}\}$ is the predicted value of the congestion window in the next RTT.
$v_i(k)$ is the time when the k-th packet of flow i is supposed to be sent according to pacing algorithm, $v_i(0) \leftarrow 0$.
For each packet to be transmitted (allowed by the congestion window and receiver window constraints):

1. $v_i(k) \leftarrow v_i(k-1) + \frac{RTT}{w_i'(t)}$
2. $s \leftarrow v_i(k) + \text{random}\left[-\frac{RTT}{2w_i'(t)}, \frac{RTT}{2w_i'(t)}\right]$
3. send k-th packet at time $\min\{s, t\}$

6.2 A randomized version of pacing

There have been many proposals of different pacing algorithms [20, 21]. The pacing algorithm suggested in these literatures is a deterministic algorithm which strictly pace out packets evenly in an RTT. We observed that this approach is very likely to introduce phase effect in our simulations.

Our algorithm randomizes the paced TCP by perturbing the paced packet transmission time with a zero-sum uniform random offset. The advantage of a randomized pacing algorithm over a deterministic pacing algorithm is that it can eliminate phase effect. In this sense, it is similar to the randomized TCP algorithm proposed by Chandrayana, et al [49]. In contrast to this research, we randomize a pacing algorithm instead of a bursty TCP to make the packet distribution more spread-out.

Algorithm 4 describes the detailed steps.[1] For each flow i, the congestion control algorithm specifies a sending window $w_i(t)$ and the pacing algorithm predicts the window in the next round-trip $w_i^{'}$. The pacing algorithm replaces the *ack-clocking* and controls the exact time that a packet is transmitted in sub-RTT time scales. The

[1] This algorithm description is simplified with an assumption that the network is always the bottleneck. Modification is necessary in real deployment where some flows may be application-limited.

algorithm keeps a virtual sending time v_i, which is the sending time of each packet if a standard pacing were enforced. The actual packet transmission time is uniformly distributed over the interval of $\left[v_i - \frac{RTT}{2w_i'}, v_i + \frac{RTT}{2w_i'}\right]$, so there is always one packet in every RTT/w interval while the phase effect is eliminated by randomizing the order of packets from different flows. If the loss signal is persistent for RTT/w or longer, the randomized pacing can ensure that all flows detect the same loss event with high probability.

6.3 Proofs of theorems

6.3.1 Theorem 2.1.2.1

For any time $s(j)$ in which a packet is sent into the network,

$$p(j) \leq w(s(j))$$

And there always exists a packet j^* which is sent at the same time ($s(j) = s(j^*)$), and

$$p(j^*) = w(s(j^*))$$

Furthermore, if $w(s(j^*)) \geq w(s(j^*+1))$, $p(j^*+1) = w(s(j^*+1))$.

Proof:

First, we prove that $p(j) \leq w(s(j))$ for any packet j.

Assume k^* is the parameter that achieve $p(j) = p(j-1) - k^* + 1$ and $p(j-1) - k^* + 1 \leq w(a(j-1-p(j-1)+k^*))$. We have:

$$j - p(j) = j - 1 - p(j-1) + k^*$$

We have $s(j) = a(j-p(j)) = a(j-1-p(j-1)+k^*)$ and $w(s(j)) = w(a(j-1-p(j-1)+$

Hence, $p(j) \leq w(s(j))$.

Second, we prove that $p(j) = w(s(j))$ for all $j$such that $p(j) \geq p(j+1)$.

If $p(j) \geq p(j+1)$, we have: $p(j)+1 > w(s(j))$. (Otherwise, $p(j)+1 \leq w(s(j))$ leads to $p(j+1) = p(j)+1$.)

We have $p(j) \geq w(s(j))$ since $p(j)$ an $w(s(j))$ are both integers. Because $p(j) \leq w(s(j))$, we have $p(j) = w(s(j))$.

Finally, we prove that there is always a packet j^* which satisfies $s(j^*) = s(j)$ and $p(j^*) = w(s(j))$.

If $p(j) \geq p(j+1)$, $j = j^*$.

Otherwise, assume j^* does not exist.

Then for $\forall j' > j$, we have $p(j') < p(j'+1)$, $s(j) = s(j') = s(j'+1)$ and $p(j'+1) = w(s(j))$.

Hence, $p(j')$ is unbounded and $w(s(j))$ is unbounded.

This cannot happen since $w(s(j))$ is a finite number. Hence, such j^* always exist.

According to (2.7),

$$p(j^*+1) = \max_{0 \le k \le p(j^*)} \{p(j^*) - k + 1 | p(j^*) - k + 1 \le w(s(j^*+1))\}$$

Let $k^* = w(s(j^*)) - w(s(j^*+1)) + 1$, we have

$$\begin{aligned} p(j^*) - k^* + 1 &= w(s(j^*)) - [w(s(j^*)) - w(s(j^*+1)) + 1] + 1 \\ &= w(s(j^*+1)) \end{aligned}$$

Hence, $p(j^*+1) \ge w(s(j^*+1))$.

