Scalable Increase Adaptive Decrease: Congestion Control supporting Low Latency and High Speed

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Mirja Kühlewind

geb. in Berlin

Hauptberichter:	Prof. Dr.-Ing. Andreas Kirstädter
Mitberichter:	Prof. Dr.-Ing. Jörg Ott (Technische Universität München)
Tag der Einreichung:	8. Dezember 2014
Tag der mündlichen Prüfung:	5. Oktober 2015

Institut für Kommunikationsnetze und Rechnersysteme
der Universität Stuttgart

2016

Abstract

Congestion control in the Internet has been an open research issue for more than two decades. A large number of proposals already exist that especially address the scalability problem of traditional congestion control in high-speed networks. However, more and more applications with narrow latency requirements are emerging which are not well addressed by existing proposals. In this work, we present TCP SIAD, a new congestion control scheme supporting both high-speed networks and low latency, based on a new design principle called Scalable Increase Adaptive Decrease (SIAD). More precisely, our algorithm aims to provide high utilization under various network conditions, and therefore allows operators to configure small buffers for low latency support. We designed TCP SIAD based on a new approach that aims for a fixed feedback rate independent of the available bandwidth and provides full scalability. Further, our approach introduces a configuration knob for the induced feedback rate and thereby controls the aggressiveness. This can be used by a higher-layer control loop to impact capacity sharing, e.g., for applications that need a minimum rate. Increasing the aggressiveness can lead to higher throughput but also induces more congestion experienced by the transmission as well as competing traffic. However, we argue that fairness cannot be addressed by congestion control, that only influences the instantaneous share for each flow, and therefore do not aim for TCP-friendliness. Instead, fairness must be policed on a per-user basis over longer time scales supported by a configurable aggressiveness of the used congestion control scheme as provided by TCP SIAD.

We evaluate TCP SIAD's scalability, adaptivity, capacity sharing, and convergence properties against well-known high-speed congestion control schemes, such as Scalable TCP and High Speed TCP, as well as H-TCP that among other goals targets small buffers. We show that only TCP SIAD is able to utilize the bottleneck with arbitrary buffer sizes while avoiding to unnecessarily buffer packets in network queues. Further, only TCP SIAD and Scalable TCP implement a fixed feedback rate independent of the link speed, where Scalable TCP reaches this on the cost of inducing a standing queue and high loss rate. Moreover, we demonstrate the capacity sharing properties of SIAD depending on the configured feedback rate. Further, due to a new Fast Increase phase TCP SIAD can quickly allocate newly available capacity and converges reasonably fast. In addition, TCP SIAD provides a much higher resilience to non-congestion losses than all other schemes in test. We conclude that TCP SIAD fulfills the stated requirements and shows high robustness to perform further testing in the Internet.

Kurzfassung

Die Entwicklung von Verfahren zur Überlastkontrolle (*Congestion Control*) im Internet ist ein aktives Forschungsfeld seit über 20 Jahren. Viele der vorgeschlagenen Verfahren adressieren dabei im Speziellen Netze mit hohen Übertragungsgeschwindigkeiten, da traditionelle Verfahren mit einem immer weiter steigenden Verzögerungs-Bandbreiten-Produkt nicht skalieren. Heutzutage gibt es jedoch auch immer mehr Anwendungen, die nicht (nur) hohe Anforderungen an die verfügbare Bandbreite stellen, sondern vor allem auch eine geringe Ende-zu-Ende Verzögerung benötigen. In dieser Arbeit wird ein neues Congestion-Control-Verfahren vorgestellt, das beide Anforderungen adressiert: Skalierbarkeit in Netzen mit hohen Übertragungsgeschwindigkeiten (*high speed*) und Unterstützung von Diensten mit speziellen Anforderungen an die Ende-zu-Ende Verzögerung (*low latency*). Das Verfahren basiert auf dem Prinzip *Scalable Increase Adaptive Decrease* und heißt daher TCP SIAD. Dieser Algorithmus erreicht hohe Linkauslastungen in verschiedensten Netzszenarien; auch im Falle von sehr kleinen Paket-Puffern vor dem Übertragungsengpass (*Adaptive Decrease*). Dies erlaubt dem Netzbetreiber, durch die Konfiguration kleiner Puffergrößen, die Ende-zu-Ende Verzögerung zu verringern, um somit z.B. interaktive Dienste besser zu unterstützen. TCP SIAD basiert zudem auf einem neuen Ansatz, der, unabhängig von der aktuell verfügbaren Bandbreite, die Feedback-Rate des Congestion Control Verfahrens konstant halt (*Scalable Increase*). Somit skaliert TCP SIAD mit der verfügbaren Bandbreite eines Links, während traditionelle Verfahren bei höherer Bandbreite weniger häufig Feedback bekommen und somit nur sehr langsam auf Veränderungen reagieren können. Gleichzeitig erlaubt dieser Ansatz einen Konfigurationsparameter einzuführen, der die Aggressivität des Congestion Control Verfahrens steuert. Dieser Parameter kann von einer Anwendung verwendet werden, um die Bandbreitenaufteilung zwischen konkurrierenden Flüssen zu beeinflussen, und somit, falls zwingend notwendig für diesen Dienst, auch (kurzzeitig) einen höheren Anteil der verfügbaren Ressourcen zu bekommen. TCP SIAD ist absichtlich nicht auf *TCP-Friendliness* ausgelegt, da Fairness nicht instantan und auf Fluss-Ebene durchgesetzt werden sollte, sondern über eine längere Zeitspanne auf Nutzer-Ebene. Dies kann jedoch ein Congestion Control Verfahren, das die momentane Senderate für nur einen Fluss steuert, nicht leisten, sondern muss vom Netzbetreiber auf andere Art und Weise umsetzt werden, z.B. durch Limitierungen pro Nutzer am Netzrand.

In dieser Arbeit wird die Skalierbarkeit, Adaptierbarkeit, erreichte Bandbreitenaufteilung und mögliche Konvergenz (*scalability, adaptivity, capacity sharing, and convergence*) von TCP SIAD bewertet und jeweils mit bekannten *high speed*-Verfahren vergleichen - wie Salable TCP, High Speed TCP und H-TCP, wobei H-TCP das einzige Verfahren ist, das für Szenarien mit kleinen Puffern am Engpass im Netz entwickelt worden ist. Es wird gezeigt, dass nur TCP SIAD in der Lage ist, einen Link unter verschiedensten Puffer-Konfigurationen auszulasten und gleichzeitig zu vermeiden, dass bei großen Puffern dauerhaft Pakete in der Warteschlange sind, so dass keine unnötig große Ende-zu-Ende Verzögerung entsteht (*standing queue*). H-TCP erreicht dies Ziel nicht, da die Größe des Reduktionsfaktors (zur Reduzierung der Senderate bei Überlast) bei H-TCP beschränkt ist. TCP SIAD und Scalable TCP implementieren beide eine feste Feedback-Rate, die unabhängig von der verfügbaren Bandbreite ist. Jedoch implementiert Scalable TCP eine feste, sehr hohe Rate, die in der Regel den Puffer am Engpass im Netz permanent voll hält und zudem eine hohe Verlustrate impliziert. Zusätzlich wird in dieser Arbeit aufgezeigt, dass TCP SIAD in der Lage ist, durch eine konfigurierbare Aggressivität (der Senderatenerhöhung) verschiedene Bandbreitenaufteilungen zu erreichen; mit konkurrierenden Flüssen, die entweder den gleichen oder sogar einen anderen Algorithmus verwenden. Durch die Einführung der *Fast Increase* Phase kann TCP SIAD neu-verfügbare (auch sehr hohe) Bandbreiten schnell belegen und konvergiert zügig. Zudem ist TCP SIAD sehr robust bei hohen (nicht durch Überlast verursachten) Verlusten. Im Gegensatz zu allen anderen Verfahren, die untersucht wurden, kann TCP SIAD einen Link in einer solchen Situation weiterhin sehr hoch auslasten. Zusammenfassend erreicht TCP SIAD als einziges Verfahren alle gestellten Anforderungen und zeigt eine sehr hohe Robustheit.

Contents

List of Figures

List of Tables

List of Acronyms

ABC	Appropriate Byte Counting	22
ACK	Acknowledgment	13
AIMD	Additive Increase Multiplicative Decrease	1
API	Application Programming Interface	51
AQM	Active Queue Management	14
BDP	Bandwidth-Delay-Product	1
CBR	Constant Bit Rate	54
cwnd	Congestion Window	20
CoDel	Controlled Delay	44
DCTCP	Data Center TCP	25
ECN	Explicit Congestion Notification	15
EWMA	Exponential Weighted Moving Average	43
FIFO	First In First Out	43
IAT	Inter Arrival Time	51
IETF	Internet Engineering Task Force	42
IP	Internet Protocol	42
IW	Initial Window	21
MIMD	Multiplicative Increase Multiplicative Decrease	8
MPLS	Multi-Protocol Label Switching	42
OWD	One-Way-Delay	27
PIE	Proportional Integral Controller Enhanced	44
PRR	Proportional Rate Reduction	23
RED	Random Early Detection	30
rmcat	RTP Media Congestion Avoidance Techniques	143
RTO	Retransmission Time-Out	14
RTT	Round-Trip Time	3
RTTM	Round-Trip Time Measurement	14

SACK Selective Acknowledgment 13

SEQ Sequence Number 13

TCP Transmission Control Protocol 1

TFRC TCP-Friendly Rate Control 16

TSOpt TCP Timestamp Option 14

UDP User Datagram Protocol 27

QoE Quality of Experience 4

QoS Quality of Service 17

VM Virtual Machine 50

1 Introduction

Congestion control has been an important part of the Internet since the late 1980s. It was introduced after a series of congestion collapses. During these events the communication network settles into an overloaded state where goodput degrades more and more. Congestion control provides a control loop which decreases the sending rate in presence of congestion. As a large amount of the Internet traffic (in terms of volume) uses the Transmission Control Protocol (TCP) where congestion control is inherently implemented, today's congestion control does a good job in avoiding a congestion collapse.

But in general an efficient congestion control scheme should

1. avoid a congestion collapse, as well as
2. be able to utilize the bottleneck link independent of the network configuration, and
3. support different application layer requirements.

While congestion avoidance is essentially provided by the control loop that reduces its sending rate based on some kind of congestion feedback, in the last 20-25 years various congestion control schemes have been proposed to either cope with certain link characteristics or specific application requirements. Even though a large variety of proposals for congestion control exists, which to some extent are even implemented, most often only the default configuration of the operating system is used. E.g. in 2011 about 16-25% of webservers still used the traditional Additive Increase Multiplicative Decrease (AIMD), up to 19% Compound TCP, the default algorithm in Windows, and more than 44% TCP Cubic or BIC, the default in Linux [149]. Many of the proposed schemes do not see wide deployment because it is not straight forward when to use which scheme. Unfortunately application designers often do not know the network conditions under which there application is used later on and system administrators do not know the application requirements.

Therefore, we aim to design a new congestion control scheme, named TCP Scalable Increase Adaptive Decrease (SIAD), that addresses both of these general requirements listed above by

(a) utilizing the bottleneck link independent of the configured buffer size to better support low latency services, as well as scaling with the Bandwidth-Delay-Product (BDP) of the network path used by introducing a Fast Increase phase when network conditions are changing, and

(b) providing a configuration interface to the application layer that determines the aggressiveness of the congestion control scheme and thereby the instantaneous share of the bottleneck capacity of the bottleneck link depending on its current requirements.

Further, to be able to compete with existing loss-based congestion control schemes in the Internet, our algorithm design must react on loss-based congestion feedback.

In the following section we describe three arising challenges to the current Internet that provide the motivation for the particular design of our proposed novel congestion control scheme, which can be summarized by

Scalability in high BDP networks
A large set of congestion control schemes have been proposed to scale better with high BDP networks. Even though most of them scale better than traditional congestion control approaches, they still have a dependency on the BDP and therefore do not fully solve the scalability problem.

Support for low latency services in the Internet
More and more applications with narrow latency requirements are emerging which are not addressed well by existing proposals. While end-to-end latency is influenced by various factors, congestion control should aim to minimize queuing delay as far as possible while maintaining high utilization.

Per-user fairness based on congestion policing
We relax the requirement to be TCP-friendly following the argumentation that fairness does not depend on the instantaneous rate but on the usage over time and therefore should be enforced on a per-user (and not per-flow) basis [25]. As congestion control determines the instantaneous sending rate of single flows, fairness must be addressed by a different mechanism, e.g. by congestion-based policing.

Based on this analysis we derive more detailed general requirements for congestion control. Subsequently, we list concrete problems and limitations of current congestion control proposals. Finally, we specify the design goals of TCP SIAD to be used to evaluate our proposal against other existing proposals with respect to these goals.

1.1 Motivation based on Current Research Challenges

Scalability is a known problem of traditional congestion control schemes and in fact is already addressed by a large set of congestion control proposals targeting high-speed networks. In contrast we additionally aim to provide better support for applications and services that require low latency as well as to design a congestion control scheme that is suitable to assist deployment of per-user congestion policing in the future Internet.

1.1.1 Scalability

The classical AIMD [67, 34]-based TCP congestion control does not scale well with high BDP paths, as their congestion feedback rate often depends on the BDP of the network path used. AIMD-based schemes increase their sending rate stepwise until the network is overloaded (for a short period of time) to probe for available capacity. If congestion is subsequently detected the AIMD scheme decreases its sending rate and starts probing again. AIMD schemes consequently induce self-congestion with a certain rate in steady state, when the network conditions do not change. The self-induced congestion provides the needed feedback as input for the congestion control loop to either increase or decrease the sending rate. This feedback rate usually decreases indirectly proportional to an increase in the congestion window and therefore the BDP. This means the larger the path bandwidth or end-to-end delay, the slower the congestion control reacts to changing network conditions as congestion feedback is received less often. This implicates two problems:

1. Congestion control potentially takes a long time to utilize newly available bandwidth when, e.g., a competing flow stops sending. As an example, to raise the sending rate from 5 Gbit/s to 10 Gbit/s with an Round-Trip Time (RTT) of 100 ms and 1500 byte packets, TCP NewReno [10] takes more than an hour, as further explained in Section 1.3.1. During this period the network resources are not utilized efficiently and a transmission consequently takes longer than needed.
2. For high bandwidth or high delay links, such a congestion control scheme is rate-limited by the theoretical limits of the network bit error rate. The throughput dependent on the loss probability p and the RTT can be given for TCP NewReno, as derived by [106, 113], by
$$B(p) = \frac{1}{RTT}\sqrt{\frac{3}{2p}}.$$
This equation is also explained in more detail in Section 2.2.1. This means when a flow fully utilizes a certain link with capacity B, it induces periodic congestion events with a rate p. But when the bit error rate on the link is larger than this loss probability, respectively, the throughput is limited and the link cannot be fully utilized.

There are many congestion control proposals for high-speed or large-delay networks targeting this problem. These usually scale much better than the traditional TCP NewReno but they often still depend on the BDP of the bottleneck link. We aim to design a congestion control scheme where the feedback rate is independent of the network path characteristics. Moreover, congestion control should be able to quickly adapt to changing network conditions. Therefore, the congestion control scheme must be able to distinguish between the stable operation state and a situation where the network or traffic conditions are changing, in order to quickly grab newly available capacity.

1.1.2 Low Latency Support

Today's Internet is mainly optimized for high throughput and low loss rates, in order to utilize network resources most efficiently. This strategy implies large enough network buffers to (a) absorb small data bursts and (b) provide sufficient space for TCP congestion control to work efficiently. Unfortunately, loss-based congestion control always fills these buffers in order to induce loss as feedback signal. Therefore, large buffers provide a relevant source of additional latency on the end-to-end path. In the worst case, with full synchronization of flows or multiplexing of a small number of flows, current loss-based congestion control algorithms require a buffer space of one BDP to be able to utilize the bottleneck link [14]. Note that different flows have different end-to-end delays and therefore the buffer needs to be dimensioned to the largest expected delay to guarantee full utilization. In fact, this network configuration is optimized for the transmission of large amounts of data where only the completion time is relevant for the user's Quality of Experience (QoE), but not for latency-sensitive services.

Today, more and more applications are emerging with narrow latency requirements or hard completion deadlines like real-time video conferencing or interactive cloud services. Typically, these kinds of services implement a lot of additional logic in the application layer to overcome current networking performance limitations with respect to latency. The experienced latency adds up different components, where delay induced by the network is one large component which is completely out of control of the application itself. Even if the congestion control algorithm of the host that operates the latency-sensitive application aims to keep the network buffers on the transmission path shallow, competing loss-based cross traffic fills these buffers and thereby induces high queuing delay for all flows competing on this bottleneck link.

While end-to-end congestion control cannot change the buffer configuration, it should at least be designed to

(a) avoid a standing queue by emptying large buffers at congestion event and

(b) keep the link utilization high even if very small buffers are configured.

This allows network operators to configure small buffers, thus avoid large queuing delay and thereby better support low latency services while still maintaining link utilization high.

1.1.3 Per-User Congestion Policing

Many congestion control proposals are quite complex because they are designed to be "TCP-friendly" [47], at least under certain network conditions as, e.g., low speed. TCP-friendliness means that a flow competing with a classical AIMD-based congestion control schemes such as NewReno [10] achieves equal rate sharing under the same network conditions. This requirement might prohibit the usage of this scheme for certain services, e.g., streaming that needs a certain minimum sending rate to work, no matter how many competing flows currently use the bottleneck link. But note, as congestion control in TCP usually is a sender-only algorithm, it can easily be changed and therefore it is basically largely deployed on a cooperative but voluntary basis. In fact, congestion control schemes implementing different levels of aggressiveness are already in use today [149]. This means two flows competing on the same bottleneck link do not share the capacity equally but depending on their aggressiveness. Under game-theoretical consideration assuming uncooperative and selfish users which only aim to maximize their own utility, network-based regulation is needed to avoid "The Tragedy of the

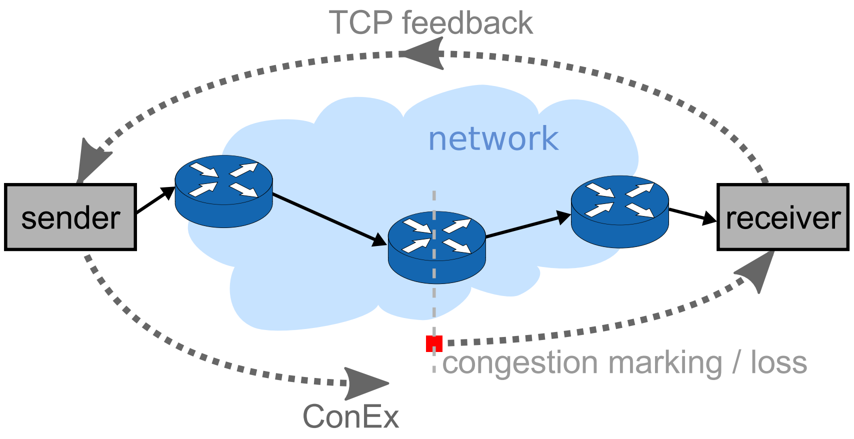


Figure 1.1: Re-Feedback Principle of the Congestion Exposure (ConEx) signaling protocol.

Commons” [60]. Therefore, it has been proposed to use congestion as measure for the price of link usage [100, 133, 37, 53, 82].

In this case the network operator needs to implement a protection mechanism to avoid service degradation for all users imposed by single misbehaving users in case of congestion. Today this is mostly done by greatly over-provisioning in the core network and limitation of the access network capacity and thereby largely avoiding congestion at points where different users are competing. With increasing access network capabilities, e.g., due to fiber-optic technology, this might not be sufficient anymore.

Note that service degradation can only occur if a certain network link is overloaded and therefore congestion in form of loss or delay occurs which affects all parties that use the link. In a congestion situation the available resources need to be shared in a fair manner. Thereby fairness should be regarded at a per-user (and not per-flow) basis [25]. This means fairness depends not only on the instantaneous rate of one flow (as with TCP-friendliness) but also on each user’s usage over time. Of course this kind of fairness definition also allows grabbing a larger share than others but only for a limited time.

Such a definition of fairness cannot be addressed solely by the congestion control. Further, resource allocation is influenced by economic and policy implications [20] and therefore should be managed by the network operator (at network ingress). However, only if resources are sparse and therefore congestion occurs, policing is needed. Congestion policing based on the Congestion Exposure (ConEx) protocol [28] has been proposed to implement per-user fairness. The principle of ConEx-based policing is displayed in Figure 1.1. A congestion-controlled flow sends data over a given network path to the receiver. Subsequently, a TCP receiver provides feedback about the congestion level as input for congestion control to the sender and the sender’s congestion control algorithm reduces its sending rate in the presence of congestion. Now the ConEx protocol additionally signals this congestion level back into the network. Based on this information a network operator can police its users by, e.g., providing the same congestion allowance to all users and starting to drop packets once this allowance is exhausted.

When policing is done based on the amount of congestion that the flows of one user cause, the sender needs a way to control the congestion rate and thereby the aggressiveness of the congestion control scheme. We leave the implementation of congestion policing to others.

But to make congestion policing feasible, congestion control should provide an upper layer configuration interface to influence the amount of congestion caused by one transmission and thereby the share of capacity over time.

1.2 Congestion Control Requirements

General requirements for the design of Internet congestion control derived from the research challenges explained above can be summarized as the following (already published by the author of this thesis in [86]): These requirements differ from what can usually be found in the literature which can be summarized as efficiency, fairness, convergence, and robustness (see Section 2.5.3). We especially do not target fairness as this should not be addressed on an instantaneous per-flow basis by congestion control but instead on a longer time scale and per user. Further, we do not only require efficiency but more specifically scalability with as well as adaptivity to changing network situations.