We have $p(j^*+1) = w(s(j^*)+1)$.

6.3.2 Theorem 2.1.2.2

$$\forall j : a(j) - a(j-1) \ge \frac{1}{c}$$

The equality holds if, and only if, $s(j) \le s(j-1) + \frac{b(J-1)+1}{c}$.

Proof:

$$\begin{aligned} a(j) - a(j-1) &= \left[s(j) + d + \frac{b(j)}{c}\right] - \left[s(j-1) + d + \frac{b(j-1)}{c}\right] \\ &= s(j) - s(j-1) + \frac{b(j)}{c} - \frac{b(j-1)}{c} \\ &= s(j) - s(j-1) + \frac{\max\{b(j-1) + 1 - [s(j) - s(j-1)]c, 0\}}{c} - \frac{b(j-1)}{c} \\ &\ge s(j) - s(j-1) + \frac{b(j-1) + 1 - [s(j) - s(j-1)]c}{c} - \frac{b(j-1)}{c} \\ &= \frac{1}{c} \end{aligned}$$

The first step is by (2.6) ; the third step is by (2.5); and the fifth step is by (2.1).

The equality in the forth step holds if, and only if, $b(j-1)+1-[s(j)-s(j-1)]c \geq 0$, which is equivalent to $s(j) \leq s(j-1) + \frac{b(j-1)+1}{c}$.

Hence, the theorem is proved.

6.3.3 Theorem 2.1.2.3

For $\forall 1 \leq j' < j$, If $p(j')$,$p(j'+1), \cdots p(j)$ are non-decreasing,

$$b(j) \leq b(j') + p(j) - p(j')$$

Proof:

Since $p(j') \cdots p(j)$ are non-decreasing, we have $\forall j' \leq k < j$:.

$$\begin{aligned} s(k+1) - s(k) &= a(k+1-p(k+1)) - a(k-p(k)) \\ &\geq \frac{[k+1-p(k+1)] - [k-p(k)]}{c} \\ &= \frac{1-[p(k+1)-p(k)]}{c} \end{aligned}$$

Hence,

$$b(k) + 1 - [s(k+1) - s(k)]c \leq b(k) + p(k+1) - p(k) \tag{6.1}$$

Since $p(k+1) \geq p(k)$, we have

$$0 \leq b(k) + p(k+1) - p(k) \tag{6.2}$$

Combine (6.1) and (6.2) into (2.5) , we have

$$b(k+1) \leq b(k) + p(k+1) - p(k) \tag{6.3}$$

Summing up (6.3) for $j' \leq k < j$, we have:

$$\sum_{k=j'}^{j-1} b(k+1) - b(k) \leq \sum_{k=j'}^{j-1} p(k+1) - p(k)$$

That is

$$b(j) - b(j') \leq p(j) - p(j')$$

The theorem is proved.

6.3.4 Theorem 2.1.2.4

$$d + \frac{b(j)}{c} \geq \frac{p(j)}{c}$$

The equality holds if, and only if, $\forall k$ that satisfies $j - p(j) + 1 \leq k \leq j : a(k) - a(k-1) = \frac{1}{c}$.

Proof:

$$\begin{aligned} a(j) &= s(j) + d + \frac{b(j)}{c} \\ &= a(j - p(j)) + d + \frac{b(j)}{c} \end{aligned}$$

The first step is by (2.6) and the second step is by (2.4).

Hence,

$$\begin{aligned} \frac{b(j)}{c} + d &= a(j) - a(j - p(j)) \\ &= \sum_{k=j-p(j)+1}^{j} [a(k) - a(k-1)] \\ &\geq \sum_{k=j-p(j)+1}^{j} \frac{1}{c} \\ &= \frac{p(j)}{c} \end{aligned}$$

The third step is by (2.13) of Theorem 2.1.2.2.

Hence, the inequality holds.

The equality in the third step holds if, and only if, $\forall k$ that satisfies $j - p(j) + 1 \leq k \leq j : a(k) - a(k-1) = \frac{1}{c}$.

6.3.5 Theorem 2.1.3.2

The system is in stable-link state upon the arrival of packet $j \Longleftrightarrow p(j) = cd + b(j)$

Proof:

This is a the special case of Theorem 2.1.2.4 when $\forall k$ that satisfies $j - p(j) < k \leq j : a(k) - a(k-1) = \frac{1}{c}$.

6.3.6 Theorem 2.1.3.3

If the system is in stable link state upon the arrival of packet j and $p(j+1) \geq cd$, then the system is in stable link state upon the arrival of packet $j+1$.

Proof:

Since the system is in *stable-link* state upon the arrival of packet j, we have

$$\forall k \text{ that satisfies } j - p(j) < k \leq j : a(k) - a(k-1) = \frac{1}{c} \tag{6.4}$$

by Definition 2.1.3.1.