Scalability Congestion control should be able to quickly adjust to new network conditions even in high-speed networks, by quickly utilizing available bandwidth as well as quickly yielding capacity for new flows. To achieve this, a congestion control scheme needs to get frequent network feedback to be able to detect changing network conditions early. The feedback rate is controlled by the probing frequency of the congestion control scheme. Unfortunately, most current congestion control schemes to not correctly scale their probing frequency with the bandwidth. While it is desirable to always have the same probing frequency, with today's congestion control feedback is usually received less often when the bandwidth increases.

Adaptivity Congestion control should adapt its decrease and increase behavior to the network conditions to be able to efficiently utilize every link independently of the provided buffer size. Therefore, congestion control should adapt its decrease factor to the buffer size of the bottleneck link queue such that there are always enough packets to keep the link full. Moreover, the buffer should be emptied after every decrease completely as any additional packets in the queue only increase the end-to-end delay by building a standing queue and therefore decrease performance. If a decrease was too strong, e.g., due to an incorrect buffer size estimation, the increase behavior of a congestion control scheme could partly compensate this underutilization by increasing more aggressively afterwards.

Capacity Sharing Congestion control schemes must be able to share the available bandwidth with different congestion control schemes, at least as long as both schemes react to the same congestion feedback signal(s). As loss is the most dominant congestion signal today, congestion control should always be able to compete with other flows that rely only on loss-based feedback. The congestion control schemes used today are constrained by the aim to achieve TCP-friendliness. This is only possible if all competing schemes implement about the same increase and decrease behavior. But in fact equal per-flow sharing is not desirable [25]. What is needed is to guarantee that every flow can at least grab some of the capacity. One approach to control the share of the network capacity, that a transmission allocates, could be realized by providing a configuration interface for the higher-layer to dynamically change the aggressiveness dependent on the application requirements. Moreover, this provides a simple way to implement a relative prioritization among a single user's flows. Ideally, one finds one solution that fits all higher-layer service requirements providing a simple interface.

Convergence Congestion control should quickly converge to a stable state with or without competing flows regardless of the congestion control scheme used. The congestion control start-up phase is designed to start carefully but then quickly achieve the maximum sending rate. If the bottleneck capacity is already fully utilized by other flows, this start-up phase might end early and it can take a long time to converge. Ideally, a starting flow finds an empty queue which can be filled to achieve a certain sending rate before imposing a congestion signal to all competing flows and then starting the convergence phase. Even though an empty queue is preferable, loss-based congestion control needs to fill the buffers frequently to get a congestion feedback signal from the network. This means the buffer is overloaded over and over for short periods of time. Therefore, congestion control should at least make sure that the buffer frequently runs empty to give starting flows the chance to grab capacity quickly. Moreover, congestion control should not unnecessarily overload the network and avoid large overshoots [1]

1.3 Limitations of Current Congestion Control

This section explains in detail limitations of traditional and partly currently used, mostly AIMD-based congestion control. With AIMD the sending rate is increased linearly with a certain increase step per RTT until congestion occurs, as explained further in Section 2.2. When a congestion notification is received, the sending rate is decreased multiplicatively by a certain factor. In the traditional congestion control such as TCP NewReno, both the increase step per RTT as well as the decrease factor are fixed values. A fixed increase rate leads to scalability problems in high-speed networks as it makes the congestion feedback rate dependent on the currently available bandwidth. The fixed decrease factor most often leads to either a standing queue or link underutilization depending on the network buffer configuration and cross traffic. Both problems are explained in detail next.

A large set of congestion control proposals exist where the increase rate or decrease factor is adapted dynamically. Most of these proposals, however, only care about a more adaptive increase behavior, while the decrease behavior, which is more important for low latency support and link utilization, is at best only partly addressed. The attempts of various congestion control proposals are discussed in detail in Section 2.2.2.

1.3.1 Congestion Feedback Rate depends on Available Bandwidth

Due to the scalability problem of AIMD-based TCP congestion control, the congestion feedback rate usually depends on network conditions such as the total link capacity of the bottleneck and

[1] While in fact one congestion feedback signal is sufficient as input for the control loop to trigger a decrease in sending rate, an aggressive congestion control scheme, that can quickly allocate new capacity, might induce several losses in the phase where the link is overloaded due to probing. This effect is called overshoot.

the number and aggressiveness of cross traffic flows. For AIMD schemes the congestion period T [83] is calculated by

$$T = \frac{(1-\beta)W_{max}}{\alpha}$$

where α is the number of increases in packets per RTT and β the multiplicative decrease factor when congestion occurs. It can be seen that T depends on W_{max}, the maximum congestion window at the current share of capacity. In contrast for Multiplicative Increase Multiplicative Decrease (MIMD) schemes T [83] can be calculated by

$$T = \frac{-log(\beta)}{log(1+\alpha)}$$

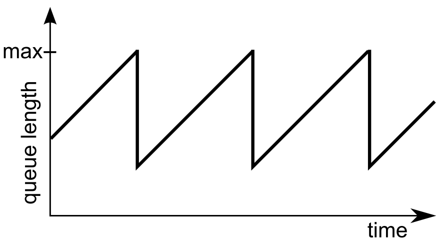
and therefore is independent of the network conditions. Unfortunately, MIMD-based schemes do not converge without any influence of randomness, e.g., introduced due to processing delays in other layers or dynamics of cross traffic, as further explained in Section 2.2.

This is not only a problem for scalability as capacity increases (mainly on access links), but also for the deployment of congestion policing. Assume one end host always uses the same congestion control algorithm with the same aggressiveness, e.g. TCP NewReno, and always sends over the same bottleneck link with the same RTT. As long as this one end host sends only one flow and is alone on the bottleneck link, it induces congestion with a certain rate. In case of TCP NewReno the sending rate increases by one packet per RTT. Therefore, exactly one packet is lost whenever a congestion events occurs and the capacity is exceeded for one RTT until the congestion feedback is received at the sender.

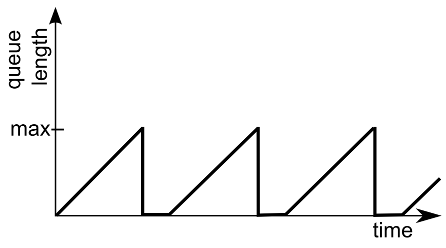
Now assume a second flow competes for the bandwidth on the same bottleneck link using the same congestion control with the same RTT. In stable state both flows share the available capacity equally. In case of TCP NewReno in total at every synchronized congestion event two packets are lost, as both flows together have a common increase rate of two packets per RTT. In most cases each flow sees one of the losses and reduces its sending rate respectively. Thereby the total loss rate is already doubled, as there is now more congestion, but still a single flow sees only one loss per congestion event, as they still increase their sending rate with the same aggressiveness than before of in this case one packet per RTT. Unfortunately, as each flow now only gets half of the bottleneck capacity, the congestion frequency is doubled as well. Therefore, in the end the total congestion rate is four times as large as before and the congestion rate as recognized by each single flow has doubled. This is especially a problem for congestion policing as a policer must be configured to allow for a certain congestion rate per user.

1.3.2 Bottleneck Link Utilization depends on Buffer Size

Congestion control schemes with a fixed decrease factor need a certain amount of buffering in the network to be able to fully utilize a link. E.g. TCP Reno halves its sending rate in case of congestion and therefore needs one BDP of buffering to have enough packets in the queue to balance out the decreases sending rate (until the RTT is halved as well as soon as the queue is empty). This is because loss-based congestion control completely fills the buffer (or at least fill to a certain level, where the queue starts notifying for congestion) and therefore introduces



(a) Standing queue.



(b) Link underultilization.

Figure 1.2: Queue sizing.

queuing delay to each packet traversing the queue. Note that a given link has a certain end-to-end delay, but per-packet RTT varies based on the queuing delay. In TCP congestion control the sending rate is often maintained as the number of packets that are allowed to be in flight during one RTT, the so-called congestion window. Therefore, the same number of packets can in fact lead to different sending rates depending on the current RTT. When a congestion notification is received, the sender reduces its sending rate and therefore the queue starts shrinking. In the case of TCP NewReno the congestion window is halved. At this point of time the queue is full (in the worst case) and with one additional BDP of buffering the RTT doubles when the congestion signal occurs. Now as the congestion window is halved and actually half of the packets in flight sit in the queue (one additional BDP), the queue is emptied within the next RTT. At the same time the RTT itself reduces to half the maximum value and thereby half the number of packets in flight is still able to fully utilize the link. This means a certain minimum amount of buffer in the network is needed to overcome this phase where the sending rate is low depending on how much the sender decreases its sending rate.

However, if the network buffer is too large, a standing queue develops. This means that even though the congestion control scheme reduces its sending rate, the buffer never becomes completely empty. This permanently introduces an additional delay that degrades performance without helping utilization. This case is sketched in Figure 1.2a. In contrast if the buffer is too small, the buffer will be completely emptied after a window reduction which causes a period where no packets are available to fill the link, as shown in Figure 1.2b, and therefore leads to link underutilization.

Therefore, optimal buffer sizing depends on the decrease behavior of the respective congestion control scheme, which is usually unknown by the network operator in the Internet. While synchronized flows with a decrease factor of 0.5 need one $BDP = RTT * C$ of buffer on a link with capacity C, it has been shown that a buffer of

$$B = \frac{RTT \cdot C}{\sqrt{n}}$$

is sufficient for highly aggregated links with n flows due to multiplexing and consequently desynchronization [14]. Still as the number of flows varies, buffers usually are over-dimensioned to guarantee full link utilization in any case. This causes a standing queue and might cause performance degradation for latency-sensitive applications. In fact the network operator is not able to configure the queue 'perfectly' as neither the current number of flows nor the congestion control scheme used by each single flow are easily known. In line with the *end-to-end arguments* in system design [126] it is more desirable to adapt the decrease behavior of the congestion control algorithm to buffer size of the network router in front of the bottleneck link. This allows network operators to simply configure smaller queues without worrying about underutilization and consequently better support low latency services.

1.4 Design Goals

Based on the analysis of current research challenges as well as shortcomings of current congestion control proposals, we state the following explicit design goals. We use these design goals to evaluate the general suitability of our proposal to address the presented research challenges as well as in comparison with existing congestion control proposals. While existing proposals are able to address single goals equally well or better than our proposal, none of the current proposal was designed to address all goals listed below.

High Link Utilization The proposed congestion control scheme should always be able to utilize the bottleneck link independent of network buffer size, especially if the buffers are configured small to better support low-latency requirements. Therefore, the proposed scheme needs to estimate the current buffer size dynamically and adapt its (decrease) behavior such that the available capacity can always be used efficiently.

Minimized Average Queuing Delay The proposed congestion control scheme should drain the queue at every decrease to avoid a standing queue and consequently minimize the average end-to-end delay. When the buffer size cannot be estimated correctly, the scheme could potentially further decrease to probe when the queue runs empty. As additional delay caused by a standing queue does not increase link utilization, the available resources are most efficiently used when queuing delay is minimized.

Fixed Feedback Rate The proposed congestion control scheme should induce a fixed feedback rate when self-congested. To reach this goal while the decrease rate is calculated dynamically, the increase rate needs to be calculated dynamically as well. Only if the feedback rate is independent of the available bandwidth, the congestion control is fully scalable.

Quick Capacity Allocation The proposed congestion control scheme should be able to quickly allocate newly available bandwidth if the network conditions have changed when, e.g., a competing flow stops sending. Especially in high-speed networks, the scheme needs to distinguish between stable and changing network conditions in order to increase the sending rate differently in each case and provide better scalability and fast convergence in these scenarios.

Configurable Aggressiveness The proposed congestion control scheme should further provide a configuration interface to influence the capacity sharing when competing with other traffic.

1.5 Outline

In this thesis we present and evaluate Scalable Increase Adaptive Decrease (SIAD), a new TCP congestion control scheme. TCP SIAD aims to support both, high speed and low latency to address requirements of today's and future communication networks as well as of new emerging latency-sensitive service. More precisely, our algorithm provides high utilization under various networking conditions by adapting the decrease behavior based on an estimate of each flow's share of packets in the bottleneck queue. This allows operators to configure small buffers for low latency support while maintaining high link utilization. Moreover, we designed the increase behavior of TCP SIAD based on a completely novel approach called *Scalable Increase* that aims for a fixed feedback rate independent of the available bandwidth and is therefore fully scalable for high-speed networks. Based on this approach, we moreover introduce an additional configuration parameter that can be used by a higher-layer control loop to influence each flow's capacity share at the cost of higher congestion. This interface can be used for applications such as streaming that need a minimum sending rate to provide any useful service. This interface supports the deployment of congestion policing in the future Internet.

Further, we define a Fast Increase phase which is entered when changing network situations are detected. In Fast Increase our proposed congestion control scheme increases more quickly than in steady state behavior. We assume in Fast Increase that new capacity is available and therefore that the new capacity limit is unknown. A similar consideration was the basis for TCP Cubic's increase curve, which is the default algorithm in Linux. In this work we introduce Fast Increase as a general concept that is beneficial to implement for any future congestion control scheme addressing high-speed networks. Note that Fast Increase as implemented in TCP SIAD allocates new available capacity much faster than TCP Cubic, as we will show in our evaluation.

In the next chapter we provide the necessary background on congestion control in TCP, evaluation of TCP protocol mechanisms, as well as the implementation of congestion control in Linux. Additionally, we discuss the state of the art in end-to-end binary-feedback best-effort congestion control as used in TCP and show that none of the current proposals to address all of our design goals. Based on a detailed description of selected proposals, we further point out similarities and differences to our design approach. Chapter 3 details the design of our TCP SIAD algorithm, lays out reasoning for the design decisions leading to it, and presents an implementation thereof in the Linux kernel. Chapter 4 presents a full evaluation of TCP SIAD, based on this implementation, including a comparison to other state of the art congestion control schemes. We show that TCP SIAD can achieve our design goals on high link utilization, low average queuing delay as well as a fixed feedback rate in a single flow scenario, in contrast to all other schemes we evaluate. Further, we demonstrate the feasibility to influence capacity sharing using TCP SIAD's configuration parameter. Moreover, we additionally show that it is also possible to achieve equal sharing with traditional congestion control schemes, if the respective network conditions are known. Finally, we demonstrate the robustness of TCP SIAD in various Internet scenarios, e.g. in a scenario with multiple bottlenecks or short flows of cross traffic. We show that TCP SIAD is very robust to high loss/congestion rates induced by either bursty cross traffic or lower layer bit error than all other scheme we evaluate. As a drawback TCP SIAD might induce higher oscillations itself due to Fast Increase. Note, there is a known trade-off between higher responsiveness and therefore faster convergence at the cost of worse smoothness and therefore larger oscillations of the sending rate. We can conclude that TCP

SIAD fulfills the stated requirements and shows high robustness to perform further testing in the Internet.

2 Internet Congestion Control in TCP

Network congestion occurs when the aggregated traffic demand is higher than the capacity of a network resource and therefore the network is overloaded. In packet-switched networks as the Internet, each network node maintains a buffer to, first of all, handle effects of statistical multiplexing and, moreover, to be able to compensate short periods of overload. When the incoming traffic load exceeds the capacity of the out-going link, packets are stored in the buffer. This, of course, induces additional delay in the end-to-end transmission. When the load increases further, the buffer overflows and packets get dropped. Both, queuing delay and packet loss, decrease the network performance and are known effects of network congestion.

If a reliable transport is used, lost data is retransmitted. Unfortunately, in case of losses due to network overload, retransmissions might even cause an increase in network load. This behavior can bring a network into a state where the network load is high but the throughput gets lower and lower until no useful communication is possible anymore. This state is known as congestion collapse. Due to a series of congestion collapses in the Internet in the late 1980s, congestion control was introduced. The main task of congestion control is to determine the sending rate of an end-to-end transmission and thereby avoid a congestion collapse. Additionally, congestion control should allow efficient utilization of the available capacity of a given end-to-end network path. Further, goals are often stated as, e.g., fast convergence when competing with intra-protocol traffic, minimizing the amplitude of oscillations, high responsiveness when new resources become available, and a fair coexistence with inter-protocol cross traffic [101]. Goals and metric for performance evaluation are more explicitly explained in Section 2.5.3.

Since this series of congestion collapses, congestion control is implemented as an essential part of the Transmission Control Protocol (TCP) [119]. TCP is a reliable transport protocol that is widely used in the Internet. Reliability is implemented by sending an Acknowledgment (ACK) on the reception of a data segment and retransmitting (potentially) lost data when no new acknowledgment is received. For this purpose a data segment can be identified by a Sequence Number (SEQ). Each new payload byte increases the SEQ by one. The ACK contains, moreover, an acknowledgment number that announces the next in-order SEQ expected by the receiver. To reduce signaling overhead, a TCP receiver might not acknowledge each received segment separately but multiple at once. A TCP receiver should at least acknowledge every second packet and delay an acknowledgement not more that 500 ms [10, 23]. Duplicated ACKs are triggered at the receiver by the arrival of out-of-order data segments and thereby do not acknowledge new data but repeat the previous acknowledgment number. When the missing data is received, a cumulative acknowledgment is sent that acknowledges all (now) in-order segments received. Additionally, TCP often implements Selective Acknowledgment (SACK) [104], where, in case of duplicated ACKs, the received sequence number ranges are announced to the sender.

Therefore, when more than one packet was lost, the sender does not have to wait until an accumulated ACK announces the next hole in the sequence number space, but can retransmit lost packets immediately.

When a loss is detected, TCP congestion control [10] becomes active and usually reduces the sending rate. The sending rate is most often determined by a (sliding-) window. Based on the *packet conservation principle* [67], new packets are only sent into the network when an old packets have exited. This leads to TCP implementations that are mostly self-clocked and take actions only when an ACK is received. When no ACKs are received at all for a certain time larger than at least one RTT, the Retransmission Time-Out (RTO) is triggered and the sending rate is reduced to a minimum value. The current sending window is the minimum of the receive window and the congestion window where the receive window is announced by the receiver to not overload the receiver buffer. As long as flow control does not become active and not signal a smaller receive window to not overload the receiver, the sending window equals the congestion window. The congestion window is estimated by the congestion control algorithm based on implicit or explicit network feedback. Inherently, the congestion control mechanism of a TCP sender assumes that loss occurs to due network congestion and therefore reacts each time congestion is notified. Note, congestion control usually at most reduces once per RTT as the congestion feedback has a signaling delay of one RTT. Unfortunately, congestion is not the only reason for the occurrence of loss. Packets can be dropped for various reasons by different nodes on the network path, e.g., due to bit errors.

To determine, e.g., a large enough value for the RTO, the RTT needs to be measured by the TCP sender. The Round-Trip Time Measurement (RTTM) mechanism in TCP either needs to store the sent-out time stamp and SEQ of all or a sample of packets or can use the TCP Timestamp Option (TSOpt) [68]. With TSOpt the sender adds the current time stamp to each packet and the receiver reflects this time stamp in the respective ACK. By subtracting the reflected time stamp from the current system time the RTT can be measured for each received ACK holding a valid TSOpt. Note that in case of delayed or duplicated ACKs, the time stamp of the first unacknowledged packet is reflected which might inflate the RTT measurement artificially but guarantees a large enough RTO.

In the next section we first give an overview of classification principles for congestion control. Based on this overview we narrow down our scope to mechanisms and approaches that are relevant for our proposal. In Section 2.2 we introduce congestion control algorithms with a focus on end-to-end binary-feedback congestion control for best-effort services implemented in TCP. We discuss general principles as well as survey a set of specific loss-based, delay-based, and hybrid proposals. Thereby proposals that explicitly address high-speed networks or requirements of latency-sensitive services are of our main interest.

[125] distinguishes two approaches to handle congestion, *congestion control* including recovery as a reactive approach when a network overload had happened and *congestion avoidance* to proactively handle in advance. Thereby congestion avoidance mechanisms for packet scheduling and buffer management in the network (network algorithms) provide feedback to the congestion control of the sending end host to perform respective rate adjustments (source algorithms) [92]. Both algorithms together provide a control loop for network congestion. Therefore, in Section 2.3 we also discuss network-supported congestion avoidance introducing basic mechanism of Active Queue Management (AQM). Moreover, approaches based on service dif-

ferentiation for low latency support are presented. These approaches could be used in addition to our congestion control proposal to quickly deploy smaller buffers and therefore low latency support in the Internet. Finally, Section 2.4 and 2.5 provide background knowledge on TCP congestion control implementation in Linux as kernel module and the used network simulation tool as well as scenarios and metrics for TCP performance evaluation.

2.1 Classification of Congestion Control Schemes

Congestion control can be classified based on very different criteria. For TCP congestion control we only focus on unicast approaches in contrast to multicast congestion control, as classified by [147, 56]. Multicast proposals face very different problems and therefore are not considered here [22, 54]. We discuss classification based four criteria which often can be found in literature:

- the type of feedback which the rate adjustment is based on,
- the way the send-out behavior and therefore the sending rate is determined,
- deployability, and
- based on which optimization criteria a certain scheme is designed.

Network feedback

Most often congestion control schemes are categorized by the type of feedback that is used for congestion notification. E.g. mechanisms can be separated into *delay-based* and *loss-based*. Delay measurements provide an early congestion feedback as the queue fills when the link is fully utilized and thereby additional delay is induced before loss occurs. Delay-based approaches often try to limit the queuing delay or, respectively, the number of packets in the queue, and thereby inherently induce only low queuing delay. Unfortunately, those approaches are not able to co-exist with loss-based mechanisms, as the delay signal triggers a reduction in sending rate before loss occurs and there this congestion signal is received earlier than the feedback for loss-based approaches [61].