By (2.2), $p(j+1) \leq p(j) + 1$. That is $j + 1 - p(j+1) \geq j - p(j)$.

For any k that satisfies $j + 1 - p(j+1) < k < j + 1$, k satisfies $j - p(j) < k \leq j$. By (6.4), $a(k) - a(k-1) = \frac{1}{c}$.

For packet $j+1$, since $p(j+1) \geq cd$, we have $p(j+1) \geq p(j) - b(j)$ by Theorem 2.1.3.2. Hence,

$$\begin{aligned} s(j+1) - s(j) &= a(j+1-p(j+1)) - a(j-p(j)) \\ &= \sum_{k=j-p(j)+1}^{j+1-p(j+1)} a(k) - a(k-1) \\ &\leq \sum_{k=j-p(j)+1}^{j+1-cd} a(k) - a(k-1) \\ &= \sum_{k=j-p(j)+1}^{j+1-cd} \frac{1}{c} \\ &= \frac{p(j) - cd + 1}{c} \\ &= \frac{b(j)+1}{c} \end{aligned}$$

The first step is from (2.4); the third step is from Thereom 2.1.2.2 and the fact that $p(j+1) \geq cd$; the forth step is from the definition of *stable-link* state and the fact that $j - p(j) < j - p(j) + 1 \leq j + 1 - cd \leq j$; the sixth step is from Thereom 2.1.3.2.

Since $s(j+1) - s(j) \leq \frac{b(j)+1}{c}$, Thereom 2.1.2.2 shows $a(j+1) - a(j) = \frac{1}{c}$.

Hence, $\forall k : j+1-p(j+1) < k \leq j+1$, we have $a(k) - a(k-1) = \frac{1}{c}$. The system is in stable state upon the arrival of packet $j+1$.

The theorem is proved.

6.3.7 Theorem 2.1.3.4

If $\forall k : j - p(j) < k \leq j : p(k) > cd$; the system enters stable-link state upon the arrival of j.

Proof:

By Theorem 2.1.2.4, $p(k) > cd \Rightarrow b(k) > 0$.

Hence, $b(k-1) + 1 - [s(k) - s(k-1)]\, c > 0$.

By Theorem 2.1.2.2, $a(k) - a(k-1) = \frac{1}{c}$.

We have $\forall k : j - p(j) < k \leq j : a(k) - a(k-1) = \frac{1}{c}$. By Definition 2.1.3, the system is in stable-link state upon the arrival of packet j.

6.3.8 Theorem 2.1.3.5

If $\forall k : j - p(j) < k \leq j : p(k-1) \geq p(k)$ and $p(j) \leq cd$, the system has $b(j) = 0$.

Proof:

Assume $b(j) > 0$.

Since $p(k-1) \geq p(k)$, we have $s(k) - s(k-1) \geq \frac{1}{c}$ and $b(k) \leq b(k-1)$.

Hence, $b(k) \geq b(j) > 0$ for $\forall k : j - p(j) < k \leq j$.

That is: $a(k) - a(k-1) = \frac{1}{c}$.

By Definition 2.1.3, the system is in stable-link state upon the arrival of packet j.

By Theorem 2.1.3.2, we have $p(j) = cd + b(j) > cd$.

This contradicts the condition that $cd \geq p(j)$.

Hence, the assumption cannot be true.

6.3.9 Corollary 2.1.4

$$\tau_{k+1} \geq \tau_k + w(s(\tau_{k+1}))$$

Proof:

According to (2.19), we have

$$\begin{aligned}
\tau_{k+1} - \tau_k &= w(s(\tau_k)) + \max\{\Delta w(\tau_k), 0\} \\
&= \max\{w(s(\tau_k)) + \Delta w(\tau_k), w(s(\tau_k))\} \\
&= \max\{w(s(\tau_{k+1})), w(s(\tau_k))\}
\end{aligned}$$

The third equation is based on (2.18).

Hence, $\tau_{k+1} \geq \tau_k + w(s(\tau_{k+1}))$.

6.3.10 Theorem 2.1.4.1:

$$a(\tau_k) \leq s(\tau_{k+1}) < a(\tau_{k+1})$$

Proof:

$s(\tau_{k+1}) < a(\tau_{k+1})$ comes directly from the ack-clocking model (2.6) as $d > 0$.

We prove $a(\tau_k) \leq s(\tau_{k+1})$ by contradiction.

Assume $a(\tau_k) > s(\tau_{k+1})$.

We have:

$$\begin{aligned} a(\tau_k) &> s(\tau_{k+1}) \\ &= a(\tau_{k+1} - p(\tau_{k+1})) \end{aligned}$$

The equality comes from (2.4).

According to Corollary 2.1.2.2,

$$\tau_k > \tau_{k+1} - p(\tau_{k+1})$$

That is:

$$\begin{aligned} \tau_{k+1} - \tau_k &< p(\tau_{k+1}) \\ &\leq w(s(\tau_{k+1})) \end{aligned}$$

The second inequality comes from Theorem 2.1.2.1.