Of course, hybrid approaches exist which use both signals. For most of these approaches, loss is the predominant congestion signal which is used to decide when the sending rate should be reduced while delay measurements influence by how much it is reduced. Alternatively, with a focus on delay-sensitive applications, proposals exist that are loss-based if loss-based cross traffic is detected and delay-based otherwise to maintain low queuing delay [29, 94, 107].

Both signals, loss and delay, are *implicit* congestion signals that occur inevitably when a buffer overflow in an overloaded network happens. In contrast Explicit Congestion Notification (ECN) allows a congested network node to *explicitly* signal congestion before a buffer overflows [120]. Loss and ECN are *binary* feedback signals that only announce if congestion occurred or not. An estimation of the queuing delay can provide more fine-grained information based on the resolution of the measurement approach. Note that loss, ECN feedback as well as delay measurements do not provide any information on the link state when the bottleneck link is not fully utilized yet.

While implicit signals as well as ECN in TCP can be used as input signal for pure *end-to-end* congestion control, there is also a set of proposals which are *network-supported*. Network-supported approaches (such as XCP [78] and RCP [39]), in contrast, often provide a more detailed feedback directly from the congested network element such as the available capacity, the number and rate of competing flows, or a fair-share target rate.

Send-out Behavior

As described above, the sending rate in TCP is foremost *window-based*. This means only when some space was previously freed in the sending window, a new packet can be sent out. Therefore, all actions taken by the congestion control mechanism are clocked by the arrival of ACKs that indicate the receipt of a packet at the receiver and signals additionally if there is congestion or not. In *rate-based* approaches, in contrast, the sending rate is controlled directly and the sent-out time of each packet is triggered by timers.

Furthermore, [56] distinguishes between *probe-based* and *model-based* approaches. A probe-based approach adjusts its sending rate directly based on the feedback provided by the receiver, while otherwise a model of the TCP behavior is used to determine the sending rate. An example for such a scheme is TCP-Friendly Rate Control (TFRC), where the *equation-based model* of [113] is used to calculate the sending rate based on the average experienced loss rate. Alternatively, *economic models* could be used as proposed by *Kelly et al.* [80, 53].

Deployability

Congestion control adjusts the sending rate which is inherently determined by the sender. Thus, in principle, congestion control could be deployed by *sender-side* only mechanisms. This is only possible if some kind of network feedback is provided either by the congested network node directly or through the receiver. TCP inherently provides feedback on packet loss and delay due to the way reliability is implemented. Therefore, most often TCP congestion control schemes can be deployed by sender-side changes only. Sometimes it is beneficial to take the control decision on the *receiver-side*, as the receiver might have more information on, e.g., the network conditions of its access link. In this case some kind of signaling is needed to constrain the sending rate of the sender. In TCP this could be done by exploiting the receive window announcement which is supposed to be used for flow control.

If a congestion control scheme requires any kind of *network support*, at least the bottleneck network node needs to support the respective signaling. In the Internet any node could be the bottleneck which makes deployment of such mechanisms hard. If it cannot be guaranteed that the bottleneck node supports the respective mechanism, there must at least be a fallback end-to-end scheme. If the bottleneck is even the element that makes the control decision, all senders that cross the bottleneck link must comply which is even harder to ensure.

Optimization Criteria

Various congestion control schemes have been proposed to improve performance aspects of specific network or traffic characteristics. Many proposals are targeted at *high-speed/high-BDP* networks. Another group of proposals focuses on *lossy links* with potentially *variable bit rates* such as in some wireless networks or on *high delay* links, e.g., in satellite communication. Regarding the traffic characteristics, TCP congestion control schemes are foremost with long-lived flows, as the congestion control algorithm mostly influences the steady state behavior, and therefore are often also optimized for these kind of traffic scenarios. However, there are also schemes that are especially designed to cope with Internet traffic that consists of a mix of long (*elephants*), short (*mice*) and application-limited flows [58].

Further, many congestion control schemes target *best-effort* services where the traffic is *elastic* and therefore the goal is to adapt the sending rate on the network conditions while always utilizing all available capacity. Services that use this kind of congestion control are often insensitive to delay variations. In contrast, there are also services with *real-time* requirements like, e.g., Voice over IP. These service might send (bursts of) *Constant Bit Rate* (CBR) cross traffic [132] instead. Further, while best-effort services using TCP usually try to grab as much capacity as currently possible, there are also scavenger protocols for background transfers that only try to grab capacity when no foreground traffic is present.

Moreover, many proposed congestion control schemes are designed to provide fair capacity sharing with flows using the same algorithm (*intra-protocol fairness*) as well as using other (usually TCP NewReno-like) schemes (*inter-protocol fairness*). *TCP-friendliness* [47] thereby means that a flow competing with a classical TCP congestion control would achieve an equal capacity sharing; at least under certain network conditions as, e.g., low speed and the same end-to-end delay. Fairness is often defined by the *Jain's fairness index* [71] which is maximized when all competing flows (at the same bottleneck) have the same rate (see Section 2.5.3). *Min-max fairness* [21] is achieved if the rate of a flow cannot be increased without decreasing the rate of an equal or smaller rate flow and therefore it additionally takes the existence of rate-limited flows into account such that it utilizes all available capacity and thereby potentially allows a higher share of the capacity for non-rate-limited flows. *Proportional fair* [79] scheduling, as often used in mobile network, aims to maximize the total throughput and thereby to optimize spectral efficiency while maintaining a certain minimum share for each user. While the mentioned fairness criteria are focused on rate sharing, fairness could also be defined based on other Quality of Service (QoS) metrics such as latency.

Further, congestion control can be regarded as an optimization problem proposing a game-theoretic approach to reach the Nash equilibrium where each player maximizes its capacity share while minimizing the cost in form of congestion [77, 99].

According to [125, 7] the main research challenges for Internet congestion control can be summarized as *"interoperability, robustness, stability, convergence, implementation complexity and fairness"* [125] as well as fairness issues regarding non-TCP-friendly and short-flow traffic, the increasing amount of different and potential variable link characteristics, and changing traffic dynamics. Additionally, the bufferbloat [52] problem, where excessive buffering induces high delays that degrade the QoE of interactive and real-time services, need to be addressed commonly by congestion control in the end system and congestion avoidance in the network. Note that [7] concludes *“there is no single congestion control approach for TCP that can universally be applied to all network environments”* and *“there are not yet the well-defined and broadly-accepted criteria to serve as good baseline for appropriately selecting a congestion control algorithm”*.

2.2 End-to-End Binary-Feedback Best-Effort Congestion Control

As this work targets TCP congestion control, we limit our scope to *sender-side*, *window-based*, *end-to-end*, and *binary feedback* approaches for *best-effort* services..

[126] argues that the *end-to-end principle* is the main reason for scalability. In end-to-end congestion control there is always one RTT feedback delay of the control loop. To preserve stability and avoid oscillations the control frequency should be equal or lower than the feedback frequency [70]. Therefore, any reaction should only be taken on a per-RTT base while feedback is received most often once per ACK, thus several times per RTT.

Further, [34] names distributedness and a minimal amount of feedback important characteristics to avoid complexity and signaling overhead. Loss only gives a *binary feedback* signal that indicates if the network is congested or not. Binary-feedback congestion control usually implements a reactive control loop where the sending rate is decreased when congestion occurs or increased in the absence of congestion. Often only *local* information extracted from the feedback signal of one single flow is used, and the control loop is *memoryless*, which means that any control decision is only based on the current state and not the history [98].

Most binary-feedback congestion control schemes can be described based on the *Chui-Jain model* [34] of linear control algorithms as reaction to a *synchronous* binary signal, which is usually loss or can also be an ECN signal. The feedback signal $y(t)$ therefore reflects the load level over an interval t, which is binary (can either be 1 or 0) and synchronous (all users receive the same signal in the respective feedback interval). $x(t)$ is the resulting rate of one user in time slot t. The increase and decrease functions are described by

$$x(t+1) = \begin{cases} \beta_I x(t) + \alpha_I & \text{if } y(t) = 0 \\ \beta_D x(t) + \alpha_D & \text{if } y(t) = 1 \end{cases} \tag{2.1}$$

where $\alpha_I \geq 0$, $\alpha_D \leq 0$, $\beta_I \geq 1$, and $0 < \beta_D \leq 1$. This general description results in four classes of control functions:

- Multiplicative Increase Multiplicative Decrease (MIMD)
  with $\alpha_I = 0$, $\alpha_D = 0$, $\beta_I > 1$, and $0 < \beta_D < 1$

- Additive Increase Additive Decrease (AIAD)
  with $\alpha_I > 0$, $\alpha_D < 0$, $\beta_I = 1$, and $\beta_D = 1$

- Additive Increase Multiplicative Decrease (AIMD)
  with $\alpha_I > 0$, $\alpha_D = 0$, $\beta_I = 1$, and $0 < \beta_D < 1$

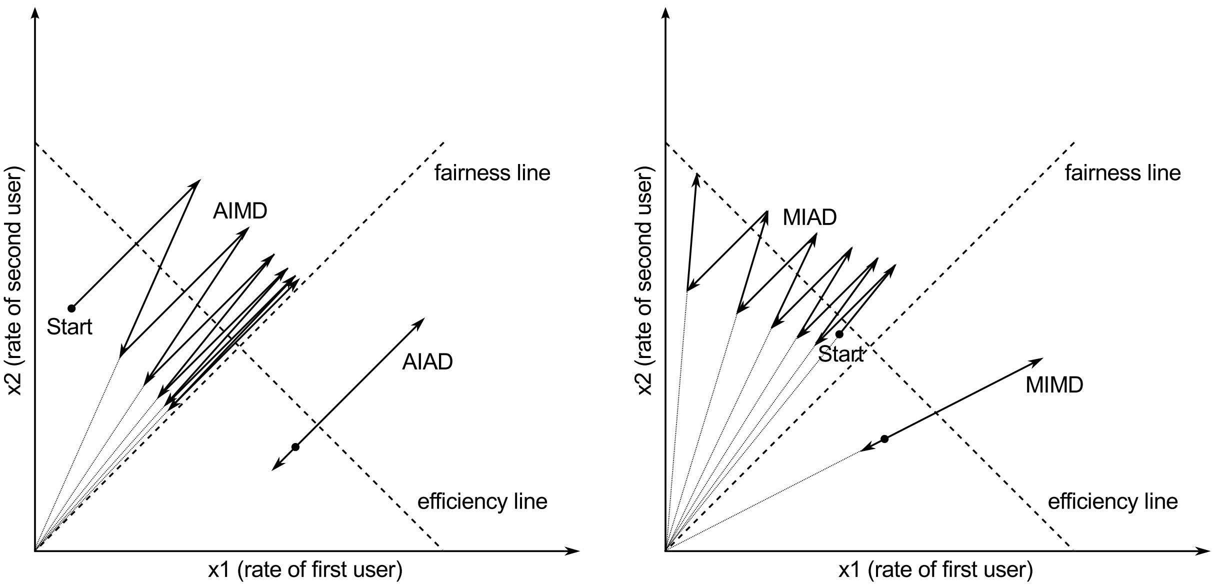


Figure 2.1: Vector diagram for two users of AIMD, AIAD, MIAD, and MIMD.

- Multiplicative Increase Additive Decrease (MIAD)
  with $\alpha_I = 0$, $\alpha_D < 0$, $\beta_I > 1$, and $\beta_D = 1$

To understand the behavior of these groups of algorithms the trajectories of rate adjustments can be represented as an n-dimensional vector diagram, where n is the number of competing users. Figure 2.1 shows the respective diagrams for two competing users. The fairness line indicates all points of equal sharing and the efficiency lines all points where the network resources are used efficiently. Therefore, their intersection represents the optimum. While all schemes can utilize network resources efficiently, only AIMD converges to equal sharing. In contrast MIMD and AIAD maintain a given sharing ratio. MIAD always converges to a situation where only one user gets all resources. Therefore, to achieve equal capacity sharing the decrease should be multiplicative and the increase should have an additive component [34]. The respective increase and decrease factors can be chosen to maintain a trade-off between responsiveness and smoothness. [55] additionally investigates *Multiplicative Additive Increase Multiplicative Decrease* (MAIMD) schemes, where only $a_D = 0$. This superclass of stable algorithms also converges to fairness and provides efficiency.

Moreover, AIMD can be generalized to a class of so-called *binomial* congestion control algorithms [19]

$$x(t+1) = \begin{cases} x(t) + \alpha / x(t)^k & \text{if } y(t) = 0 \\ x(t) - \beta x(t)^l & \text{if } y(t) = 1 \end{cases} \tag{2.2}$$

Note that it must hold that $k + l = 1$ and $l \leq 1$ to be TCP-friendly [19]. But if l can be chosen smaller than 1, the congestion response is smoother and therefore might be better applicable for streaming or even real-time media. [98] shows that only AIMD in the group of binomial algorithms provides monotonic convergence to fairness and additionally has the lowest packets loss rate. They propose to adapt the increase value α such that a constant packet loss rate can be achieved independent of the number of competing flows.

2.2.1 TCP Congestion Control

Most TCP congestion control schemes and implementations include the use of four algorithms called *Slow-Start* (SS), *Congestion Avoidance* (CA), *Fast Retransmit*, and *Fast Recovery*. The first three of these algorithms were defined early on with the introduction of TCP Tahoe congestion control (in the 4.3BSD-Tahoe kernel) in 1988 [67]. Fast Recovery was added later on with TCP Reno [138, 11, 10] to more quickly recover from short-term, mostly self-induced congestion and to allow for sending new data already during Fast Recovery. TCP NewReno [48, 49, 65] specifies another modification that improves the retransmission behavior in Fast Recovery.

In Figure 2.2 the evolution of the Congestion Window (cwnd) over time of an AIMD-based TCP congestion control scheme is shown. *cwnd* maintains the number of packets that are allowed to be in-flight during one RTT and thereby determines the sending rate (as long as the connection is not receive-window-limited). While *cwnd* could also be counted in number of bytes, in this thesis we always use the congestion window in number of packets. This is also how it is implemented in the current Linux kernel. Note, if all packets are of equal size, this is equivalent. In greedy connections, where always enough data are available to send, all packets are usually full-sized (based on the Maximum Segment Size (MSS) of the lower layer). Further, note that the *cwnd* can also be larger than the number of packets in-flight when the sending rate of the connection is, e.g., application-limited. We call the amount of data that has been sent but not yet cumulatively acknowledged *flight size* [10]. A TCP sender sends out new packets as a reaction to a received ACK as long as the flight size is smaller than the congestion window.

Figure 2.2 shows the typical *saw-tooth* behavior of TCP congestion control. At the beginning the *Slow Start* algorithm is used to careful start sending but then quickly allocate capacity. Afterwards the connection is in *Congestion Avoidance* phase where an AIMD scheme is used to periodically probe for available resources. By this principle TCP congestion control is able to adapt to changing network situation, e.g., due to starting or stopping flows. Both algorithms, Slow Start and Congestion Avoidance, are explained in more detail in the next section.

This probing scheme frequently overloads the network link and therefore induces congestion that is used as a input signal to trigger a reduction in sending rate. Due to the end-to-end feedback delay, congestion remains on the link for (at least) one RTT. Based on the aggressiveness of the increase behavior and the AQM in the network node (see Section 2.3.1), one or more packets can get lost (or ECN-marked) during this interval. E.g. with DropTail and a loss-based scheme with an increase rate of 1 packet per RTT exactly one packet usually gets lost. We define all losses or congestion notifications that occur in the same RTT as one *congestion event*, similar to the definition in [83]. The period between two congestion events is called *congestion epoch*, as also introduced by [148] as illustrated in Figure 2.2. The decrease behavior of TCP congestion control as well as its recovery mechanism are explained in section 2.2.1.2.

2.2.1.1 Slow Start and Congestion Avoidance

When a TCP flow starts a transmission there is no knowledge about the available bandwidth and the current traffic situation on a link. Therefore, to not overload the link, TCP usually starts sending with a small rate and then tries to increase its sending rate step-wise in each RTT. Just

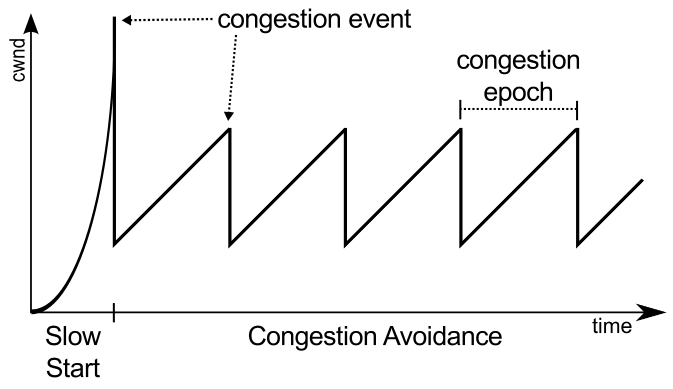


Figure 2.2: Congestion window over time.

recently, TCP's Initial Window (IW) was increased to be 10 packets [35]. Slow Start is the most commonly used start-up algorithm and aims to quickly allocate the available bandwidth. Therefore, Slow Start exponentially increases its sending rate by doubling the *cwnd* in each RTT. This can be described by a multiplicative increase, thus using $\beta_I = 2$ and $\alpha_I = 0$ as in Eq. 2.1.

Slow Start is ended when a congestion notification occurs or the Slow Start threshold *ssthresh* is reached. At the beginning of a connection most often *ssthresh* is set to the maximum possible value (depending on the implementation) resulting in an overshoot and potentially a large number of losses. If information on a previous connection to the same endpoint (with the same IP address) is available, this information can optionally be used to initialize *ssthresh* to a lower value. Further, on during the transmission, *ssthresh* is set to $flightsize/2$ (or a minimum value of 2 full-sized packets) whenever a congestion event occurs.

In Congestion Avoidance most often the AIMD scheme is applied. TCP Tahoe and Reno both increase the congestion window linearly by one packet per RTT, thus using $\beta_I = 0$ and $\alpha_I = 1$ in Eq. 2.1.

As TCP is ACK-clocked, any calculation is performed on a per-ACK basis. Assuming that each packet triggers an ACK, the respective increase behavior is usually implemented the following way.
In Slow Start

$$cwnd \leftarrow cwnd + 1 \text{ [per ACK]} \tag{2.3}$$

implements the multiplicative, exponential increase.
In Congestion Avoidance

$$cwnd \leftarrow cwnd + \frac{\alpha}{cwnd} \text{ [per ACK]} \tag{2.4}$$

with $\alpha = 1$ approximates the additive, linear increase of one packet per RTT. Note that the RTT can grow during Congestion Avoidance due to an increase in queuing delays. As *cwnd* gives the number of packets per RTT, the increase is not completely linear in Congestion Avoidance.

At the beginning of a connection, TCP sends *IW* number of packets back to back and subsequently waits about one RTT until the first ACK is received. Note that most TCP stacks acknowledge only every second packet (delayed ACKs) and therefore additionally have a heuristic to detect that the sending flow is in Slow Start and therefore acknowledge each packet separately during this phase to not slow down the sender unnecessarily. In Slow Start whenever an ACK is received by the sender, it increases its *cwnd* by one and thereby sends out two new packets back to back. Therefore, as long as the link capacity is not fully utilized, packets are not spread equally over one whole RTT but mostly bundled 'at the beginning' of one RTT. To not overload the buffer with one of these bursts and consequently get a congestion notification even though the link is not fully utilized yet, Slow Start requires about $\frac{1}{2}$ BDP of buffer [67] and of course at least sufficient buffer to store IW number of packets. Note, only if a single flow (without cross traffic) is fully utilizing the bottleneck link (later in the connection after Slow Start in Congestion Avoidance), all packets and thereby also all ACKs are spread equally over one RTT. When cross traffic is involved, packets can be clustered.

Additionally, Appropriate Byte Counting (ABC) [9] was proposed to increase the congestion window according to the number of newly acknowledged bytes. This method ensures that *cwnd* is increased by (at maximum) one full packet per RTT even when delayed ACKs are used and thereby also protects against ACK Division attacks where additional ACKs are sent by the receiver that do not acknowledge any new data but artificially inflate the sender's congestion window (if increased per ACK and not based on the number of newly acknowledged bytes). Unfortunately, in implementations where *cwnd* is maintained in packets, ABC increases too slowly if not full-sized packets are sent (for any reason) while the *cwnd* is still decreased by one for each packet that is sent out. Note that ABC is not available anymore in the current Linux kernel version.

A TCP connection falls back into Slow Start by setting *cwnd* to *IW* when the connection was idle for more than one retransmission timeout because at this point of time for sure any information on the bottleneck link of the network path is outdated.

2.2.1.2 Fast Retransmit and Fast Recovery

Initially in TCP Tahoe, any congestion notification would reset the congestion window to the minimum value. This means the connection falls back to Slow Start until *ssthresh* is reached. In general congestion is assumed either based on the detection of packet loss or due to an ECN signal received by the sender. Loss, however, is estimated based on the RTO, or, if Fast Retransmit is used, based on (initially four and now) three incoming duplicate ACKs. Therefore, Fast Retransmit immediately performs the retransmission without waiting for the RTO to expire. To further speed up this recovery process a TCP receiver sends an immediate ACK (without further delays) when an out-of-order segment is received or a segment fills a gap in the sequence number space.

With the introduction of TCP Reno, Fast Recovery was added which is entered after Fast Retransmit on the reception of duplicate ACKs. Fast Recovery handles the send-out of new data until a non-duplicate ACK is received. As a duplicate ACK can only be generated when a packet was received, it is known that one packet has left the network. This means the number of packets in flight is decreased and a new packet could be sent out. The flight size is not decreased as this packet has not been explicitly acknowledged yet, therefore the *cwnd* must be artificially inflated during Fast Recovery to potentially sent out new data on the reception of a duplicated

ACK. Note that in the Linux implementation the number of packets in flight is differently calculated than the flight size as defined in RFC 5681 [10] as SACK'ed and lost segments are already taken into account.

At the end of recovery when previously lost and now retransmitted data (or in case of ECN the first packet send after the receipt of the congestion notification) finally got acknowledged, *cwnd* is set to *ssthresh* and therefore halved as well. To not reduce the congestion window in one step and thereby not being able to send any new packets for half an RTT, Rate-Halving [105] can be used to reduce *cwnd* step-wise during Fast Recovery sending out new data only for every second acknowledgment. The current Linux implementation uses Proportional Rate Reduction (PRR) [103] instead of Fast Recovery and Rate-Halving. While Rate-Halving might decrease the congestion window as well as the Slow Start threshold to an even lower value than $flightsize/2$ when multiple packets got lost, PRR adjusts the congestion window such that always the targeted value is reached at the end of this recovery phase.

Therefore, the decrease function can be derived from Eq. 2.1 to

$$cwnd \leftarrow cwnd - \beta \cdot cwnd \text{ [on congestion event]} \tag{2.5}$$

with $\beta = 0.5$ for TCP NewReno as well as many other schemes. Note that even when multiple packets got lost or multiple congestion notifications where received, the congestion window is reduced only once per RTT.

Based on the described increase and decrease behavior the throughput of TCP NewReno in steady state according to [106] and [113] is given by

$$B(p) = \frac{1}{RTT}\sqrt{\frac{3}{2p}} \tag{2.6}$$

where p is the loss rate. Or more generalized for any arbitrary AIMD scheme according to [46, 51]

$$B(p) = \frac{1}{RTT}\sqrt{\frac{\alpha(2-\beta)}{2\beta}\frac{1}{p}}. \tag{2.7}$$

Therefore, to achieve TCP-friendliness, which means equally sharing when competing with TCP NewReno flows, the following relation needs to be true:

$$\alpha = \frac{3\beta}{2-\beta} \tag{2.8}$$

Note that this configuration only reaches equal sharing if the competing flows have the same end-to-end delay. If the competing flows do not operate based on the same RTT, not even competing TCP NewReno flows share the bottleneck link equally and therefore do not achieve *RTT-fairness*, which is the usual case in the Internet.

2.2.2 Overview of Congestion Control Schemes

In this section we introduce algorithms that either are designed based on (at least partly) similar goals or implement mechanisms that can be used to fulfill the design goals as stated in Section 1.4.

We summarize for each approach intended design goal(s) as well the algorithm itself. Additionally, we briefly discuss the relation to our design goals and summarize with a discussion at the end.

We divided the algorithms into two classes based on our main goals, support of low latency and high speed. Even though delay-based approaches are usually not explicitly targeted for low latency support, they often inherently induce only low additionally queuing delay, and therefore are surveyed in the next section. Afterwards we discuss schemes explicitly target for high-speed networks which are most often loss-based and may or may not use delay measurements additionally.

The goal of delay-based approaches most often is reducing packet loss and thereby achieve higher link utilization. This is usually reached by defining a maximum delay threshold and thereby avoiding queue overflows. Additionally, many delay-based approaches also avoid large rate oscillation (as AIMD does). Unfortunately, these approaches are usually not able to co-exist with loss-based mechanisms. An increase in delay triggers the reduction earlier than for a loss-based flow that only reduces its sending rate when the delay is maximum, consequently the queue overflows and loss occurs. In a situation where a delay-based flow is competing against a loss-based flow, the purely delay-based congestion control performs several reductions while the queuing delay further increases due to the loss-based flow. Therefore, in fact reacting to an earlier congestion signal, finally leads to starvation. This means delay-based approaches are hard to deploy in the public Internet.

To be able to compete with existing loss-based congestion control in the Internet, we premise that our algorithm design must react on loss-based congestion feedback. Even though we do not evaluate our proposal in comparison to delay-based schemes, hybrid schemes, as our proposal, usually estimate the current queue length based on delay measurements. This technique was first introduced and utilized by delay-based schemes, as presented in the next section.

Moreover, while loss can be assumed in many cases to be a consequence of congestion, delay measurements are very blurred. First of all, there might be multiple queues on the end-to-end transmission path that can introduce delay. Moreover, there are other mechanisms on the lower layers or in the end-host that add delay, e.g., due to the interrupt handling at receiver side, error protection mechanisms on the lower layer, or access control. Often not only the current delay needs to be measured but also changes in delay or the basic transmission delay without queuing delay. This base delay is often assumed as the minimum that can be seen during the connection. Unfortunately the base delay must not be constant over the whole transmission time, e.g., when the routing is changed. All mechanisms that are either purely delay-based or hybrid by at least using the delay signal in some way, must address these issues.

2.2.2.1 Low Latency (and Delay-based) Congestion Control

Next, we mainly survey selected delay-based approaches starting with TCP Vegas, one of the most well-known loss-based schemes which is based on Jain's delay-based CARD [69] and Wang/Crowcroft's DUAL [143]. TCP Vegas is part of mainline Linux kernel and also implemented in FreeBSD but not widely used [149] as it cannot co-exists with the pre-dominate loss-based schemes in the Internet. As FAST TCP is a delay-based approach targeted for long distance and therefore high latency links, it has recently been picked up by Akamai [110] with the acquisition of FastSoft.

Moreover, there are approaches that focus on the effect that delay-based congestion control reacts earlier than loss-based. Thereby they provide specific congestion control for background traffic that goes "out of the way" in the presence of foreground traffic. These approaches are called *scavenger protocols* such as TCP Nice, TCP-LP and LEDBAT. Further, TCP Rapid belongs to this group as it uses packet inter-arrival times to estimate the currently available bandwidth. TCP-LP is available in Linux but hardly used [149]. LEDBAT was proposed by BitTorrent and consequently is implemented in BitTorrent's Delivery Network Accelerator (DNA), as the only congestion control scheme, and as an experimental option in uTorrent [131]. Further, LEDBAT is implement in Windows as well as iOS for downloads of software updates.

The FreeBSD kernel further implements Hamilton-Delay (HD) congestion control, CAIA-Hamilton-Delay (CHD) congestion control, and CAIA Delay-Gradient (CDG) congestion control. These delay-based congestion control schemes are proposed with the goal to co-exist with loss based traffic by the maintainers of the FreeBSD network stack.

Finally, we introduce Data Center TCP (DCTCP), an ECN-based protocol specially designed for low latency support in data centers. DCTCP was developed by Microsoft and therefore is implemented in Windows for the use in data centers, when very low RTTs can be detected only.

TCP Vegas

TCP Vegas was designed to provide higher link utilization than TCP Tahoe or Reno by avoiding oscillations.

Therefore, TCP Vegas [24] uses an linear increase/decrease mode in Congestion Avoidance. It calculates the current throughput based on the current congestion window and currently measured RTT.

$$Actual\ (\text{throughput}) = \frac{cwnd}{RTT} \tag{2.9}$$

Further, it compares the actual throughput to an expected throughput which is calculated based on the base RTT that can be measured when the link is not congested.

$$Expected\ (\text{throughput}) = \frac{cwnd}{RTT_{min}} \tag{2.10}$$

Therefore, the difference of the actual throughput and the expected throughput gives an estimate of the current number of packets that this flow is maintaining in the queue.

$$Diff = Expected - Actual \tag{2.11}$$

If the difference $Diff$ of the actual throughput and the expected throughput is smaller than a threshold α, the congestion windows is linearly increased by one packet over the next RTT. If $Diff$ is larger than a second threshold β, it respectively is decreased by one packet over the next RTT.

$$cwnd \leftarrow \begin{cases} cwnd+1 & \text{if } Diff < \alpha \\ cwnd-1 & \text{if } Diff > \beta \end{cases} \tag{2.12}$$

where $\alpha = 2$ and $\beta = 4$. To also use delay-based feedback during Slow Start, TCP Vegas only increases exponentially during every second RTT to avoid bursts and thereby delay variations. If the difference of the actual throughput to the expected throughput in the RTT where the congestion window stays constant is larger than a third threshold γ, TCP Vegas leaves Slow Start.

TCP Vegas therefore provides the basis, by estimating the number of packets in the queue, for many loss-based as well as hybrid schemes including TCP SIAD.

FAST TCP

FAST TCP [74], based on TCP Vegas, aims to improve convergence speed in high-speed/long-latency networks by adapting its increase rate based on how far the current queuing delay is away from the target.

FAST TCP periodically updates the congestion window every second RTT based on the measured average RTT of the last RTT

$$cwnd \leftarrow min\Big\{2cwnd, (1-\gamma)cwnd + \gamma\Big(\frac{RTT_{min}}{RTT}cwnd + \alpha(cwnd, qdelay)\Big)\Big\} \tag{2.13}$$

where $\gamma \in (0,1]$ and $\alpha(w,q)$ is a function of the current congestion window and queuing delay $qdelay = RTT - baseRTT$. $baseRTT$ is the RTT of a path if there is no queuing delay. $baseRTT$ can only be measured when all queues on the path are empty. Usually the measured minimum RTT RTT_{min} is assumed to be this base line.

$$\alpha(w,q) = \begin{cases} aw & \text{if } q = 0 \\ \alpha & \text{otherwise} \end{cases} \tag{2.14}$$

where α is the number of packets that one flow wants to maintain in the queue similar as used in TCP Vegas, and a is a multiplicative increase factor when the queue is empty. This congestion window change is performed over one RTT while in the second RTT the window stays constant and thereby a new average RTT can be measured.

Therefore, TCP FAST was one of the first approaches where the increase rate was adapted dynamically based on the network feedback. TCP SIAD also adapts the increase factor dynamically but only from one congestion epoch to the next and not within one congestion epoch as FAST TCP (except in Fast Increase). This is not necessary as TCP SIAD additionally aims to not underutilize the link when a decrease is performed.

TCP Nice

TCP Nice [141] is designed for background transfers, thus to minimize the interference on foreground traffic, but still to be able to utilize the available spare network capacity efficiently.

With TCP Nice congestion is detected when more than a fraction f of packets measures a higher queuing delay than a threshold t. Therefore, for each incoming ACK it is checked if the currently measured RTT is larger than $RTT_{min} + (RTT_{min} + RTT_{max}) \cdot t$, or respectively

$$RTT_{curr} > (1-t) \cdot RTT_{min} + t \cdot RTT_{max} \tag{2.15}$$

where RTT_{curr} is the currently measured RTT of this sample and RTT_{max} an estimate of the RTT when the bottleneck queue was full. If this is true for more than $f \cdot cwnd$ samples during one RTT, the congestion window is halved. Otherwise TCP Vegas' Congestion Avoidance is performed. This means TCP Nice decreases the congestion window multiplicatively in the presented of cross traffic to faster release the capacity. Note that TCP Nice even allows a congestion window smaller than one packet.

TCP Nice re-introduces the multiplicative decrease also for delay-based schemes, acknowledging that quickly releasing capacity to other flows is important for convergence.

TCP Low Priority (TCP-LP)

In contrast to TCP Nice, TCP-LP [91] is based on per-packet estimates d_i of the One-Way-Delay (OWD) using the TCP Timestamp Option (TSOpt).

TCP-LP calculates the smoothed OWD by

$$sd_i = (1-\gamma)sd_{i-1} + \gamma d_i \tag{2.16}$$

where γ is the smoothing factor, and an early congestion notification is assumed if

$$sd_i > d_{min} + (d_{max} - d_{min})\delta. \tag{2.17}$$

With the receipt of the initial early congestion notification, TCP-LP halves the congestion window and enters an *inference phase* for one RTT. During this phase the congestion window is not increased, and when another early congestion notification is received, the congestion window is set to 1 to quickly yield for high priority traffic. Otherwise TCP-LP restarts increasing the window by one packet per RTT.

Therefore, TCP-LP still implements an AIMD scheme but is not based on loss but on an early congestion notification estimated by OWD measurements.

Low Extra Delay BAckground Transport (LEDBAT)

LEDBAT [131] is another algorithm for background traffic which has be proposed by BitTorrent for peer-to-peer file sharing over User Datagram Protocol (UDP) and is also based on OWD measurements.

LEDBAT defines a $TARGET$ value of not more than 100 ms for the maximum queuing delay and adjusts its window based on the difference from the currently measured queuing delay to the target:

$$off\_target = \frac{TARGET - qdelay}{TARGET} \tag{2.18}$$

$$cwnd += GAIN \cdot off\_target \cdot \frac{bytes\_newly\_acked \cdot MSS}{cwnd} \tag{2.19}$$

where *GAIN* can be used to speed up the adaptation. LEDBAT does not increase by more than *ALLOWED_INCREASE* packets per RTT though, where *ALLOWED_INCREASE* should be 1 to remain TCP-friendly. Further, it adapts its window based on the number of newly acknowledged bytes, as ABC would do. The queuing delay *qdelay* is calculated based on a filtered value of the current OWD and the base delay which is the minimum measurement sample seen over the last $BASE\_HISTORY = 10$ minutes. LEBDAT assumes, as TCP Nice, one measurement sample per ACK but of the OWD and not the RTT. When loss or ECN marks occur, LEDBAT halves its congestion window. Therefore, if *TARGET* is larger than the maximum waiting time of the queue, LEDBAT basically falls back to TCP NewReno but increases even slower when a queue builds up.

By filtering the delay measurements and updating the base delay, LEDBAT addresses measurement errors due to additionally delays in other layers or changes in the end-to-end delay, e.g., because of route changes. Further, LEDBAT discussed additionally algorithms to handle the different clock offset and skew of the sender and receiver. This is important to address for any delay-based or hybrid scheme.

Competitive and Considerate Congestion Control Protocol (4CP)

Competitive and Considerate Congestion Control Protocol (4CP) [97] is also designed for file transfers realizing a low-priority service.

CP4 maintains a virtual window *win* that can also become negative but is bounded between $-minbnd$ and *maxcwnd*.

$$win \leftarrow wnd + \frac{1}{cwnd} \text{ [per ACK]} \tag{2.20}$$

$$win \leftarrow wnd - \frac{1}{tarp \cdot cwnd} \text{ [on congestion event]} \tag{2.21}$$

where $tarp \leftarrow tarp + a \cdot \frac{g(tarp)-cwnd}{cwnd}$ is a target loss probability (tarp) and per default $g()$ is the loss equation for TCP NewReno that can be derived from Eq. 2.6. The actual congestion window *cwnd* is equal to *win* as long as it is larger than a minimum value *mincwnd*.

$$cwnd \leftarrow max(wnd, mincwnd) \tag{2.22}$$

TCP RAPID

TCP RAPID [84] congestion control uses chirping for estimating the available capacity by sending all data continuously in so-called *chirps* of $N = 30$ packets. Chirping was initially proposed for the pathChirp bandwidth estimation tool [123] based on measurements of the inter-arrival time of packets. With RAPID congestion control the rates of the packets within a chirp are selected in such a way that the average rate converges towards the currently estimated available bandwidth. This means TCP RAPID utilizes only the newly measured spare capacity

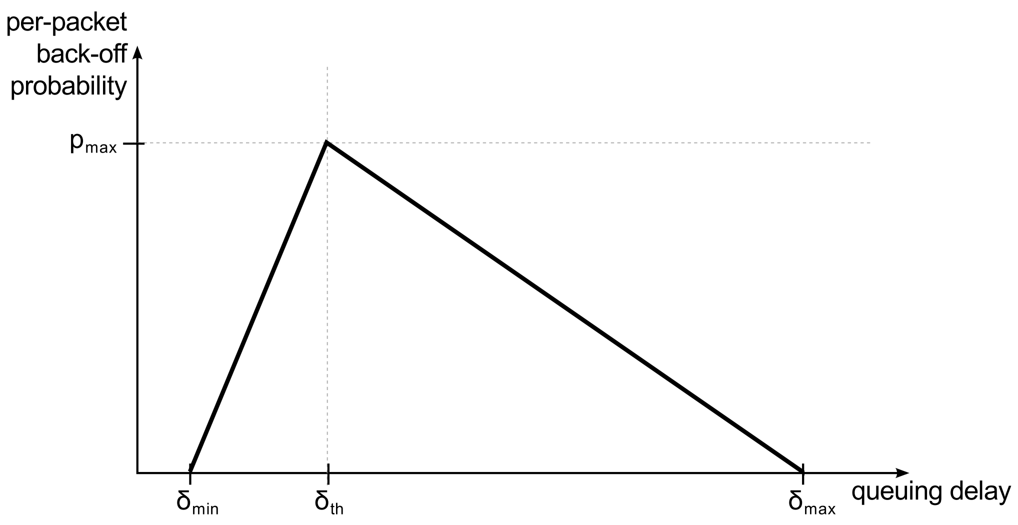


Figure 2.3: Hamiliton-Delay (HD): per-packet back-off probability function.

on a network link and therefore makes TCP RAPID a scavenger protocol. TCP RAPID is not ACK-clocked but needs to send packets at specific points of time. Therefore, it is more complex to implement in Linux [87].

If ACK-clocking is not required, bandwidth estimation as proposed by pathChirp and implemented by TCP RAPID could be used in high-speed network to quickly grab new capacity. Note, as multiple flows might be competing for the same resource, a congestion control scheme should not try to grab all estimated available capacity at once.

Hamiliton-Delay (HD), CAIA-Hamilton-Delay (CHD), and CAIA Delay-Gradient (CDG)

While delay-based congestion control schemes usually cannot co-exist with loss-based approaches, there are also attempts to design a class of delay-based AIMD [94] algorithms that can co-exist with traditional loss-based congestion control but still maintain low queuing delay otherwise. Therefore, Hamilton-Delay (HD) congestion control[30, 29] bases its probabilistic decrease function on a delay target value that can be dynamically adapted. More precisely the delay target is increased if loss-based cross traffic is detected and decreased to a minimum otherwise. Therefore, on each ACK, HD compares a random number X (between 0 and 1) to the current back-off probability $g(q_i)$, as shown in Figure 2.3, where q_i is the current queuing delay and δ_{th} is a threshold delay value. If the X is larger, HD halves its congestion window or, otherwise, increases by one packet per RTT.

CAIA-Hamilton-Delay (CHD) [62] additionally aims to cope with non-congestion packet losses as well as to improve the co-existence with loss-based cross traffic by introducing a shadow window. CAIA Delay-Gradient (CDG) [64], moreover, improves the robustness to noise in the delay measurements by using a delay gradient instead of the target delay threshold. This delay gradient is based on the differences in delay seen for the minimum and maximum delay in last and the previous RTTs.

The proposed delay-based AIMD schemes only modify the decrease behavior to induce low additional queuing delay in the absence of loss-based traffic. However, TCP SIAD also aims to improve the increase behavior to be used in high-speed networks. Further, we aim to allow operators to configure small queues without an decrease in utilization to permanently have low queuing delay and not only in absence of other traffic.

Relentless TCP

Relentless TCP [102] is designed to provide both scalability and high utilization with small buffers. Still similar to TCP NewReno, it simply increases by one packet per ACK, but decreases by one for each loss (separately within one congestion event). This approach keeps link utilization high (and the buffer completely filled but also induces a fixed and very high loss rate and therefore does not allow standard TCP to compete at all. Therefore, Relentless TCP is not applicable to the Internet.

Data Center TCP (DCTCP)

Data Center TCP (DCTCP) consists mainly of two parts, an ECN-based congestion control scheme in the end-host and an AQM scheme in the bottleneck network node that keeps queuing delay and delay variance very low. Therefore, DCTCP implements three changes:

1. a specific Random Early Detection (RED) configuration in the network nodes,
2. a different decrease function as reaction to congestion in the sender, and
3. a more accurate congestion feedback mechanism from the receiver to the sender.

As DCTCP implies changes in both end-hosts and the bottleneck network node, it was developed for operation in data centers only.

The AQM scheme for DCTCP operation is in principle simple: if the current instant queue is larger than a certain threshold K, every arriving packet is ECN-marked. This mechanism can be implemented as a specific parameterization of RED [50] where $Min\_Thresh = Max\_Thresh = K$ and $w = 1$ (see Section 2.3.1).

Further, a DCTCP sender updates the congestion window *cwnd* according to the following equation on congestion notification

$$cwnd \leftarrow (1 - \alpha/2) \cdot cwnd \tag{2.23}$$

where α is the moving average of the fraction of marked packets in the last RTT.

$$\alpha \leftarrow (1 - g) \cdot \alpha + g \cdot F \tag{2.24}$$

where F is the fraction of the ECN-marked packets in the last RTT and g is a weighting factor that is recommended to be $1/2^4$ [8]. This congestion control algorithm allows the sender to gently reduce the congestion window depending on the extent of congestion and at maximum halving the window as TCP NewReno would do.

With the ECN feedback algorithm as described in Section 2.3.2 only one congestion feedback signal can be sent per RTT. This is appropriate for conventional TCP congestion control which reacts only once per RTT but not for DCTCP where the reduction depends on the number of markings per congestion event. Therefore, DCTCP aims to get exactly one ECN-Echo for each congestion-marked packet. However, to be able to use delayed acknowledgements, Alizadeh et al. define in [8] a two state machine for handling ECN feedback.

DCTCP is one of the few algorithms that is explicitly designed for low latency support, which is one of our main design goals, but cannot be deployed in the public Internet.

2.2.2.2 High-Speed (and Loss-based) Congestion Control

All algorithms presented in this section are mainly motivated by the scalability problem in high-speed or long-latency networks of the traditional TCP congestion control as TCP NewReno. This is one common goals with the approach presented in this work. Some of the algorithms discussed below also share other goals but cannot fulfill completely all goals as listed in Section 1.4.

Note that Scalable TCP, High Speed TCP, TCP BIC, TCP Cubic, TCP Illinois, H-TCP, TCP Westwood, and YeAH-TCP are available in the mainline of the Linux kernel. Except for TCP Cubic which is the default configuration for Linux, none of the schemes see strong usage [149]. We compare TCP SIAD to Scalable TCP, TCP High Speed TCP, TCP Cubic, TCP Illinois, and H-TCP in our evaluation. TCP Cubic is also implemented in FreeBSD and Compound TCP is the default algorithm in Windows [1, 95].