According to Corollary (2.1.4), $\tau_{k+1} - \tau_k \geq w(s(\tau_{k+1}))$.

We reach a contradiction.

Hence, $a(\tau_k) \leq s(\tau_{k+1})$.

The theorem is proved.

6.3.11 Theorem 2.1.4.2:

$$\forall \tau_k : w(s(\tau_k)) = p(\tau_k)$$

Proof:

Let the last packet that is sent before $a(\tau_{k-1})$ to be j'. That is:

$$j' = \max\{j : s(j) < a(\tau_{k-1})\}$$

We have $w(s(j')) = p(j')$ Since $j' \geq j, \forall j : s(j) = s(j')$.

Because $s(\tau_{k-1}) < a(\tau_{k-1}) \leq s(\tau_k)$, we have $\tau_{k-1} \leq j' < \tau_k$. Hence, $w(s(j')) = w(s(\tau_{k-1}))$.

We prove the theorem with two cases.

When $\Delta w(\tau_{k-1}) \leq 0$:

For all j such that $j' \leq j \leq \tau_k - 1$, we have $w(s(j)) \geq w(s(j+1))$ because $a(\tau_{k-2}) \leq s(\tau_{k-1}) \leq j \leq s(\tau_k) < a(\tau_k)$.

Since $p(j') = w(s(j'))$, we have $p(j) = w(s(j))$ for all $j : j' < j \leq \tau_k$, according to Theorem 2.1.2.1.

Hence, $p(\tau_k) = w(s(\tau_k))$.

When $\Delta w(\tau_{k-1}) > 0$:

We show that packet $j'+1$ is sent at the time of $a(\tau_{k-1})$. By definition of j', we have $s(j'+1) \geq a(\tau_{k-1})$. Hence,

$$\begin{aligned} w(s(j'+1)) &= w(a(\tau_{k-1})) \\ &= w(s(\tau_{k-1})) + \Delta w(\tau_{k-1}) \\ &= w(s(j')) + \Delta w(\tau_{k-1}) \end{aligned}$$

We have $p(j'+1) = \max_{0 \leq k \leq p(j')}\{p(j') - k + 1 | p(j') - k + 1 \leq w(a(j' + 1 - p(j') + k))\}$.

Let $k = \tau_{k-1} - j' + p(j')$, we have:

$$k \leq p(j')$$

because

$$\tau_{k-1} \leq j'$$

and

$$k \geq 0$$

because

$$\begin{aligned} a\left(\tau_{k-1}\right) > s\left(j'\right) &\Rightarrow \tau_{k-1} > j' - p\left(j'\right) \\ &\Rightarrow \tau_{k-1} - j' + p\left(j'\right) > 0 \end{aligned}$$

We also have

$$\begin{aligned} p\left(j'\right) - \left[\tau_{k-1} - j' + p\left(j'\right)\right] + 1 &\leq j' - \tau_{k-1} + 1 \\ &< p\left(j'\right) \\ &= w\left(s\left(j'\right)\right) \\ &= w\left(s\left(\tau_{k-1}\right)\right) \\ &< w\left(a\left(\tau_{k-1}\right)\right) \\ &= w\left(a\left(j' - p\left(j'\right) + \tau_{k-1} - j' + p\left(j'\right)\right)\right) \end{aligned}$$

Hence,

$$\begin{aligned} p\left(j'+1\right) &\geq p\left(j'\right) - \left[\tau_{k-1} - j' + p\left(j'\right)\right] + 1 \\ &= j' - \tau_{k-1} + 1 \end{aligned}$$

and

$$\begin{aligned} s\left(j'+1\right) &= a\left(j' + 1 - p\left(j'+1\right)\right) \\ &\leq a\left(j' + 1 - j' + \tau_{k-1} - 1\right) \\ &= a\left(\tau_{k-1}\right) \end{aligned}$$

Hence, $s(j'+1) \leq a(\tau_{k-1})$. But definition of j', we have $s(j'+1) \geq a(\tau_{k-1})$. Hence, $s(j'+1) = a(\tau_{k-1})$.

Because there is a packet $j'+1$ sent at the time $a(\tau_{k-1})$, according to Theorem 2.1.2.1, there exists a packet j^* in which $s(j^*) = s(j'+1)$ and $p(j^*) = w(s(j^*)) = w(a(\tau_{k-1}))$. Hence, $s(j^*) = a(j^* - p(j^*)) = a(\tau_{k-1})$.

That is

$$\begin{aligned} j^* &= \tau_{k-1} + p(j^*) \\ &= \tau_{k-1} + w(s(j^*)) \\ &= \tau_{k-1} + w(a(\tau_{k-1})) \\ &= \tau_k \end{aligned}$$

Hence, $\tau_k = j^*$ and $p(\tau_k) = w(\tau_k)$.