Scalable TCP

Scalable TCP [81] uses a straight-forward approach to provide high utilization in high BDP networks by using a MIMD scheme with $\alpha = 0.01$ and $\beta = 0.125$.

$$cwnd = cwnd + 0.01 \tag{2.25}$$

$$cwnd = cwnd - 0.125 \cdot cwnd \tag{2.26}$$

Note, MIMD schemes do not converge under synchronization or without any randomness and, even worse, also not in case of RTT-unfairness [34].

Scalable TCP provides a constant feedback rate and therefore is highly scalable, but might not converge and, as it has been explained in Section 4, usually induces a very high loss rate. Further, Scalable TCP does often cause a standing queue due to the fixed, small decrease factor.

High Speed TCP

High Speed TCP [43] is an alternative approach to Scalable TCP targeted for high-speed networks.

To achieve scalability at high speed but also stay TCP-friendly at low speed, it uses two operation modes: If the window is smaller than $Low\_Window = 38$ packets it operates similar to TCP NewReno, while above Low_Window the increase and decrease factors α and β are no longer fixed but functions over the window size $\alpha(cwnd)$ and $\beta(cwnd)$.

$$cwnd = cwnd + \frac{\alpha(cwnd)}{cwnd} \text{ [per ACK]} \tag{2.27}$$

$$cwnd = (1 - \beta(cwnd)) \cdot cwnd \text{ [on congestion event]} \tag{2.28}$$

According to equation 2.6 TCP NewReno operates with a window size of $Low\_Window = 38$ at a loss/congestion feedback rate of $Low\_p = 10^{-3}$. In the high-speed mode, High Speed TCP aspires a loss rate of $High\_P = 10^{-7}$ at a window size of $High\_Window = 83000$. By calculating a linear slope between these two points in log scale, High Speed TCP implements a response function of

$$cwnd = \left(\frac{p}{Low\_P}\right)^{S} \cdot Low\_Window \tag{2.29}$$

with $S = \frac{\log(High\_Window) - \log(Low\_Window)}{\log(High\_P) - \log(Low\_P)}$ in high-speed mode. To calculate the decrease factor β, a desired decrease factor $High\_Decrease = 0.1$ is specified for a window size of $High\_Window$ packets.

$$\beta(cwnd) = (High\_Decrease - 0.5)\frac{\log(cwnd) - \log(Low\_Window)}{\log(High\_Window) - \log(Low\_Window)} + 0.5 \tag{2.30}$$

This adaption of the decrease factor enables High Speed TCP to utilize the link with smaller and smaller buffers the larger the BDP gets. Subsequently, the increase factor is calculated to stay TCP-friendly according to Eq. 2.7 by

$$\alpha(cwnd) = cwnd^2 \cdot p(cwnd)\frac{2\beta(cwnd)}{2 - \beta(cwnd)} \tag{2.31}$$

where $p(cwnd)$ is given by the envisioned response function of $cwnd = 0.12/p^{0.835}$ for $High\_Window$ and $High\_P$ as

$$p(cwnd) = \frac{0.078}{cwnd^{1.2}}. \tag{2.32}$$

In the Linux kernel implementation $\alpha(cwnd)$ and $\beta(cwnd)$ have been discretized to pre-compute a table as given in appendix B of RFC3649 [43].

High Speed TCP defines two modes based on fixed thresholds. The principle of using different modes to cope with high-speed networks had been adopted by various proposals. In contrast to High Speed TCP, we aim to distinguish between steady state and situations where the network conditions are changing and therefore would need to dynamically adapt any thresholds.

TCP BIC

TCP BIC [148] considers not only TCP-friendliness and bandwidth scalability, but also RTT-fairness.

TCP BIC has two modes for the increase, namely *binary search increase* and *additive increase*, combined with a multiplicative decrease with $\beta = 0.125$ similar to Scalable TCP. In binary search increase the window size at the time of a congestion notification is stored as the maximum window $cwnd_{max}$. The window after the reduction is stored as the minimum window and a target value $cwnd_{target}$ is calculated as the midpoint between minimum and maximum. This binary search increase is combined with an additive increase strategy as the window is increased to the target in maximum steps of $S_{max} = 32$ for TCP-friendliness. When the target is reached, the current target is used as the new minimum and a new target is again calculated the midpoint between this new minimum and the maximum. This is repeated until the difference is no larger than the minimum increment threshold $S_{min} = 0.01$. Therefore, as long as the congestion window is smaller than $cwnd_{max}$ and the distance of the target to $cwnd_{max}$ is larger than S_{min} the following increase is performed

$$cwnd = cwnd + \frac{min(cwnd_{target} - cwnd, S_{max})}{cwnd} \text{per ACK} \tag{2.33}$$

This means this increase strategy initially increases linearly and reduces the increase as it gets closer to the saturation point W_{max}. As the number of losses depends on the increase rate, this binary increase strategy reduces packet loss compared to other high-speed approaches. If the window is larger or equal to $cwnd_{max}$, there is no new target value. In this case the increase rate is set to $\alpha = 1$ packet per RTT and α is increased by 1 every RTT up to $\alpha = S_{max}$. Therefore, TCP BIC probes more carefully around the saturation point with a low increase rate to avoid large number of losses but then increases the increase rate until it reaches a linear increase rate of S_{max}. Moreover, if the new maximum after a window reduction is smaller than the previous maximum, TCP BIC sets the new maximum to $cwnd_{max} = (cwnd_{prev\_max} - cwnd_{min})/2$. This strategy is called Fast Convergence, as in this case a new flow usually grabs some capacity and TCP BIC aims to yield some space.

In fact TCP BIC implements two modes, the binary increase and a fast mode when the target is unknown following a similar idea as in Slow Start. This approach goes in-line with our design goal to quickly grab newly available capacity. Further, TCP BIC treats the case separately when less capacity gets available in Fast Convergence.

TCP Cubic

TCP Cubic [59] is "designed to simplify and enhance the window control of BIC" [122] by using a cubical window growth function

$$cwnd = C(t-K)^3 + cwnd_{max} \tag{2.34}$$

where $C = 0.4$ is a scaling factor, t is the elapsed time from the last window reduction and $K = \sqrt[3]{cwnd_{max}\beta/C}$ with decrease factor of $\beta = 0.8$. On congestion, the congestion window is still decreased multiplicatively but respectively with a decrease factor of $\beta = 0.8$.

$$cwnd = \beta \cdot cwnd = 0.8 \cdot cwnd \text{ [on congestion event]} \tag{2.35}$$

This configuration is TCP-friendly if the packet loss rate is larger than 0.000144. Based on this, the increase factor α is calculated dynamically at arrival of each ACK. In the Linux code the minimum increase is set to 1/100 packet per RTT, similar to $S_{min} = 0.01$, the maximum increase is limited to 1 packet per ACK (as in Slow Start), and β is 717/1024 $\sim$ 0.7.

TCP Cubic retains the idea of using $cwnd_{max}$ as the expected saturation point, but combines an increase behavior similar to TCP BIC's binary increase and fast increase above $cwnd_{max}$ to one increase function.

TCP Africa

TCP Africa [83] addresses the problem that most high-speed proposals have an increased aggressiveness in high-speed domains to allocate new capacity quickly which can also lead to poor fairness with, e.g., TCP Reno in these kind of network scenarios. Note that is similar to the idea as proposed by Adaptive Reno (ARENO) [134].

Therefore, TCP Africa introduces two modes, the *fast mode* and the *slow mode*. In slow mode TCP Africa slows down the increase rate when the path gets closer to the saturation point as queuing delay increases. Based on a threshold α that gives the number of packets that can be accepted to be queued at the bottleneck, TCP Africa switches from *fast mode* to *slow mode*. Therefore, as long as

$$\frac{cwnd \cdot (RTT_{avg} - RTT_{min})}{RTT_{avg}} < \alpha \tag{2.36}$$

where RTT_{avg} is an exponentially smoothed RTT estimate and RTT_{min} the minimum RTT observed so far on the path, TCP Africa stays in *fast mode* and increases its window by

$$cwnd = cwnd + \frac{fast\_increase(cwnd)}{cwnd} \tag{2.37}$$

Note, it is proposed to use the same increase function for $fast\_increase(W)$ as used in High Speed TCP (see equation 2.31). In *slow mode* the same increase function than for TCP Reno is used, namely

$$cwnd = cwnd + \frac{1}{cwnd} \tag{2.38}$$

to provide fairness to TCP Reno-like flows. α is proposed to be 1.65.

TCP Africa is a hybrid congestion control scheme that reduces its window based on loss but also uses a measurement of the queuing delay to estimate the number of packets in the queue. While in TCP SIAD we aim to use the same estimate to adapt the decrease factor such that we can provide high link utilization with small queues, TCP Africa introduces a threshold based on the queue length to adapt the increase behavior and thereby improve TCP-friendliness.

Compound TCP

Similar to TCP Africa, Compound TCP [139] also uses delay information to improve fairness, even though the approach is quite different. Instead of switching between modes, Compound

TCP adds, in Congestion Avoidance only, an additional so-called Delay Window *dwnd* to the congestion window *cwnd*. The delay window provides a higher increase in sending rate when the link is underutilized and thus the bottleneck queue is empty. When the delay increases, *dwnd* becomes zero and Compound TCP increases with the same increase rate than TCP Reno of 1 packet per RTT.

Therefore, Compound TCP has an effective sending window of

$$win = min(cwnd + dwnd, awnd) \tag{2.39}$$

where *awnd* is the advertised window from the receiver. Consequently the increase function must be

$$cwnd = cwnd + \frac{\alpha}{cwnd + dwnd} \text{ [per ACK].} \tag{2.40}$$

Additionally *dwnd* is updated once every RTT by

$$dwnd(t+1) = \begin{cases} dwnd(t) + (\alpha \cdot win(t)^k - 1) & \text{if } diff < \lambda \\ (dwnd(t) - \zeta \cdot diff) & \text{if } diff \geq \lambda \\ (win(t) \cdot (1-\beta) - cwnd/2) & \text{if loss is detected} \end{cases} \tag{2.41}$$

where $diff$ is estimated in an identical way as done by TCP Vegas and given in equation 2.11 and λ is a threshold configured to 30 packets. α, β, and k are parameters that tune the overall increase as well as decrease behavior and are set to $\alpha = 1/8$, $\beta = 1/2$, and $k = 0.75$ to achieve comparable scalability to HSTCP. ζ defines how quickly *dwnd* decreases to zero when the delay increases. Note that *dwnd* cannot become negative and is additionally set to zero when *cwnd* is smaller than 41 packets.

TCP Illinois

TCP Illinois [96] aims to quickly allocate bandwidth when the queue is empty and thus the queuing delay is below a certain thresholds.

Therefore, TCP Illinois adapts the increase and decrease factors α and β once per RTT in Congestion Avoidance by the following functions, as also shown in Figure 2.4

$$\alpha = f_1(d_a) = \begin{cases} \alpha_{max} & \text{if } d_a \leq d_1 \\ \frac{\kappa_1}{\kappa_2 + d_a} & \text{otherwise} \end{cases} \tag{2.42}$$

$$\beta = f_2(d_a) = \begin{cases} \beta_{min} & \text{if } d_a \leq d_1 \\ \kappa_3 + \kappa_4 \cdot d_a & \text{if } d_2 < d_a < d_3 \\ \beta_{max} & \text{otherwise} \end{cases} \tag{2.43}$$

where $d_a = T_a - T_{min}$ is the current average queuing delay. T_a is the average RTT measured over the last RTT and T_{min} is the minimum RTT ever seen. With a maximum average queuing delay $d_m = T_{max} - T_{min}$, $\alpha_{min} = f_1(d_m)$ and consequently

$$\kappa_1 = \frac{(d_m - d_1)\alpha_{min}\alpha_{max}}{\alpha_{max} - \alpha_{min}} \tag{2.44}$$

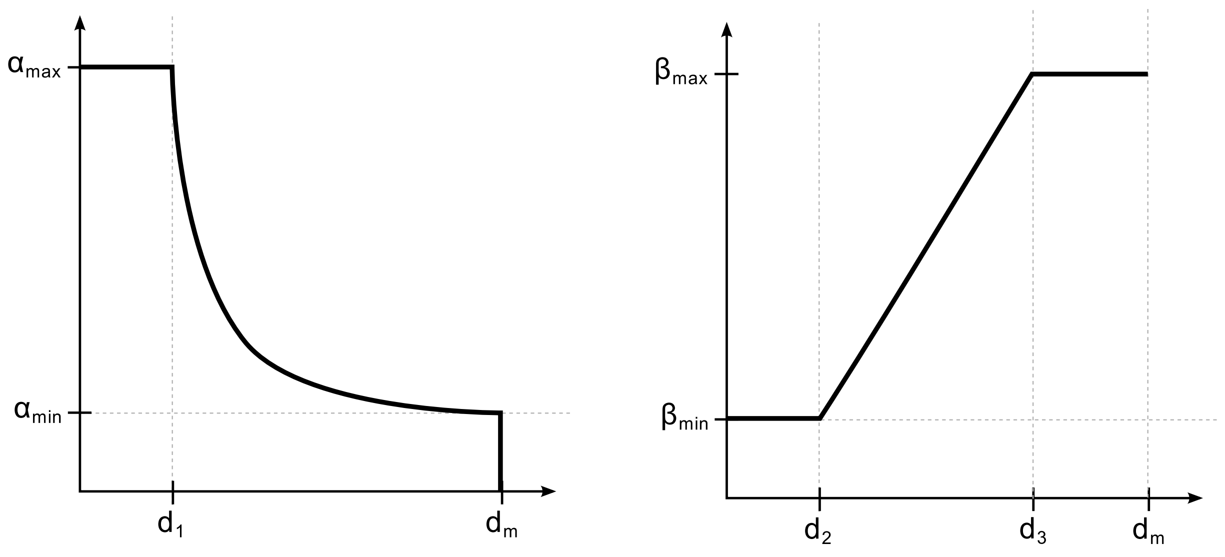


Figure 2.4: TCP Illinois: schematic curves for α and β over d_a.

$$\kappa_2 = \frac{(d_m - d_1)\alpha_{min}}{\alpha_{max} - \alpha_{min}} - d_1 \tag{2.45}$$

$$\kappa_3 = \frac{\beta_{min} d_3 - \beta_{max} d_2}{d_3 - d_2} \tag{2.46}$$

$$\kappa_4 = \frac{\beta_{max} - \beta_{min}}{d_3 - d_2} \tag{2.47}$$

where $\alpha_{min} = 0.1$, $\alpha_{max} = 10$, $\beta_{max} = 1/2$, $\beta_{min} = 1/8$ and $d_i = \eta_i \cdot d_m$ with $\eta_1 = 0.01$, $\eta_2 = 0.1$ and $\eta_3 = 0.8$. Moreover, α is only calculated dynamically when the delay estimate stays above the threshold for at least 5 samples. Note every ACK provides a new sample. And if the congestion window is smaller than 10 packets, constant values are used, namely $\alpha = 1$ and $\beta = 1/2$.

Similar to our approach TCP Illinois calculates the increase and decrease factors dynamically, even though targeting different goals. While TCP Illinois calculates both values once per RTT, we envision to re-calculate both parameters only once per congestion epoch. Moreover, whenever a congestion event occurs due to a buffer overflow and thus d_m is reached, β_{max} is used in TCP Illinois, that means the window is halved. Therefore, in steady state operation the dynamic calculation of the decrease factor β does not have any influence.

H-TCP

As introduced in the previous section, there is also class of delay-based AIMD algorithms that aim to co-exist with loss-based traffic while otherwise maintaining low queuing delay. In both cases, when congestion is detected by loss or by an increase in delay, AIMD reduces its sending rate multiplicatively by a factor β:

$$cwnd = \beta \cdot cwnd \text{ [on congestion event]} \tag{2.48}$$

Note that β in this case is $1-\beta$ from equation 2.5. To not underutilize the link, when the sending rate is reduced early due to a delay-based congestion indication, it is also proposed to adapt β based on the current estimation of the queue delay [135, 94], similar as derived by TCP Vegas in 2.11, TCP Africa in 2.36 and Compound TCP (see Eq. 2.36).

$$\beta = \frac{RTT_{min}}{RTT(t)} \tag{2.49}$$

This approach for calculating the decrease factor was further used by H-TCP [93]. H-TCP restricted β to be in the range of [0.3, 0.5]. Therefore, one H-TCP flow cannot utilize the link when the buffer is smaller than 0.6·BDP. Further, H-TCP resets β to 0.5 if the throughput of the flow changes more than 20% from one congestion event to the next. This provides faster convergence when the network conditions have changed, e.g., due to a new starting flow.

For the increase behavior H-TCP aims to address scalability in high-speed and long-distance networks by switching between a high-speed and low-speed mode similar to HSTCP. While HSTCP only changes the increase factor α, H-TCP has additionally to maintain a certain relationship between α and β to provide fairness to TCP Reno-like flows. Therefore, on each ACK, α is calculated by

$$\alpha = \begin{cases} 1 & \Delta \leq \Delta^L \\ 1 + 10(\Delta - \Delta^L) + (\frac{\Delta - \Delta^L}{2})^2 & \Delta > \Delta^L \end{cases} \tag{2.50}$$

and then set to

$$\alpha = 2(1-\beta)\alpha \tag{2.51}$$

where Δ is the time since the last congestion event and Δ^L is a threshold to switch between high and low speed mode. In the Linux implementation Δ^L is set to the minimum time period that the internal clock provides and thereby is also used for TSOpt. Additionally, in the Linux implementation α can optionally be scaled with the minimum RTT to achieve RTT-fairness. H-TCP is in the kernel mainline since version 2.6.13.

However, while H-TCP already implements a decrease behavior fitting our design goals, it addresses scalability by the same approach as High Speed TCP that still has a dependency on the available bandwidth. Further, H-TCP addresses only smaller buffers by allowing a decrease factor between 0.3 and 0.5 but still causes a standing queue in case of large buffers where the decrease factor would need to be larger than 0.5 in order to empty the queue before the sending rate is increased again.

TCP Westwood

TCP Westwood (TCPW) [31] aims to improve congestion control for wireless scenarios where losses does not only occur because of congestion but also link layer errors. Therefore, it uses a bandwidth estimate based on the ACK reception rate and the acknowledged amount of data. A bandwidth estimate sample b_k is calculated at the arrival time t_k of each ACK acknowledging d_k bytes of data by

$$b_k = d_k/\delta_k \tag{2.52}$$

where $\delta_k = t_k - t_{k-1}$ and t_{k-1} is the arrival time of the previous ACK. Further, these samples are low-pass filtered using the Tustin approximation as explained in detail in [31] to filter noise and delays by delayed ACKs. Additionally a mechanism to determine the number of acknowledged packets is proposed to correctly handle delayed and accumulated ACKs. Finally this bandwidth estimate B is used to set the congestion window after a congestion notification by

$$cwnd = (B \cdot RTT_{min})/seg\_size \tag{2.53}$$

where seg_size is number of payload bits and RTT_{min} is the smallest RTT seen during the connection. The increase in Congestion Avoidance and Slow Start is the same than used by TCP NewReno.

Similar to H-TCP and also SIAD, TCPW aims to reduce the queuing delay. While TCPW uses a bandwidth estimate based on the ACK inter-arrival time, TCP SIAD directly uses the ratio of the maximum and minimum RTT.

YeAH-TCP

YeAH-TCP [17] names as one design goal the ability to achieve high utilization with small link buffers.

To reach this goals, YeAH-TCP implements the same loss-based decrease behavior than TCP Westwood [31]. Further, for the increase behavior YeAH-TCP implements a “Fast” and a “Slow” mode, similar to TCP Africa. In Fast mode the increase behaves like Scalable TCP and in Slow mode like TCP Reno. YeAH-TCP is in Fast mode if the estimated number of packets in the bottleneck queue q is smaller than Q_{max} and the ratio between the queuing delay and the propagation delay L is smaller than $1/\varphi$ with

$$qdelay = RTT_{mincurr} - RTT_{min} \tag{2.54}$$

where $RTT_{mincurr}$ is the minimum estimated in the current window of $cwnd$ packets and RTT_{min} is the total minimum thus the base RTT and

$$L = \frac{qdelay}{RTT_{min}} \tag{2.55}$$

$$q = \frac{qdelay}{RTT_{mincurr}} cwnd. \tag{2.56}$$

Q_{max} and φ are tunable parameters where Q_{max} is the maximum number of packets that single flow is allowed to buffer and $1/\varphi$ is the maximum congestion level. Further, when YeAH-TCP is in Slow mode a precautionary decongestion algorithm is used that reduces the congestion window by q and sets $ssthresh$ to $cwnd/2$ when $q > Q_{max}$ to keep the queue small. Note YeAH-TCP restricts the decrease to be in the range of $[0.125, 0.5] \cdot cwnd$. Further, this algorithm is only used when not competing with greedy loss-based flows. Therefore, YeAH-TCP additionally proposes a detection mechanism to fall back to TCP NewReno’s increase bahavior that is based on counting the numbers of RTTs that YeAH-TCP stays in one or the other of the two modes.

This means YeAH-TCP does not only aim for high utilization with small buffers but also tries to keep the queuing delay low when not competing with loss-based traffic. We do not explicitly target a different behavior if alone on the link as we, instead, aim to enable the use of small buffers everywhere in the Internet that, even when fully filled, induce only small additional queuing delay.

TCP Fusion

TCP Fusion [76] aims to adapt the decrease such that the link stays utilized even with small buffer sizes. But at the same time it aims to stay TCP-friendly.