6.3.12 Theorem 2.1.4.3:

$b(\tau_k) \leq \Delta w(\tau_k)$ or the system is in stable-link state upon the arrival of τ_k.

Proof:

Let $j' = \min\{j : s(j) \geq a(\tau_{k-1})\}$ to be the first packet that is sent after the arrival of last decision packet.

By definition of j', we have $s(j'-1) < a(\tau_{k-1}) \leq s(j')$.

Hence, $p(j') \leq p(j'-1)$ (Otherwise, $s(j'-1) = s(j')$.)

Since $s(j') \geq a(\tau_{k-1})$, $a(j' - p(j')) \geq a(\tau_{k-1})$ and $j' \geq \tau_{k-1} + p(j')$.

According to (2.20), for all j such that $\tau_{k-1} \leq j' - p(j') \leq j < j'$, $w(j) = w(s(\tau_{k-1}))$.

Since $p(\tau_{k-1}) = w(s(\tau_{k-1}))$, according to Theorem (2.1.2.1), we have $p(j) = w(s(j))$ for all j such that $\tau_{k-1} \leq j < j'$.

Hence, for all j such that $j' - p(j') \leq j < j'$, we have

$$p(j) = w(s(\tau_{k-1}))$$

and because $p(j') \leq p(j'-1)$, we have: $\forall j : j' - p(j') < j \leq j', p(j) \leq p(j-1)$.

If $p(j') > cd$, we have $p(j) > cd$ for all $j : j' - p(j') < j \leq j'$. In this case, the system is in link stable state upon the arrival of packet j'.

If $p(j') \leq cd$, we have $b(j') = 0$ according to Theorem 2.1.3.5.

Now we divide the problem into three situations.

Case 1: $\Delta w(\tau_{k-1}) \leq 0$.

$p(j') = w(s(j')) = w(a(\tau_{k-1}))$ according to Theorem 2.1.2.1.

Hence, the $p(j)$ sequence is non-increasing from j' to τ_k, as the sequence from $j' - p(j')$ to j'. Hence, we have $b(\tau_k) = 0$ or the system is in link stable state upon the arrival of packet τ_k.

Case 2: $\Delta w(\tau_{k-1}) > 0$ and $p(j') \leq cd$.

By $p(j') \leq cd$, we have $b(j') = 0$.

By $\Delta w(\tau_{k-1}) > 0$, we have $p(j') = p(j'-1) = w(s(\tau_{k-1}))$

And since $\Delta w(\tau_{k-1}) > 0$, the $p(j)$ sequence is non-decreasing from j' to τ_k. According to Theorem 2.1.2.3, we have

$$\begin{aligned} b(\tau_k) &\leq b(j') + p(\tau_k) - p(j') \\ &\leq w(s(\tau_k)) - w(s(\tau_{k-1})) \\ &= \Delta w(\tau_{k-1}) \end{aligned}$$

Hence, $b(\tau_k) \leq \Delta w(\tau_{k-1})$.

Case 3: $\Delta w(\tau_{k-1}) > 0$ and $p(j') > cd$.

By $\Delta w(\tau_{k-1}) > 0$, we have $p(j)$ sequence is non-decreasing from j' to τ_k. Hence, $p(\tau_k) \geq p(j') > cd$.

Hence, the $p(j) > cd$ for all j such that $j' - p(j') < j \leq \tau_k$. Hence, the system is in link stable state upon the arrival of packet τ_k according to Theorem 2.1.3.4.

6.3.13 Theorem 2.1.5

Given the ack-clocking model described in (2.3)(2.4)(2.5)(2.6) and the TCP Vegas congestion control algorithm described in (2.17)(2.23)(2.18)(2.19)(2.20), a single TCP flow converges to equilibirum regardless of capacity c, propagation delay d and initial state.

If $\alpha d > 1$, given any initial state, we have

$$\exists J : \forall j > J : cd + \alpha d - 1 < w\left(s\left(j\right)\right) < cd + \alpha d + 1 \text{ and } \alpha d - 1 < b\left(j\right) < \alpha d + 1$$

Proof:

First, we prove that $\exists K_1 : \forall k > K_1 : w\left(a\left(\tau_k\right)\right) > cd + \alpha d - 1$.

Given any intial state $w\left(s\left(\tau_k\right)\right)$, if $w\left(s\left(\tau_k\right)\right) \leq cd + \alpha d - 1$, we have: $p\left(\tau_k\right) = w\left(s\left(\tau_k\right)\right) \leq cd + \alpha d - 1$.

According to Theorem 2.1.4.3, we have that either $b\left(\tau_k\right) \leq \Delta\left(\tau_{k-1}\right) \leq 1$ or the system is in link stable state upon the arrival of τ_k.

For the first case, since $b\left(\tau_k\right) \leq 1$, we have $p\left(\tau_k\right) \leq cd + b\left(\tau_k\right)$ by Theorem 2.1.2.4. We have

$$\begin{aligned}
\frac{p\left(\tau_k\right)}{d} - \frac{p\left(\tau_k\right)}{D\left(\tau_k\right)} &= \frac{p\left(\tau_k\right)}{d} - \frac{p\left(\tau_k\right)}{d + \frac{b\left(\tau_k\right)}{c}} \\
&= \frac{b\left(\tau_k\right) p\left(\tau_k\right)}{d\left(cd + b\left(\tau_k\right)\right)} \\
&\leq \frac{b\left(\tau_k\right)\left(cd + b\left(\tau_k\right)\right)}{d\left(cd + b\left(\tau_k\right)\right)} \\
&\leq \frac{b\left(\tau_k\right)}{d} \\
&\leq \frac{1}{d} \\
&< \frac{1}{\alpha}
\end{aligned}$$

In this case, $\Delta w\left(\tau_k\right) = 1$.

For the second case, since the system is in link stable state, we have $p\left(\tau_k\right) =$

$cd + b(\tau_k)$. We have

$$\begin{aligned}
\frac{p(\tau_k)}{d} - \frac{p(\tau_k)}{D(\tau_k)} &= \frac{p(\tau_k)}{d} - \frac{p(\tau_k)}{d + \frac{b(\tau_k)}{c}} \\
&= \frac{b(\tau_k)[cd + b(\tau_k)]}{d(cd + b(\tau_k))} \\
&= \frac{b(\tau_k)}{d} \\
&= \frac{p(\tau_k) - cd}{d} \\
&\leq \frac{cd + \alpha d - 1 - cd}{d} \\
&< \frac{1}{\alpha}
\end{aligned}$$

Hence, $\Delta w(\tau_k) = 1$ as long as $w(s(\tau_k)) \leq cd + \alpha d - 1$. Since $cd + \alpha d - 1$ is finite, there exists an K_1 which satisfies $w(s(\tau_{K_1})) > cd + \alpha d - 1$.

Second we prove that $\forall k \geq K_1$, $w(s(\tau_k)) > cd + \alpha d - 1$.

Assume the window size will be smaller than or equal to $cd + \alpha d - 1$ for some $k \geq K_1$. Let the smallest of such k to be k'. That is:$w(s(\tau_{k'})) \leq cd + \alpha d - 1$ and $\forall k : K_1 \leq k < k' : w(s(\tau_k)) > cd + \alpha d - 1$.

By this assumption, we have: $\Delta w(\tau_{k'-1}) = -1$ and $w(s(\tau_{k'-1})) \leq cd + \alpha d$.

But since $cd + \alpha d - 1 < w(s(\tau_{k'-1})) \leq cd + \alpha d$, we have: $p(\tau_{k'-1}) = cd + b(\tau_{k'-1})$ and

$$\begin{aligned}
\frac{p(\tau_{k'-1})}{d} - \frac{p(\tau_{k'-1})}{D(\tau_{k'-1})} &= \frac{p(\tau_{k'-1})}{d} - \frac{p(\tau_{k'-1})}{d + \frac{b(\tau_{k'-1})}{c}} \\
&= \frac{b(\tau_{k'-1})\, p(\tau_{k'-1})}{d(cd + b(\tau_{k'-1}))} \\
&= \frac{b(\tau_{k'-1})}{d} \\
&= \frac{p(\tau_{k'-1}) - cd}{d} \\
&\leq \frac{cd + \alpha d - cd}{d} \\
&\leq \frac{1}{\alpha}
\end{aligned}$$

Hence, $\Delta w(\tau_{k'-1}) \geq 0$. This contradicts our assumption that $\Delta w(\tau_{k'-1}) = -1$. Hence, k' does not exist.

Second, we can prove that $\exists K_2 > K_1 : \forall k > K_2 : w(a(\tau_k)) < cd + \alpha d + 1$. The proof is very similar to the first step and is ignored here.[2]

Finally, let $K = \max\{K_1, K_2\}$, we have $\forall k > K : cd + \alpha d - 1 < w(a(\tau_k)) < cd + \alpha d + 1$.

Let $J = \tau_K$, according to Theorem 2.20, we have

$$\forall j > J : cd + \alpha d - 1 < w(s(j)) < cd + \alpha d + 1$$

[2]In fact, it is easier because the link-stable state is always satisfied for all $k : k > K_1$

Bibliography

[1] Jon Postel, "Rfc 793 - transmission control protocol," Sep 1981.

[2] Phil Karn and Craig Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM Transactions on Computer Systems*, vol. 9, no. 4, pp. 364–373, 1991.

[3] R. Jain, "A timeout-based congestion control scheme for window flow-controlled networks," *IEEE J. Selected Areas in Commun.*, vol. 4, no. 7, Oct 1986.

[4] V. Jacobson, "Congestion Avoidance and Control," *ACM SIGCOMM '88*, pp. 314–329, Aug. 1988.

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

[6] S. Floyd and T. Henderson, "RFC 2582: The New Reno Modification to TCP's Fast Recovery Algorithm," April 1999.