Therefore, it implements the same decrease behavior as TCPW-RE (TCP Westwood Rate Estimation) [142], and as also used by H-TCP, but limits the decrease to not more than half the window to ensure to get an equal share of the capacity when competing with TCP Reno-like cross traffic.

$$cwnd \leftarrow max(\frac{RTT_{min}}{RTT}cwnd, \frac{cwnd}{2}) \text{ [on congestion event]} \tag{2.57}$$

For the increase it also calculates the difference $Diff$ of the actual throughput and the expected throughput as given in Eq. 2.11 and a virtual congestion window of

$$cwnd \leftarrow \begin{cases} cwnd + \frac{W_{inc}}{cwnd} & \text{if } Diff < \alpha \\ cwnd + \frac{-diff+\alpha}{cwnd} & \text{if } Diff > 3 \cdot \alpha \\ cwnd & \text{otherwise} \end{cases} \tag{2.58}$$

Additionally TCP Fusion calculates an equivalent congestion window for TCP Reno $cwnd_{Reno}$. It only uses the congestion window calculation described above if the caculated congestion window value is larger than $cwnd_{Reno}$, otherwise $cwnd = cwnd_{Reno}$.

As we also target small buffers, TCP SIAD uses the same decrease function but without any restriction. However, we do not aim for TCP-friendliness in general.

Adaptive Congestion Control Protocol (ACP)

Adaptive Congestion Control Protocol (ACP) [75] introduces a congestion control policy which is named Adaptive Increase Adaptive Decrease scheme.

ACP decreases the congestion window by the estimated queue fill of a flow, in case of a congestion notification based on loss, and increases based on a “fairness ratio” metric for fast convergence to the equal sharing equilibrium. Therefore, it calculates the queue growth by

$$\zeta^T = \frac{P^T}{T + \delta t_d}\delta t_d \tag{2.59}$$

where T is the control interval, P^T the number of packets sent during T and δt_d the increase in delay. Based on this, a fairness ratio can be estimated by

$$F = \frac{\zeta_c^t}{\delta \cdot cwnd}. \tag{2.60}$$

Therefore, ACP measures changes in fairness during a period $t_c = 200ms$ assuming that resources where shared fairly at the beginning of the measurement period. Based on F and $\Phi = goodput - throughput (= -Diff)$, as used by TCP Vegas, ACP implements three increase states, *fast probing*, *humble increase*, and *fairness claiming*:

$$cwnd(t+t_c) \begin{cases} cwnd(t) + \alpha \lfloor \frac{t-t_0}{t_c} \rfloor & \text{if } \Phi \geq 0 \\ cwnd(t) + \alpha & \text{if } \Phi < 0, F \geq 1 \\ cwnd(t) + \alpha + \kappa(1-F)^2 & \text{if } \Phi < 0, 0 \leq F < 1 \end{cases} \tag{2.61}$$

where t_0 is the arrival time of the last congestion notification, $\alpha = 1$ gives an increase factor similar to TCP Reno, and $\kappa = 25$ provides fast convergence such that $\kappa(1-F)^2 = 1$ when $F = 0.8$. Further, ACP implements an *early control state* to additionally decrease if the current queue length $\zeta(t)$ is larger than $\gamma + cwnd(t)$ or a competing flow performed a window reduction. This is estimated if the goodput-throughput difference ϕ over the control period t_c is positive while $\Phi < 0$. In early control state or based on a congestion notification, the congestion window is decrease by

$$cwnd \leftarrow cwnd(t) - \zeta(t) \tag{2.62}$$

ACP decreases its sending not only in case of congestion notified by loss but also based on an early notification based on delay. While ACP uses this technique to address fairness, we introduce a similar idea to ensure that the queue is fully emptied once each congestion epoch (as far as possible by the reduction of one single flow).

E-TCP

[57] proposes a congestion controller that also aims for small buffer support and is derived from the analysis of the evolution model of the Internet; therefore called E-TCP. To address scalability E-TCP defines a minimum feedback rate. The congestion control is realized by a multiplicative increase similar to Scalable TCP, but with an increase rate of $\alpha = 0.04$

$$cwnd = cwnd + \frac{1}{25} \tag{2.63}$$

and the respective decrease function to not go below a given feedback rate $p_0 = 0.01$

$$cwnd = cwnd - \frac{cwnd}{25 \cdot (2 + 0.01 \cdot cwnd)} \tag{2.64}$$

While E-TCP resolves the scalability problem by enforcing a minimum fixed feedback rate, the approach taken by TCP SIAD is completely different as it additionally allows for controlling the aggressiveness, e.g., as needed for congestion policing.

MulTCP

MulTCP [38] implements weighted proportional fairness by letting the user set a weight for a certain connection. Thereby MulTCP allows the user to open just one connection that allocates the same share of the available capacity as otherwise N simultaneous connections would do. This is implemented by adjusting the aggressiveness of TCP Reno by N:

$$cwnd \leftarrow cwnd + \frac{N}{cwnd} \text{ [per ACK]} \tag{2.65}$$

$$cwnd \leftarrow \frac{N - 0.5}{N} cwnd \text{ [on loss]} \tag{2.66}$$

Note that MulTCP decreases the sending rate for each lost packet as it assumes exactly one loss for each of the virtual N TCP flows as congestion notification.

MulTCP was proposed in combination with pricing based on the respective weight and duration of a connection to maximize the total utility of the network. As we envision a per-user congestion policing in the Internet, we also aim for a configuration possibility. However we do not want to relate the aggressiveness directly to a corresponding number of TCP NewReno flows.

2.2.3 Summary and Discussion

Scalable TCP provides a feedback independent of the available bandwidth, as required by our design goals, but on the cost of a large overshoot due to the multiplicative increase behavior. Therefore, we do not follow the MIMD design scheme due to potentially high loss rates as well as the convergence problem of MIMD. Instead, we oriented our design on some mechanisms used by TCP BIC/Cubic as well as the decrease behavior of H-TCP.

H-TCP re-calculates the decrease factor for each congestion event based on RTT measurements and thereby an estimate of the number of packet in the queue. This estimation was first introduced by TCP Vegas and is used by various delay-based as well as hybrid schemes. By using the same decrease but a different increase pattern than H-TCP, we do not need to restrict the decrease range. Therefore, we cannot only cope with smaller buffers but also minimize standing queues in case of large buffers.

Additionally, we aim to empty the queue once in each congestion epoch. That also means that TCP SIAD should be able to measure the base/minimum RTT after every decrease. Therefore, we do not further address the problem of base RTT updates, e.g., due to an increased end-to-end delay after a routing change, as proposed by LEDBAT.

By calculating the increase such that a constant feedback rate is achieved, we inherently need to use the maximum congestion window $cwnd_{max}$ before a window reduction as a target value, similar as TCP BIC and TCP Cubic. We use linear increase below this target and an exponential increase above the target to quickly grab new capacity, similar to TCP BIC.

Similar to TCP Illinois and APC, we need to calculate both the increase rate and the decrease factor dynamically. However the design goals and as such the approach taken are very different. While TCP Illinois and APC recalculate the increase rate and decrease factor once per RTT or at least several times in each congestion epoch, we aim to only update based on a congestion notification. Further, TCP Illinois and APC aim to speed up the increase during one congestion epoch while staying TCP-friendliness in congestion situations. In contrast we aim to permanently adapt to certain network situations, mainly the available bandwidth and buffer size. Subsequently if the network conditions are not changing we maintain the congestion control in steady state such that the link is always utilized.

We explicitly would like to design an algorithm that introduces a configuration interface to control the aggressiveness, similar to MulTCP, as this is required for the deployment of congestion policing in the future Internet. Other than MulTCP, we do not base our algorithm design on TCP NewReno. Our design goal for a fixed feedback rate instead leads to a novel and completely different approach. Moreover, we explicitly do not target TCP-friendliness. Therefore, we did not pick up the idea of, e.g., TCP Africa and TCP Fusion to implement a slow mode when the sending rate comes closer to the expected saturation point.

DCTCP is explicitly target for low latency support, similar to our design goals. DCTCP adapts the decrease factor based on the number of ECN marks seen within one RTT. Therefore, DCTCP requires not only changes to the congestion control algorithm of the sender but also need a more accurate ECN feedback from the receiver and a specific configuration of at least the bottleneck network node. This means it can currently only be applied in a controlled environment like a data center and not used in the Internet. As DCTCP still seems to introduce an interesting approach, we contribute to standardization of a more accurate ECN feedback protocol in the Internet Engineering Task Force (IETF) [88, 26] as a first step for the deployment of DCTCP-like congestion control in the future Internet.

2.3 Network-supported Congestion Avoidance

While congestion control aims to recover from congestion, congestion avoidance "allows the network to operate in the region of low delay and high throughput" [72]. Therefore, a congestion avoidance scheme consists of two parts: a feedback mechanism in the network performing congestion detection, feedback filtering, and feedback selection plus a control mechanism in the end host implementing signal filtering, a decision function, and the increase/decrease algorithm [72]. To achieve efficiency as well as fairness both mechanisms have to work appropriately together. A large set of increase/decrease algorithms have already been discussed above. This section therefore focuses on network-based feedback mechanisms.

Note that a control loop where a rate controller in the bottleneck signals the respective feedback information to the end system that consequently adapts its sending rate can also be regarded by a control theoretical approach. [32] investigates the closed-loop congestion control problem in packet networks assuming per connection-queuing and feedback information on the current queue occupancy. This model could be applied to, e.g., Multi-Protocol Label Switching (MPLS) networks, if resource demands are known and the underlying traffic is not congestion controlled itself, but is not applicable to pure Internet Protocol (IP) network with binary feedback only.

Feedback could be provided by, e.g., signaling messages between the router and the source, end-to-end probes by the source, or in-band information in forward or backward direction of the TCP connection. In the Internet usually congestion is implicitly signaled by dropping packets in the bottleneck router queue due to overflow or by using AQM or explicitly by marking packets in the IP header. Further, First In First Out (FIFO)-based queue management is most often used. With a simple *DropTail* queue, packets get dropped passively as the queue overflows.

In contrast, AQM aims to avoid queue overflows by signaling congestion early, as further explained in the next section. As TCP congestion control operates most often based on loss as an implicit congestion signal, various AQM schemes have been proposed to drop a small number of packets before a router queue overflows. Additionally, instead of dropping packets an explicit signal could be used. Based on the proposal of an one bit feedback in packets in forward direction of [121], ECN [120] is standardized for TCP. ECN can be negotiated in the TCP handshake and subsequently used during the rest of the TCP connection, as further explained in Section 2.3.2.

2.3.1 Active Queue Management (AQM)

AQM signals congestion before the queue overflows. By this, spare capacity in the queue is provided that can be used by small traffic bursts and, as such, burst losses can be avoided. Additionally, AQM avoids global synchronization of competing flows [50]. Therefore, if not all flows reduce their congestion window at the same time, the number of packets that are needed in the queue to keep the link utilization high can be reduced. Similar to congestion control, various algorithm for AQM have been proposed such as RED [50] including various modification as Adaptive RED [45], Dynamic RED [16], and Stabilized RED [112] as well as BLUE [40], PURPLE [116], Random Exponential Marking (REM) [15], Adaptive Virtualized Queue (AVQ) [90], and more recently Controlling Delay (CoDel) [109] and PIE [114] that both are especially targeted to control the average queuing delay.

We explain the RED algorithm in detail to exemplarily illustrate the operation of AQM. RED probabilistically decides about the dropping or marking of arriving packets based on the average queue occupancy. The average queue occupancy is calculated based on an Exponential Weighted Moving Average (EWMA) of the instantaneous queue length q with the weighting factor w, thereby realizing a low pass filter to allow for short bursts.

$$\tilde{q} \leftarrow (1-w)\cdot\tilde{q} + w\cdot q \tag{2.67}$$

When the average queue occupancy is below a minimum threshold (Min_Thresh), no arriving packets are marked. Above this threshold arriving packets are marked with a certain probability linearly reaching up to a maximum marking probability (*Max_Prob*) at the maximum threshold (Max_Thresh) and a probability of 1 above, as displayed in Figure 2.5. As the right parameterization of RED depends on the network configuration such as the bottleneck link rate as well as the traffic load, number of flows, and sending behavior of each single flow, it is usually impossible to configure RED optimally. Therefore, a default configuration has been proposed that is claimed to be suitable for many Internet scenarios by setting w to 0.002, *Max_Prob* to 0.1, Min_Thresh to $\frac{1}{4}$ and Max_Thresh to $\frac{3}{4}$ of the queue size [42].

In the recent years new AQM schemes have been proposed to minimize queuing delays and thereby better support services that require low latency. Further, an IETF AQM working group currently exists that works on algorithms for queue management as well as recommendations for their usage and evaluation. This working group has been founded based on input from the

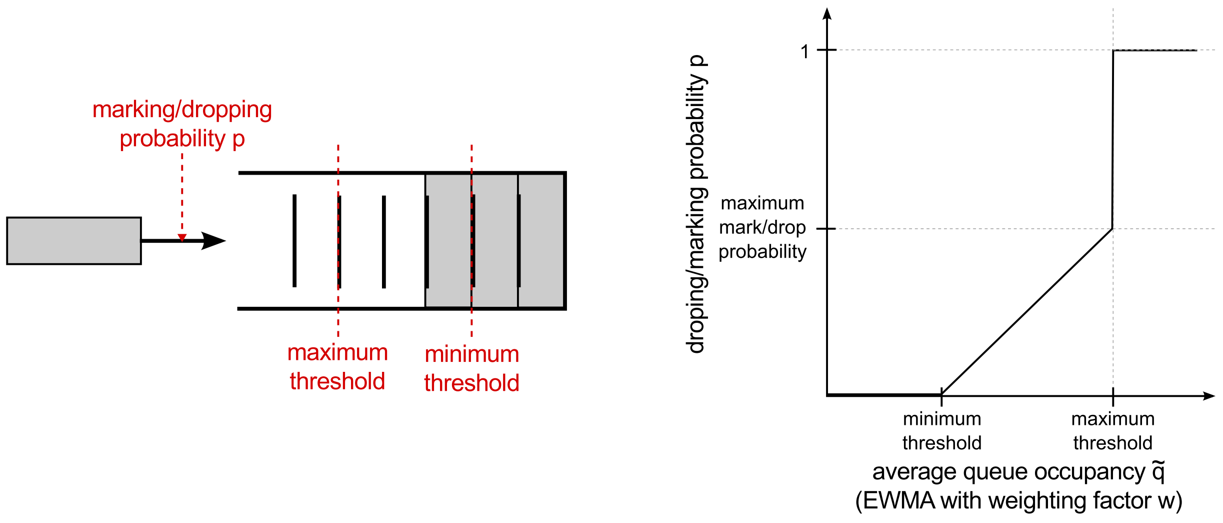

marking/dropping probability p
maximum threshold
minimum threshold
droping/marking probability p
1
maximum mark/drop probability
minimum threshold
maximum threshold
average queue occupancy q̃
(EWMA with weighting factor w)

Figure 2.5: Random Early Detection (RED) principle and drop probability.

Bufferbloat project [52, 2] that aims to reduce latency in the Internet originating from large buffers in the network or end-hosts. While large buffers alone are not the problem, flows using loss-based congestion control always fills these buffers and thereby maximize the queuing delay.

Especially two new AQM schemes, namely Controlled Delay (CoDel) [109] and Proportional Integral Controller Enhanced (PIE) [114], have recently been proposed that try to keep the queuing delay below a desired value target value.

CoDel [109] monitors the delay for each packet in the queue by adding a time stamp at enqueue and evaluating the queuing delay at dequeue. If the minimum queuing delay of all packets within a certain time interval of length *interval* is larger than the given target delay *target*, CoDel assumes persistent congestion and drops a packet. As long as the queue delay is above *target*, CoDel stays in *dropping* state and calculates the time for the next drop *next_drop* based on the time of the last drop *t* and the number of drops performed so far *count* (within the last *interval* milliseconds) using the following control law

$$drop\_next = t + \frac{interval}{\sqrt{count}}. \tag{2.68}$$

interval is proposed to be set by default to 100 ms and *target* to a fixed value of 5 ms. Note, in the provided pseudo code and Linux implementation, the number of drops *count* of the last $16 \cdot interval$ milliseconds (instead of just one *interval*) is used when the *dropping* state is entered.

PIE [114], however, estimates the current queuing delay based on the (exponentially smoothed) departure rate and the current queue length, where the departure rate itself is estimated based amount of data that has left the queue within a certain measurement interval. On enqueue PIE drops the packet with a drop probability p that is periodically calculated based on the queuing delay estimate *est_del* as well as the target value *target_del* by

$$p = p + \alpha \cdot (est\_del - target\_del) + \beta \cdot (est\_del - est\_del\_old). \tag{2.69}$$

where α is 0.25 Hz and β is 2.5 Hz (as these multiplications can be implemented by shift operations). Therefore, p depends on the actual deviation from the target value (multiplied by

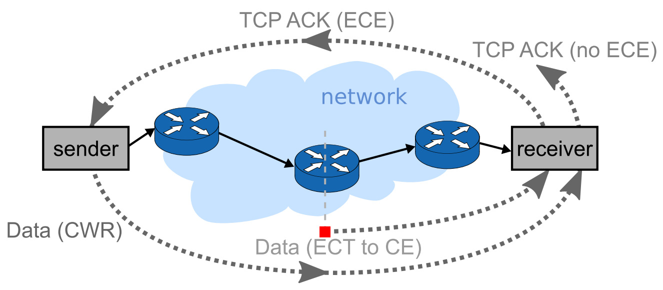


Figure 2.6: The ECN feedback scheme.

α) and a trend of the delay development over time (multiplied by β). It is recommended to update the drop probability as well as the queue delay estimate every 30 ms where the previous delay estimate is stored in *est_del_old*. The target value *target_del* is by default set to 20 ms. Moreover, PIE implements a burst tolerance of 100 ms before starting dropping.

In Section 4 we evaluate the interaction of TCP SIAD and different AQM schemes based on RED, CoDel and PIE. While RED is already implemented in most available network routers, CoDel and PIE are new schemes that are expected to reach deployment in future networks.

2.3.2 Explicit Congestion Notification (ECN)

ECN [120] is a TCP/IP mechanism in the Internet that allows network nodes to mark packets instead of (early) dropping them. In the TCP handshake an ECN sender can negotiate for ECN support with the receiver, as specified in RFC3168 [120]. If the receiver is ECN-capable, a sender can mark each IP packet as ECN-capable transport (ECT) by setting one of the two IP ECN bits. Setting one or the other bit leads to two flags, the ECT(0) or ECT(1), which are used with ECN-Nonce to provide an integrity mechanism. When an IP packet is marked as ECT(0) or ECT(1), a network node can mark this packet as Congestion Experienced (CE) by also setting the other IP bit. A network node uses an AQM mechanism like, e.g., RED to mark packets before the queue overflows.

When an ECN receiver sees a CE flag, it sets the ECN-Echo bit in the TCP header of the ACK. The ECE bit is then set in all subsequent ACKs until a packet with the Congestion Window Reduced (CWR) bit set in the TCP header is received from the original sender to acknowledge the ECE. Therefore, for one received CE a whole RTT of ECE marked ACKs is sent. During this period additional received CE marks have no influence and cannot be recognized by the sender. The sender sets the CWR bit on reception of the first ECE bit after reducing the sending rate/congestion window as shown in Figure 2.6. The sender does not reduce the congestion window more than once per RTT. ECN-Nonce [137] uses one more TCP bit to signal a one-bit Nonce Sum (NS), which counts the number of ECT(1) flags received. The sender can use this information for integrity checking and thus detect when congestion information was not fed back correctly.

Currently within the IETF TCP Maintenance and Minor Extensions (tcpm) working group a more accurate ECN feedback is under standardization that aims to not only feed back one signal per RTT but a more fine-grained congestion feedback signal [88, 26]. This information can be used by future congestion control schemes such as DCTCP.

Further, to better support low latency requirements immediate ECN feedback without any smoothing delay would be needed [27]. E.g. RED calculates the drop probability based on the smoothed average queue length to allow for short traffic bursts that induce congestion for time period smaller than the feedback delay. This helps to keep link utilization high in case of short bursts as congestion control can anyway only react on a RTT basis. Otherwise in case of persistent congestion the feedback signal is delayed and therefore induces congestion longer than necessary. Moreover, as today ECN is defined as a “drop equivalent” and therefore provides only small performance gains in networks optimized for low loss rates, it has consequently not seen wide deployment. With a change in semantics, ECN could be used as an enabler for new low latency services when implementing a different response to congestion in future congestion control schemes.

2.3.3 Services Differentiation to Support Low Latency

While all flows traversing the same bottleneck queue experience the same queuing delay and loss rates, different QoS can be implemented by separating flows or groups or flows into different queues. In this case an additional scheduling mechanisms is needed to decide which queue should be served in which order. One simple example for a scheduling algorithm is Round Robin serving all queues in circular order.

Several proposals exist that aim provide better low latency support by offering a separate low latency service next to the currently available low-loss Best-Effort service in the Internet [41, 117]. These proposals are based on the assumption that most applications require either low latency or low loss. Therefore, each of the two networking services provide only benefits for a certain application class (in contrast to absolute priorities as used by Differentiated Services (DiffServ) [108]). While the current Best-Effort service can provide low loss rates but potentially imposes high (queuing) delays, a low-latency service focuses on small additional queuing delays but potentially impose high loss rates. Therefore, a sending entity can decide between the trade-off of these to QoS parameters and mark each data packet respectively. Then the network node before the bottleneck link will treat both classes differently.