[7] Matthew Mathis and Jamshid Mahdavi, "Forward acknowledgement: refining TCP congestion control," in *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*. 1996, pp. 281–291, ACM Press.

[8] S. Floyd, "Highspeed tcp for large congestion windows," 2002.

[9] T. Kelly, "Scalable TCP: Improving Performance in HighSpeed Wide Area Networks," 2003.

[10] Lisong Xu, Khaled Harfoush, and Injong Rhee, "Binary Increase Congestion Control for Fast Long-Distance Networks," in *INFOCOM*, 2004.

[11] Douglas Leith and Robert N. Shorten, "H-TCP: TCP for high-speed and long-distance networks," in *Proceedings of PFLDnet 2004*, 2004.

[12] Carlo Caini and Rosario Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *INTERNATIONAL JOURNAL OF SATELLITE COMMUNICATIONS AND NETWORKING*, vol. 22, pp. 547–566, 2004.

[13] R. Jain, "A delay based approach for congestion avoidance in interconnected heterogeneous computer networks," *Computer Communications Review, ACM SIGCOMM*, pp. 56–71, 1989.

[14] Z. Wang and J. Crowcroft, "A new congestion control scheme: Slow start and search (tri-S)," *ACM Computer Communication Review, SIGCOMM*, vol. 21, no. 1, pp. 32–43, 1991.

[15] Z. Wang and J. Crowcroft, "Eliminating periodic packet losses in the 4.3-Tahoe BSD TCP congestion control algorithm," *ACM Computer Communications Review*, April 1992.

[16] Lawrence S. Brakmo and Larry L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, 1995.

[17] David X Wei, Cheng Jin, Steven H Low, and Sanjay Hedge, "FAST TCP: Motivation, Architecture, Algorithms, Performance," *IEEE/ACM Transactions on Networking, to appear*, 2007.

[18] Mark Allman and Ethan Blanton, "Notes on burst mitigation for transport protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 53–60, 2005.

[19] Lixia Zhang, Scott Shenker, and David D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," in *Proceed-*

ings of the ACM SIGCOMM 1991 Conference on CommunicationsArchitectures and Protocols, 1991, pp. 133–147.

[20] J. Kulik, R. Coutler, D. Rockwell, and C. Partridge, "A simulation study of paced TCP," Tech. Rep. BBN Technical Memorandum No. 1218, BBN Technologies, 1999.

[21] D. Hong, "FTCP Fluid Congestion Control," 2000.

[22] David Wei, Sanjay Hedge, and Steven Low, "A burstiness control for TCP," in *Proceedings of PFLDNet 2005*, 2005.

[23] Hao Jiang and Constantinos Dovrolis, "Why is the internet traffic bursty in short time scales?," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 2005, pp. 241–252, ACM Press.

[24] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi, "The macroscopic behavior of the TCP congestion avoidance algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 3, pp. 67–82, 1997.

[25] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose, "Modeling TCP throughput: a simple model and its empirical validation," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*. 1998, pp. 303–314, ACM Press.

[26] Vishal Misra, Wei-Bo Gong, and Don Towsley, "Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red," in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 2000, pp. 151–160, ACM Press.

[27] C. V. Hollot, Vishal Misra, Donald F. Towsley, and Weibo Gong, "A control theoretic analysis of RED," in *INFOCOM*, 2001, pp. 1510–1519.

[28] Steven H. Low, Fernando Paganini, Jiantao Wang, Sachin Adlakha, and John C. Doyle, "Dynamics of tcp/red and a scalable control," in *Proceedings of IEEE Infocom*, March 2002.

[29] Hyojeong Choe and Steven Low, "Stabilized Vegas," in *Proceedings of IEEE Infocom*, March 2003.

[30] Shao Liu, Tamer Basar, and R. Srikant, "Pitfalls in the fluid modeling of rtt variations in window-based congestion control," in *Proceedings of IEEE INFOCOM, Miami, FL, March 2005*, 2005.

[31] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[32] D. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks*, vol. 17, pp. 1–14, 1989.

[33] T. Kelly, "Scalable TCP: Improving Performance in HighSpeed Wide Area Networks," 2003.

[34] Yee-Ting Li, Douglas Leith, and Robert N. Shorten, "Experimental Evaluation of TCP Protocols for High-Speed Networks," http://hamilton.ie/net/eval/HI2005.htm.

[35] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of TCP pacing," in *Proceedings on INFOCOM 2000*, 2000, pp. 1157–1165.

[36] R.L. Cruz, "A calculus for network delay. I. Network elements in isolation," *IEEE Transactions on Information Theory*, vol. 37, pp. 114–131, Jan 1991.

[37] David X. Wei, "Congestion control algorithms for high speed long distance tcp connections," Tech. Rep., California Institute of Technology, Jun 2004.

[38] Injong Rhee and Lisong Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," in *Proceedings of PFLDNet 2005*, 2005.

[39] F. Baccelli and D. Hong, "AIMD, fairness and fractal scaling of TCP traffic," in *Proceedings on IEEE Infocom 2002*, 2002.

[40] Doug J Leith and R. Shorten, "Impact of Drop Synchronisation on TCP Fairness in High Bandwidth-Delay Product Networks," in *PFLDNet*, 2006.

[41] Vern Paxson, "End-to-end Internet packet dynamics," in *Proceedings of the ACM SIGCOMM '97 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Cannes, France, September 1997, vol. 27,4 of *Computer Communication Review*, pp. 139–154, ACM Press.

[42] M. Borella, D. Swider, S. Uludag, and G. Brewster, "Internet Packet Loss: Measurement and Implications for End-to-End QoS," 1998.

[43] "The Network Simulator - NS-2," URL: http://www.isi.edu/nsnam/ns/index.html.

[44] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[45] "PlanetLab: An open platform for developing, deploying, and accessing planetary-scale services," URL:http://www.planet-lab.org.

[46] Sally Floyd, Mark Allman, Amit Jain, and Pasi Sarolahti, "Internet draft: Quick-start for tcp and ip," Oct 2006.

[47] Nandita Dukkipati and Nick McKeown, "Why flow-completion time is the right metric for congestion control," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 59–62, 2006.

[48] Y. Yang and S. Lam, "General aimd congestion control," 2000.

[49] Kartikeya Chandrayana, Sthanunathan Ramakrishnan, Biplab K. Sikdar, and Shivkumar Kalyanaraman, "On randomizing the sending times in tcp and other

window based algorithms.," *Computer Networks*, vol. 50, no. 3, pp. 422–447, 2006.

[50] David X. Wei, Pei Cao, and Steven H. Low, "Fairness Convergence of Loss-based TCP," URL: http://www.cs.caltech.edu/~weixl/pacing/sync.pdf.

[51] Robert Shorten, Fabian Wirth, and Douglas Leith, "A positive systems model of TCP-like congestion control: asymptotic results," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, pp. 616–629, 2006.

[52] Cheng Jin, David X. Wei, and Steven H. Low, "TCP FAST: motivation, architecture, algorithms, performance," in *Infocom 2004*, Mar 2004.

[53] R. Shorten, D. Leith, J. Foy, and R. Kilduff, "Analysis and design of congestion control in synchronised communication networks," in *Proceedings on 12th Yale Workshop on Adaptive and Learning Systems*, may 2003, http://ww.hamilton.ie/doug_leith.htm.

[54] "GridFTP," URL: http://www.globus.org/toolkit/docs/4.0/data/gridftp/.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003, pp. 29–43, ACM Press.

[56] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018: TCP Selective Acknowledgement Options," Oct 1996.

[57] S. Jansen and A. McGregor, "Simulation with Real World Network Stacks," in *Proceedings of Winter Simulation Conference*, Dec 2005, pp. 2454– 2463.

[58] "Speeding up NS-2 scheduler," URL: http://www.cs.caltech.edu/~weixl/ns2.html.

[59] A. Tang, J. Wang, and S. Low, "Counter-intuitive behaviors in networks under end-to-end control," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 2, 2006.

[60] A. Tang, J. Wang, S. Low, , and M. Chiang, "Network equilibrium of heterogeneous congestion control protocols," in *Proceedings of IEEE Infocom*, March 2005.

[61] "Linux Kernel Documents: TCP protocol," linux-2.6.16.13/Documentation/networking/tcp.txt.

[62] Stephen Hemminger, "Network Emulation with NetEm," in *Proceedings of Linux Conference AU*, April 2005.

[63] P. Sarolahti and A. Kuznetsov, "Congestion Control in Linux TCP," *USENIX Annual Technical Conference*, pp. 49–62, 2002.

[64] Matt Mathis, Jeff Semke, J. Mahdavi, and Kevin Lahey, "Rate Halving Algorithm for TCP Congestion Control," Jun 1999.

[65] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP," Jul 2000.

[66] "A mini-tutorial for NS-2 TCP-Linux," URL: http://www.cs.caltech.edu/~weixl/ns2.html.

[67] "A Linux TCP implementation for NS-2," URL: http://www.cs.caltech.edu/~weixl/ns2.html.

[68] Randy Brown, "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[69] JongSuk Ahn and SeungHyun Oh, "Dynamic Calendar Queue," *Thirty-Second*, vol. 00, pp. 20, 1999.

[70] Kah Leong Tan and Li-Jin Thng, "SNOOPy Calendar Queue," in *Proceedings of the 32nd Winter Simulation Conference, Orlando, Florida*, 2000, pp. 487–495.

[71] M. Allman, "RFC 3465 - TCP Congestion Control with Appropriate Byte Counting (ABC)," Feb 2003.

[72] S. Floyd, M. Handley, and J. Padhye, "A comparison of equation-based and aimd congestion control," 2000.

[73] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, December 2004.