There are several proposals for such a service differentiation. Alternative Best Effort (ABE) [66], Best Effort Differentiated Services (BEDS) [41], and Rate-Delay network services (RD) [117] are three examples. The latter two assort all packets into two different queues, one for each service. Both queues are configured differently in terms of maximum queue size and AQM parameters. While in the proposed mechanisms the queue set-up can be quite simple, the challenges lie in scheduling mechanism to dequeue the packets. Often high additional complexity is introduced to address how to avoid unfairness between the two services and how to handle unresponsive flows.

2.3.4 Summary and Discussion

Congestion control in the end host always interacts with the congestion avoidance in the network. If the mechanisms that are used in the network would be known (to the end host), congestion control could be further optimized. However, in our case where we aim to deploy a end-host-based mechanism in the Internet, the developed schemes must be able to cope with various congestion avoidance schemes in the network and the respective feedback mechanisms. In the evaluation we will therefore investigate TCP SIAD's behavior while using different AQM schemes and parameterizations in the bottleneck queue.

2.4 TCP Congestion Control Implementation in Linux

In Linux TCP congestion control is implemented in the network stack of the kernel [128, 144, 124] providing five congestion states [130]:

TCP_CA_Open A connection is in TCP_CA_Open state if no congestion is sensed.

TCP_CA_CWR When local congestion is signaled by the network interface device driver or congestion feedback is received based on ECN signaling, TCP_CA_CWR is entered.

TCP_CA_Disorder When the first duplicated ACK is received, TCP_CA_Disorder is entered. If the next ACK acknowledges new data, TCP_CA_Disorder is left and we go back to TCP_CA_Open.

TCP_CA_Recovery Else if a second duplicated ACK is received, TCP_CA_Recovery is entered. When entering recovery state, a new Slow Start threshold $ssthresh$ is requested from the congestion control algorithm. During recovery the congestion window is (stepwise) reduced to $ssthresh$.

TCP_CA_Loss The TCP_CA_Loss state is entered when congestion is detected due to an RTO. In this case all data in flight are assumed to be lost and the congestion window is reset to a minimum value.

Any congestion state is left when the last byte that was sent before entering this state is acknowledged by the ACK number; thus, one RTT after the reception of the congestion signal.

New congestion control schemes can be implemented as a kernel module [118]. Modules are parts of the kernel code that can dynamically be loaded during run time without rebooting the system. Components like drivers which are needed depending on the hardware and potentially should be updated from time to time are often implementable as kernel modules as well. Since 2005 it is possible to implement congestion avoidance algorithm as Linux kernel modules using the following structure [36]. This *struct* is defined in the Linux kernel in */include/net/tcp.h*, shown as in version 3.5.7 which we used for our implementation.

```
struct tcp_congestion_ops {
  struct list_head  list;
  unsigned long flags;

  /* initialize private data (optional) */
  void (*init)(struct sock *sk);
  /* cleanup private data  (optional) */
  void (*release)(struct sock *sk);

  /* return slow start threshold (required) */
  u32 (*ssthresh)(struct sock *sk);
  /* lower bound for congestion window (optional) */
  u32 (*min_cwnd)(const struct sock *sk);
  /* do new cwnd calculation (required) */
  void (*cong_avoid)(struct sock *sk, u32 ack, u32 in_flight);
  /* call before changing ca_state (optional) */
  void (*set_state)(struct sock *sk, u8 new_state);
  /* call when cwnd event occurs (optional) */
  void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
  /* new value of cwnd after loss (optional) */
  u32  (*undo_cwnd)(struct sock *sk);
  /* hook for packet ack accounting (optional) */
  void (*pkts_acked)(struct sock *sk, u32 num_acked,
    s32 rtt_us);
  /* get info for inet_diag (optional) */
  void (*get_info)(struct sock *sk, u32 ext,
    struct sk_buff *skb);

  char      name[TCP_CA_NAME_MAX];
  struct module   *owner;
};
```

If all mandatory methods are implemented and if a new scheme is configured to be used, the respective congestion control code is called during TCP processing, as explained next. Thereby information on the current TCP state is provided to the congestion control module that can be used within the algorithm.

- `init()` is called when a new socket is created and the connection was successfully established after TCP's 3-way handshake. It can also be called when the connection was idle for a long time.
- `release()` is called before a socket is closed to, e.g, release any memory that was allocated beyond the reserved congestion control space.
- `ssthresh()` is called once when a congestion state is entered. This method is mandatory and implements the decrease behavior.
- `min_cwnd()` is called to request the minimal allowed congestion window.
- `set_state()` is called before any congestion state is changed.
- `cong_avoid()` is called for each ACK in a non-congestion state. This method is mandatory and implements the increase behavior.

- `cwnd_event()` is called when any of the below listed congestion events occurs. The input parameter `ev` can be used to identify which event had happened.
- `undo_cwnd()` is called when the congestion window was reduced previously due to wrong congestion detection based on a transient situation, e.g. reodering. A larger congestion window should be returned to restore the situation before the window reduction.
- `pkts_acked()` is called for each ACK.
- `get_info()` is a monitoring hook.

Note that all methods have a pointer to the socket memory as input parameter. This pointer can be used to retrieve the TCP socket memory that stores the TCP connection state such as the current congestion window `snd_cwnd`, the Slow Start threshold `snd_ssthresh` and a counter `snd_cwnd_cnt` to increase the congestion window. Moreover, there are 64 Bytes of memory space allocated that can be used by the congestion control module to store own state by declaring a struct which has the same name than the congestion control module.

Seven events are defined, as shown below, that are signaled in `cwnd_event()`.

```
/* Events passed to congestion control interface */
enum tcp_ca_event {
  CA_EVENT_TX_START,    /* first transmit
        * when no packets in flight */
  CA_EVENT_CWND_RESTART, /* congestion window restart */
  CA_EVENT_COMPLETE_CWR, /* end of congestion recovery */
  CA_EVENT_FRTO,    /* fast recovery timeout */
  CA_EVENT_LOSS,    /* loss timeout */
  CA_EVENT_FAST_ACK,    /* in sequence ack */
  CA_EVENT_SLOW_ACK,    /* other ack */
};
```

This background information is needed to understand various implementation details of TCP SIAD as described later on in Section 3.3 as well as specific effects of this implementation that can be seen in the evaluation. Background information on approaches for evaluating congestion control are explained as discussed (regarding the applicability for TCP SIAD) in the next section.

2.5 TCP Congestion Control Performance Evaluation

Due to the complexity and rapid development of protocol and control mechanisms in TCP, abstract models of the TCP behavior often do not reflect the characteristics of real systems well enough. In addition, when evaluating TCP mechanisms like congestion control, it is important to use recent TCP implementations, because changes as, e.g., using an initial TCP congestion window of 10 packets [35]or PRR [103] strongly influences the congestion control behavior. Therefore, our evaluation is based on IKR SimLib [3], a freely available library for event-driven (network) simulation written in Java, together with the VMSimInt extension [146], an approach for packet-level simulations which is based on the integration of Virtual Machines (VMs) into the simulation environment. In the following section the IKR SimLib with VMSimInt extension is briefly described.

Two IETF working groups, namely the AQM and the RTP Media Congestion Avoidance Techniques (rmcat) working group, currently define evaluation criteria and guidelines as well as evaluation test cases for AQM and, respectively, congestion control for real-time media. This is still work in progress and is currently focused on simple initial scenarios where the available bandwidth, the end-to-end delay or the number of competing flows is varied or changed within a simulation run [89, 136, 127]. While congestion is often self-induced by congestion control, [150] additionally performs evaluations based on an artificial loss model. More complicated models of rate and delay variations, e.g., for evaluating performance over 3G wireless link [33] can be found in the literature. Section 2.5.2 introduces the reader to evaluation scenarios and tests cases as proposed by the TCP Evaluation Suite [13, 63]. The TCP Evaluation Suite is target for initial evaluation of new TCP extensions as congestion control in comparison with existing schemes as well as exploring if a new proposal is safe for further experimentation on the Internet. Therefore, this is most relevant for the evaluation of TCP SIAD.

Finally, we discuss goals and metrics for congestion control evaluation as they can be found in the literature on congestion control analysis as well as are used by existing proposals for evaluation.

2.5.1 Event-driven Network Simulation Integrating Virtual Machines

Network or transport protocols as well as extensions to existing protocols such as TCP congestion control are often evaluated by simulation. Other than measurements in testbeds or emulation, simulation provides a controlled environment without any external influence and consequently delivers fully reproducible results. This allows a fair comparison to existing approaches and a detailed evaluation of the microscopic behavior of the proposed algorithm. The IKR SimLib [3] with VMSimInt extension [146] provides a network simulation tool that integrates real kernel code in a simulated network. Therefore, in this work the IKR SImLib is used as it makes it possible to use the TCP SIAD Linux congestion control kernel module implementation in simulations.

Compared to emulation-based approaches which connect real computer systems or several VMs on one computer to a simulated network, VMSimInt provides full isolation from the host system. The time perceived by a kernel running in a VM is controlled by the simulation framework and completely independent of the host time. As the simulated time does not proceed while the VM's processor is operating, the performance of the host computer or the virtualization tool does not influence the simulation results.

The VMSimInt integrates QEMU [6] virtual machines in a simulated network. Therefore, the VMSimInt architecture consist of the components as shown in Figure 2.7, namely a QEMU adapter, the QEMU itself which has been extended to work within the simulation and of course the simulation tool. The QEMU adapter wraps the QEMU process for use in the simulation program and provides the respective interfaces to, e.g., exchange Ethernet frames or send other control commands. Further, it is responsible for the synchronization of the QEMU with the

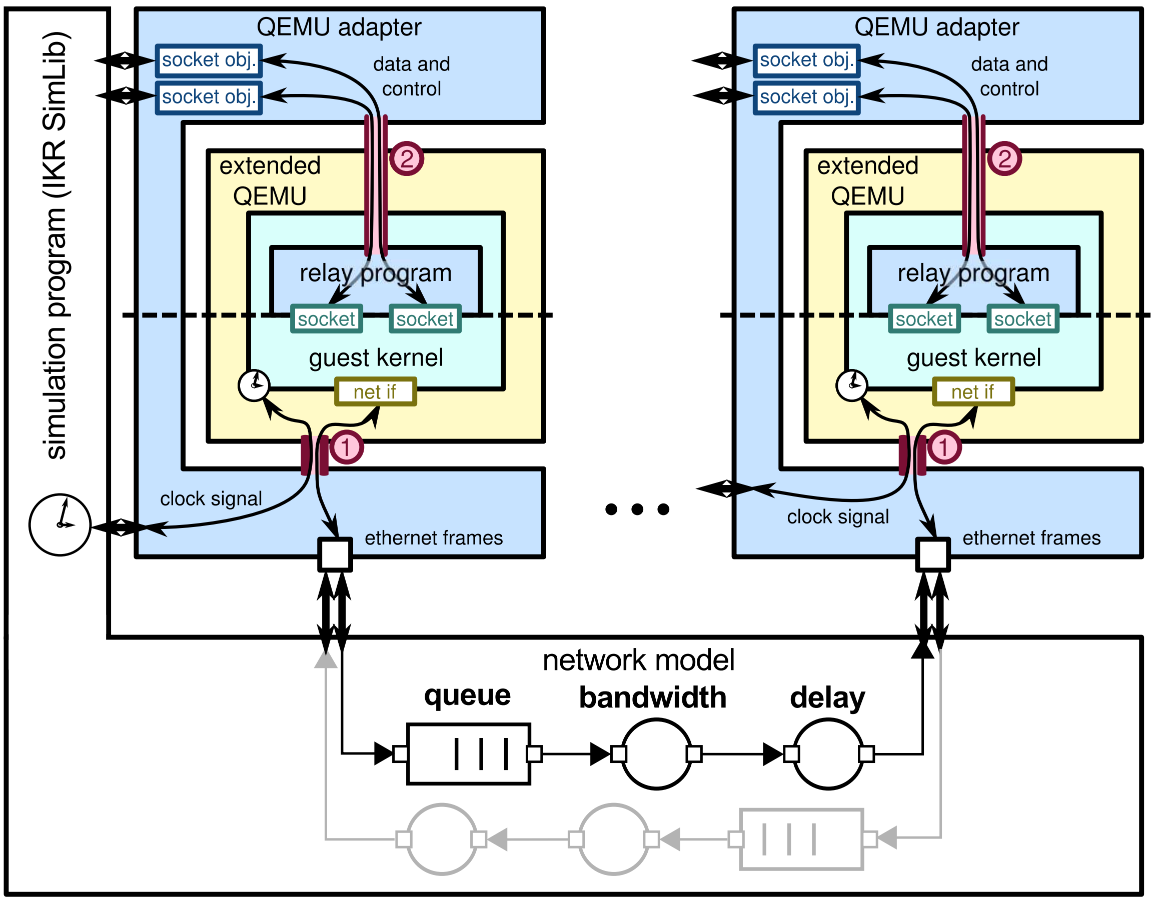


Figure 2.7: Architecture of VMSimInt.

simulation clock of the calendar in the event-driven simulation program. The extended QEMU runs the virtual machines and thereby intercepts all input and output processing and redirects everything to the QEMU adapter (instead of to the host system as QEMU would normally do). Each virtual machine executes unmodified operating system code in a separate process. Moreover, a helper relay program inside each VM allows the simulation environment to access the socket Application Programming Interface (API) of a VM and thereby to generate TCP traffic or configure TCP sockets (by setting `sysctl` or socket options). During simulation, packets sent by any VM are not forwarded to the host's network but are handled and forwarded within the simulated network, which is modeled in the simulation environment. This allows the use of the real Linux network stack in simulation generating packets with dummy payload (provided by the simulator itself) and real TCP/IP and Ethernet headers added by the Linux kernel. The respective QEMU adapter interface can be used to open greedy connections as well as on/off traffic generated by a static traffic model (with artificial Inter Arrival Time (IAT) and flow size distributions) or based on trace files.

2.5.2 TCP Evaluation Suite

The TCP Evaluation Suite [13, 63] proposes an initial set of scenarios for evaluation of TCP extensions in simulation or testbeds. These scenarios and respective tests are targeted to provide a basis for initial evaluation and thereby easy comparison to other schemes as well as to provide good confidence that the proposed TCP extension is safe for experimentation in the Internet. Therefore, most of the described tests use traffic generated based on measured Internet traffic traces as further explained in the next section. Only few tests examine simple scenarios with a limited number of greedy sources. But these kinds of tests are also needed to fully evaluate the microscopic behavior of a proposed algorithm and thereby cover various extreme cases. Note that this is not the intention of the proposed TCP Evaluation Suite, so we evaluate a large set of additional scenarios in chapter 4.

2.5.2.1 Traffic Generation

In [13] it is proposed to model sessions of individual users with Poisson-distributed inter-arrival times. One session can consist of one or multiple greedy or non-greedy TCP flows separated by "think" times where both, think times and burst sizes, have heavy-tailed distributions which potentially can be determined based on empirical studies. This approach allows modeling of interactive as well as greedy traffic of different application types but does not represent any user reaction if, e.g., heavy congestion occurs and a user might cancel a transmission manually.

Tmix [145] is a well-known TCP traffic generator (used in ns-2/3 [4, 5] or the GENI testbed [140]) that provides a connection vector with request size (of the request message that is sent to the server), response size (of the reply message from the server), and think time between the two and therefore can implement the above specified behavior. For Tmix there exist a set of nine example traces [111] applicable for the dumbbell topology proposed by the TCP evaluation suite and further described below. It is proposed to scale and shuffle (within certain bins) the given arrival times to vary the load in different scenarios and therefore to be able to better cope with the non-stationarity of these traces.

2.5.2.2 Topologies and Tests

Most tests are run based on a dumbbell topology, as further described next. Further, a parking lot topology, as described afterwards and shown in Figure 2.9, is proposed for the evaluation of scenarios with multiple bottlenecks. Finally, a set of additional tests are described to access transient effects such as convergence of two greedy flows.

Dumbbell Model

The dumbbell scenario described in the TCP Evaluation suite consists of nine hosts and two routers connected by a central link that usually is the bottleneck as displayed in Figure 2.8. The access links have different delays to achieve a realistic distribution of flows with different RTTs (and an average RTT of 100 ms) on the bottleneck link. The proposed delay values are shown in Figure 2.8.

Based on this dumbbell topology a set of basic scenarios is defined in [13, 63] as given in Table 2.1. For each scenarios three load levels for the bottleneck link should be tested, namely moderate (60%), high (85%), and overload (110%) [63], based on scaling of traffic traces as

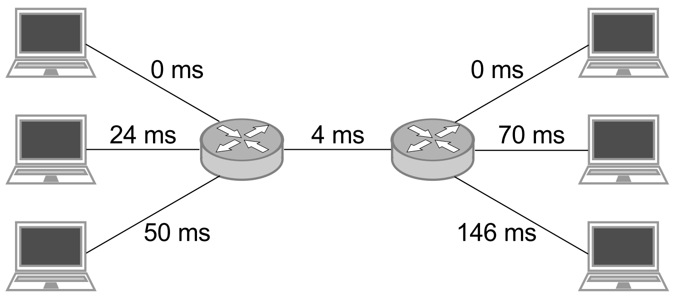


Figure 2.8: Dumbbell topology.

introduced above. Note that the overload case is not steady as the number of active flow continuously increases over the simulation time and therefore the congestion level increases as well and more and more congestion will be seen. All flows in the tests use the TCP extension under evaluation. The buffer size (BS) of the DropTail bottleneck queue is most often configured to induce 100 ms of queuing delay or as denoted below. For the access link scenario also different AQM mechanisms such as RED should be used.

All simulations run for at least 100 s but the statistical evaluation does not include a warm up phase of a given length depending on the traffic characteristics and load. It is proposed to collect statistics on the aggregated link utilization, the average packet drop rate, and the average queuing delay. Further, metrics based on end-to-end or per-flow measurements as well as for the evaluation of stability properties could be regarded.

Moreover, the access link scenario should be used with 85% offered load for evaluation of the delay-throughput trade-off that many congestion control schemes have. Therefore, tests with buffer sizes of 10%, 20%, 50%, 100%, and 200% of the base BDP for a 100 ms base RTT flow are envisioned. The average delay and throughput can be illustrated in a delay/throughput graph.

Table 2.1: Parameter value of basic simulation scenarios as given in [63] to set the bandwidth (BW), the One-Way-Delay (OWD), and base RTT (BS) for the access links and the central link.

Scenario	Link	BW	OWD	BS
Access Link	central	100 Mbps	2 ms	100 ms
	access	100 Mbps	(see Fig.)	
Data Center	central	1 Gbps	0	10 ms
	access (1,2,4)	1 Gbps	10 μs	
	access (3,5,6)	1 Gbps	100 μs	
Trans-Oceanic Link	central	1 Gbps	65 ms	100 ms
	access	1 Gbps	(see Fig.)	
Geostationary Satellite	central (downlink)	40 Mbps	300 ms	100 ms
	central (uplink)	4 Mbps	300 ms	1000 ms
	access	100 Mbps	(see Fig.)	
Wireless LAN	central	100 Mbps	2 ms	
	access (right)	~6 Mbps	by CSMA model	
Dial-up Link	central	64 kbps	5 ms	1250 ms

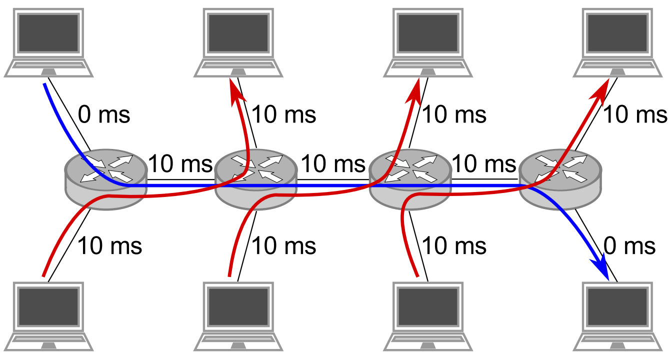


Figure 2.9: Parking lot topology.

To evaluate the impact of a new TCP extension on existing TCP traffic, the access link scenario, but with rates of 56 kbps, 10 Mbps, and 1 Gbps, is used. In this case, each node will send each flow in the respective trace file twice (at the same time). One of the flows is using TCP NewReno with SACK (and without ECN), and the other one is using the TCP extension under evaluation. In this scenario two offered loads of 50% and 100% should be tested. The throughput and loss ratios between both traffic classes should be measured.

Multiple Bottlenecks Model

A "parking lot" topology with three hops is proposed to evaluate the influence of one flow traversing multiple bottlenecks as shown in Figure 2.9. In this scenario all bottleneck links have a transmission rate of 100 Mbps and the access links a rate of 1 Gbps. All flows should have the same base RTT of 30 ms which, e.g., can be achieved by a symmetric scenario with multiple links that induce 10 ms delay as in Figure 2.9 or an asymmetric scenario with, e.g., only a few links with 30 ms delay.

Traffic should be generated based on the traffic traces resulting in test with 30%, 40%, and 50% offered load per source.

The ratio of the average throughput of single bottleneck flows to the average throughput of multiple bottleneck flows as well as the packet drop rates on the bottleneck links should be evaluated.

Transient Effects

The following in [13, 63] proposed three tests of the TCP evaluation suite evaluate how quickly existing flows can release bandwidth to new starting flows as well as how quickly existing flows can grab newly available bandwidth.

The first test assumes a topology with two sending and two receiving nodes as shown in Figure 2.10. The first sender has a greedy TCP flow while the other sender sends Constant Bit Rate (CBR) traffic. The bottleneck link has a rate of 100 Mbps and the bottleneck queue a size of 120% of the base BDP of an 100 ms flow. The cases where the CBR traffic starts sending

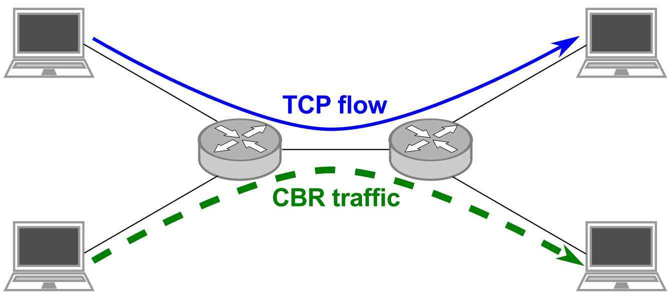


Figure 2.10: Topology with two flows (TCP and CBR).

with 75 Mbps and stops again as well as where the CBR traffic increases its rate in 30 steps of 2.5 Mbps from 0 to 75 Mbps at 1 s intervals should be tested.

In the case of the stopping CBR flow, the time of the greedy TCP flow to reach 60%, 80%, and 90% of the available bandwidth should be measured. Otherwise with starting CBR traffic the influence on the drop rate of the CBR cross traffic is of interest.

The second test evaluates two greedy flows between two senders and two receivers (as above) but with 10% cross traffic load generated by trace-based connections between three additional senders and receivers as in the Access Link scenario above but with a bottleneck bandwidth of 56 kbps, 10 Mbps, and 1 Gbps. The second greedy flows starts later when the first flow is already in equilibrium. Either both flows have a base RTT of 80 ms or one of the flows has an base RTT of 30 ms while the other has 120 ms. In this test the second flow has Slow Start disabled by setting the Slow Start threshold to the initial window.

The time until $1500 \cdot 10^n$ bytes arrive should be monitored to evaluate how quickly an existing flow can release capacity for different values of n.

The third test uses the same scenario with two greedy flow and cross traffic as above but with different buffer sizes between 25% and 200% of the base BDP of an 100 ms flow. While the first flow starts at the beginning of the test, the second flow starts randomly within the first 10% of the test duration. To evaluate intra-protocol fairness, both flows have the same RTT of either 10 ms, 20 ms, 40 ms, 80 ms, or 160 ms. Or the first flows has an RTT of 160 ms while the second has an RTT from the list above to evaluate RTT fairness.

The average throughput values of both flows should be compared.

2.5.3 Goals and Metrics

To evaluate and compare proposed algorithms certain goals need to be defined, as well as tests and metrics to verify if a goal could be reached. In this section, we discuss goals and metrics that are generally used to evaluate congestion control algorithms in literature. We do not consider any user-based metrics like, e.g., the QoE of a certain application as we only use very simple traffic models, as described above, which do not fully reflect a realistic application behavior that, e.g., could be mapped to an utility function.

Efficiency

One of the goals of congestion control is using the available network resources efficiently. This includes that congestion control also aims to avoid the state of overload as this decreases performance as well. Therefore, there is a desired load level X_{goal} which operates the network "at the knee" [34] at high utilization but avoids overload. Efficiency can be reached if the total resource allocation $X(t) = \sum x_i(t)$ of all flows i is close to X_{goal}. Overload ($X(t) > X_{goal}$) and underload ($X(t) < X_{goal}$) should both be avoided. Note that efficiency can be reached in different operating points, e.g., when only one user allocates all resources the network is utilized efficiently. Therefore, often *global efficiency* is desired where the performance (e.g. maximum throughput and minimum loss) of the total system is the highest [72]. A certain distribution of the resources is the desired goal when regarding fairness, which is discussed next.

To measure the efficiency of the utilization of the available link capacity, *throughput* is usually regarded. Throughput can be measured on a per-flow basis or over an aggregated number of flows, e.g. at a certain observation point. Throughput is defined as the amount of data that was transmitted over a certain time. The maximum throughput that can be reached is limited by the capacity that the smallest link on a path, the bottleneck, can transmit. Additionally, in TCP we distinguish between *goodput* and throughput [44]. While throughput simply considers each byte that was transmitted over a link, goodput considers only unique data (no retransmissions). As TCP retransmits lost data, goodput is lower than (or equal to) throughput. Normally the goodput is harder to measure, as some state and protocol information might be needed to detect the respective overhead.

The *packet loss rate* can be used as another metric additionally to throughput to assess efficiency. The packet loss rate as well as the ECN marking rate can either be measured at a certain observation point or on a per-flow basis as losses and marks can occur at multiple points on a network path. The packet loss rate can be assessed as the mean of One-way-Packet-Loss [12] sample values that can be either one or zero at a certain point of time. With respect to congestion control often not only the total packet loss/mark rate is of interest but also the *congestion event rate* [44], where all lost packets during one RTT are consider to be part of one congestion event as most congestion control scheme react only once per RTT to congestion. The congestion event rate differs from the packet loss/mark rate when multiple losses (or congestion marks) occur within one RTT.

Depending on the application design, the packet loss rate might not provide a direct relation to the application performance. E.g. when only the total transfer time is important, only additional delays induced by retransmissions might influence the user's experience. Moreover, certain applications can, e.g., handle a high loss rate but might be limited by the number of subsequent losses that occur in a burst. Therefore, to obtain application performance additional One-way Loss Pattern Sample Metrics [85] can be defined. As we do not evaluate our congestion control proposal with respect to the usage with a certain application but with respect to efficient usage of network resources, we only focus on the total packet loss rate and the congestion event rate.

Many congestion control schemes implement a trade-off between (high) throughput and (low) delay. Therefore, the per-packet queuing delay induced by the allocation of buffer space should also be regarded when evaluating efficiency in the usage of network resources. The queuing delay is the time one packet spends in the queue from the point of time at enqueue to the point of time at dequeue or the sum of these queuing delays when multiple bottlenecks are on a network path. Therefore, the queuing delay can be directly derived from the queue size at enqueue if the capacity of the associated link is constant. Of course, it is desirable to minimize the queuing delay. For a certain flow or observation point, the average as well as minimum, maximum, and variation in queuing delay (jitter) might be of interest. In scenarios where the bottleneck is always fully utilized, the variation in queuing delay directly translates in to the size of oscillation by which the sending oscillates between the minimum and maximum, which influences stability and convergence as discussed further below.

Fairness

Fairness assesses the sharing of the network resources between multiple simultaneously competing users. The maximum fairness criterion as defined by Jain's Fairness Index [71] is usually used which is defined as

$$F(x) = \frac{(\sum x_i)^2}{n\sum(x_i)^2} \tag{2.70}$$

where n is the number of competing flows and x_i is the rate of the ith flow. Therefore, the fairness F(x) is maximized if $x_i(t) = x_j(t) \forall i, j$ sharing the same bottleneck, that means when the resources are shared equally.

Fairness can be evaluated in scenarios where all competing flows use the same congestion control algorithm or, respectively, different ones. The first case is often called *intra-protocol fairness* in contrast to *inter-protocol fairness*. Only *TCP-friendliness* is usually regarded in terms of inter-protocol fairness when competing with TCP NewReno-like congestion control.

It should be mentioned that TCP NewReno does not provide an equal resource sharing and therefore is not (self-)fair as defined above when flows with different end-to-end delay are competing with each other. If fairness can also be achieved in scenarios with flows that operate on a different base delay, this is called *RTT fairness*.

Convergence

Congestion control is required to converge to a steady state from any starting state under stable network conditions. The steady state does not need to be a single operation point but can also be an *equilibrium* state where the congestion window oscillates around the optimal point with bounded variations [34]. The steady-state performance can be regarded in terms of link utilization, throughput, RTT as well as capacity sharing/fairness [125]. If a congestion control scheme is not stable, e.g. in terms of load, the total network load slowly either increases to infinity of decreases to zero [72].

Convergence is usually evaluated based on speed and variance. Therefore, the *responsiveness* is defined as the time to reach the equilibrium or steady operating point and the *smoothness* describes the size of the oscillation [34]. Ideally both, the response time and the oscillation size should be minimized. There is a trade-off between responsiveness and smoothness as larger oscillation usually allows for faster adaption to new network conditions [44].

When the convergence is regarded with respect to link load, the smoothness S can be defined by

$$S = \frac{L_{max} - L_{min}}{C} \tag{2.71}$$

where L_{max} and L_{min} are the maximum and respectively minimum load when a certain target has been reached before and C is the link capacity. For n users that use the same AIMD with increase factor α and decrease factor β, S is given by

$$S = \frac{n\alpha - \beta C}{C} = \frac{n\alpha}{c} - \beta \tag{2.72}$$

as derived in [55]. For the responsiveness R to reach the target load, two cases must be considered, as either the network can already be fully loaded (e.g. when a new flow is starting) or the network is currently underloaded (e.g. a flow just stopped). This leads to responsiveness

$$R = \begin{cases} \lceil \frac{C - L(0)}{n\alpha} \rceil & \text{if } L(0) \leq C \\ \lceil \log_\beta \frac{C}{L(0)} \rceil & \text{if } L(0) > C \end{cases} \tag{2.73}$$

where $L(0)$ is the load level at the starting point [55]. Note that the absolute response time depends on the RTT as α is the increase per RTT.

Robustness

A congestion control scheme that is considered for deployment in the Internet must operate robustly in a wide range of scenarios. While convergence is usually shown in static scenarios, robustness must be demonstrated for different offered loads, link speeds, packet sizes, RTTs, congestion levels as well as in challenging environments, e.g. corrupted packets, reordering or variable bandwidth and delay due to route changes or specific lower layer mechanisms [125, 44]. Especially congestion control schemes that rely on delay measurements must be robust to measurement errors due to other protocol mechanisms that can further delay the signal or clock synchronization difficulties. Therefore, the robustness can be assessed by evaluating how many and which scenarios the proposed scheme is able to reach, e.g., the target load.

With respect to robustness also deployability and implementation complexity should be discussed. End-to-end congestion control that should be deployable in the Internet, must be implementable with sender-only changes and work with a binary feedback that is based on loss or ECN, or sampled delay measurements if the receiver supports TSOpt. Moreover, the implementation complexity does confine the deployability when, e.g., certain mathematical operations must be supported or a certain amount of state information must be stored. In general, it always should be a goal to design an algorithm as simple as possible [72], as this can help to avoid implementation errors that can influence the stability in (unevaluated) border cases. A more simple approach should be preferred as long as a more complex approach does not provide significant performance improvements. As complexity is hard to measure, at least the implementation complexity should be discussed compared to other approaches. Additionally, the interoperability with other approaches that potentially have other optimization goals must be shown if deployment in the Internet is envisioned [125].

2.5.4 Summary and Discussion

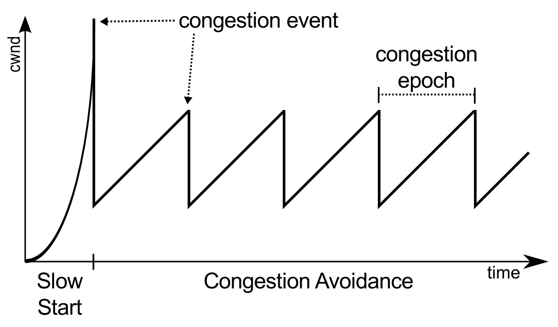
As already stated above we use the IKR SimLib with VMSimInt extension for evaluation. This setup provides a controlled simulation environment that allows us to generate reproducible results without influence of outer factors such as processing delays. Further, it allows us to integrate our Linux implementation and thereby provides a realistic environment for investigation.

While the TCP Evaluation Suite proposes a set of initial scenarios mainly for comparison with existing schemes, we focus on a larger set of simple scenarios to first of all evaluate the microscopic behavior of our proposed algorithm. We further evaluate extreme and corner cases to investigate isolated effects of our algorithm. Thereby, we can provide confidence that TCP SIAD is robust to be used in the Internet. As further explained in Section 4 we orient a few of our parameter choices on the TCP Evaluation Suite. However, we did not use any traffic traces to re-play Internet traffic. The existing traffic traces cannot be used to generate a certain traffic load in steady state as this kind of traffic mix usually takes a long time to reach steady state. Further, due to the Internet-typical traffic mix with a heavy-tail distribution of flow sizes, the traces include mostly small flows. These flows usually do not or only for a short time leave Slow Start and, as we did not change the Slow Start behavior, this does not target our evaluation. Instead, we consider scenarios with one long-living flow in a traffic mix with short flows to explicitly evaluate the influence of short traffic peaks.

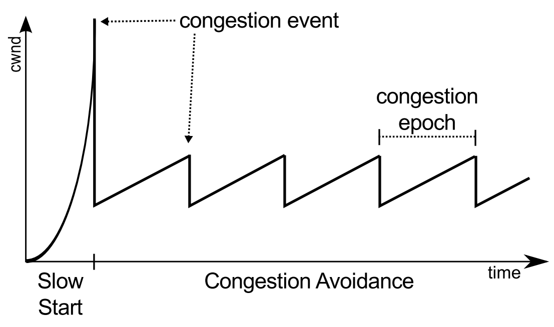
Based on the selected set of scenarios and test cases we evaluate our algorithm with respect to the requirements as stated in Section 1.2, namely scalability, adaptivity, capacity sharing, and convergence. These requirements are slightly different than the ones listed in the previous section. While convergence is a classical requirement for congestion control, we evaluate efficiency with a focus on scalability and adaptivity as both features are explicitly addressed by our design goals. As explained in the introduction, we do not target equal sharing between competing flows. However, we aim to support a configurable capacity sharing. Note that due to RTT differences and other influences of randomness, equally sharing is anyway only rarely reached in the Internet. Finally, we also evaluate general robustness for the use in the Internet in high congestion scenarios and TCP SIAD's vulnerability to delay estimation errors.

3 TCP Scalable Increase Adaptive Decrease (SIAD) Algorithm

In this chapter we describe the proposed congestion control algorithm in detail and provide reasoning why we choose specific design approaches. We further explain the implementation as Linux kernel module that is used for the simulative evaluation in the following chapter. We can conclude from the previous section that none of the discussed existing schemes can fulfill all the design goals as listed in Section 1.4. Therefore, we propose TCP *Scalable Increase Adaptive Decrease* (SIAD), which addresses all goals and is designed based on a fully new approach called *Scalable Increase* that aims for a fixed congestion feedback rate independent of the available network bandwidth. This means, TCP SIAD solves the scalability problem as it scales in any future network, which can only be provided as well by Scalable TCP. However, Scalable TCP is based on a completely different approach and therefore scales at the cost of inducing a high congestion rate. Further, TCP SIAD addresses the problem of today's Internet to sufficiently support services that require low latency. H-TCP is an existing approach that implements the same decrease function as TCP SIAD to better adapt to small queues. Unfortunately, it does not address the standing queue problem with large buffers as TCP SIAD does. We designed TCP SIAD without requiring TCP-friendliness, as fairness should be addressed on a per-user, not per-flow basis, and therefore by mechanisms other than congestion control. This is an important aspect which evolved over the recent years. Instead TCP SIAD provides a configuration possibility to the application to influence the aggressiveness and therefore the instantaneous capacity sharing between flows.



(a) SIAD e.g. with 1 BDP buffer.



(b) SIAD with smaller buffer.

Figure 3.1: The Scalable Increase Adaptive Decrease (SIAD) scheme.

The proposed algorithm design is named TCP SIAD as composed out of two basic components which are called *Scalable Increase* and *Adaptive Decrease*. TCP SIAD uses a linear increase

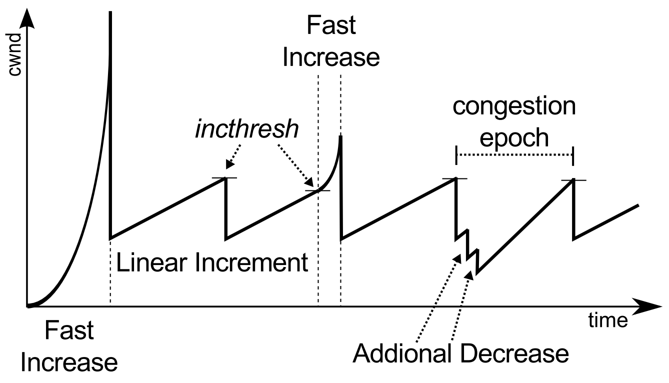


Figure 3.2: Example behavior of TCP SIAD.

of α packets per RTT and a multiplicative decrease with factor β when congestion occurs. Therefore, it (still) implements the AIMD scheme as introduced in Section 2.2. In contrast to traditional AIMD-based proposals, α and β are not fixed but re-calculated for each congestion epoch. While Scalable Increase aims to receive the congestion feedback with a constant frequency independent of the link capacity, Adaptive Decrease aims to empty the queue exactly, without causing underutilization or a standing queue.

Figure 3.1a shows that TCP SIAD with one BDP of buffering in the network and an appropriately configured feedback rate can behave similar to TCP NewReno. If in the same scenario the buffer size is now reduced, Adaptive Decrease calculates a smaller decrease factor to keep link utilization high. Additionally, Scalable Increase calculates a smaller increase rate to maintain the same congestion feedback frequency than before as exemplary shown in Figure 3.1b.

Additional to this basic principle of SIAD, we introduce three extensions which we briefly introduce next and are schematically shown in Fig. 3.2.

Additional Decrease One or more Additional Decreases can be performed during a congestion epoch (not only on congestion notification). The Additional Decrease aims to empty the queue completely at least once in a congestion epoch in case the regular decrease was not able to, e.g., due to competing traffic or measurement errors.

Fast Increase In Congestion Avoidance we introduce two phases, *Linear Increment* and *Fast Increase*. When the congestion window grows above the Linear Increment threshold *incthresh*, which is also a target value for the α calculation of Scalable Increase, TCP SIAD enters the *Fast Increase* phase. In this situation TCP SIAD does not have a target value for the congestion window anymore as potentially new capacity is available. TCP SIAD now slowly increases the increase rate to quickly allocate newly available bandwidth.

Trend The calculation of the target value, namely the Linear Increment threshold *incthresh*, depends on the maximum value of the congestion window before the decrease as well as on the history of these maximum values. By this approach we introduce a trend that influences the increase factor α in dynamic scenarios and thereby improves convergence.

Note, Additional Decrease might decrease more than needed to empty the queue but this ensures that the queue becomes empty. Fortunately, whenever TCP SIAD has decreased more than needed and therefore underutilizes the link, the Scalable Increase scheme calculates an even larger increase factor to reach the target value in time to maintain the configured length of a congestion epoch. Therefore, underutilization usually only occurs for a short time and can be partly compensated by Scalable Increase compared to schemes with a fixed increase rate.

In the following section we discuss the design approaches taken for the presented components as well as give the reasoning our design decisions. Afterwards, in Section 3.2, we describe the complete algorithm in detail. In Section 3.3 we explain implementation details for the integration as Linux kernel module and highlight important effects for the interpretation of the evaluation results in the next chapter. Finally, we summarize this chapter as well as discuss implementation complexity and further degrees of freedom in our design decisions.

3.1 Design Approaches

As mentioned already in the previous chapters, we decided to design a congestion control scheme that is reacting to loss (or an explicit congestion signal as ECN) as this is the premier congestion feedback signal in the current Internet. Therefore, to be able to co-exist with existing Internet traffic, we need to react to the same feedback signal. In contrast, delay-based approaches usually react to an earlier congestion signal, namely increasing delay when the queue builds up, and as such are not able to allocate capacity when competing with loss-based traffic. By using delay measurement to adapt the decrease factor, TCP SIAD uses delay as a secondary congestion signal. Therefore, TCP SIAD is a hybrid scheme.

Further, to frequently probe for newly available capacity and at the same time being able to quickly yield capacity to, e.g., newly starting flows, TCP SIAD is designed based on the AIMD scheme. Therefore, TCP SIAD does not converge to a single steady state but operates in an equilibrium state between a minimum and maximum congestion window value as long as the network conditions are stable.

Scalable Increase in Linear Increment phase

One of our design goals is that the period between two congestion events should be constant, and therefore independent of the current link capacity or number of competing flows. This means that in steady state the time period after a window reduction until the same window size as before the congestion event is reached again should be equal in every congestion epoch. As the decrease is multiplicative, and the decrease factor can even change from congestion epoch to congestion epoch, the absolute number of packets by which the window is reduced depends on the current maximum window and thereby the available capacity. To still maintain the same time period for each congestion epoch, the increase factor has to be adapted dynamically after each window reduction. We call this approach Scalable Increase as it fully resolves the scalability problem due to providing a fixed feedback rate by dynamic adaptation of the increase rate based on the size of the antecedent decrease.

In this section we discuss two questions regarding the detailed design of the Scalable Increase:

1. What is the right time period for one congestion epoch?
2. How should the increase curve to reach the target value look like?

We address the first question by introducing a configuration parameter. This parameter gives either directly the absolute desired time period or alternatively the number of RTTs for one congestion epoch. This allows the most flexibility for many usage scenarios. Further, the smaller the feedback period is, the smaller is the time to detect that new capacity is available. Therefore, the configured time for a congestion epoch will also influence the responsiveness of the congestion control.

In the Internet it often makes sense to configure the congestion feedback rate dependent on the current RTT. E.g., when communicating with a cache that is only a few milliseconds away, the feedback rate should be higher than when communicating from Europe with a server in the USA. In contrast in more homogeneous scenarios, e.g. as a data center, it might be more sensible to configure the feedback rate based on a fixed time interval as it is important to adapt within a certain time range. Note that if the congestion epoch is configured in RTTs, and not as a fixed time, in a scenario as shown in Figure 3.1b the congestion epoch gets smaller, as the average RTT decreases if the maximum buffer size is smaller.

In fact, Scalable Increase provides exactly the configured feedback rate if there is just one flow on the bottleneck link and thus congestion is only induced by itself. When competing with other flows, these flows might impose additional congestion with a higher frequency. In this case the configured rate cannot be reached anymore, but instead the share of the capacity between the competing flows is determined by the ratio of the configured feedback rates. This means the configuration interface could be used by a higher-layer control loop in the application to impact the capacity share between competing flows (on the cost of higher congestion) and therefore to better cope with the requirements of the specific application; potentially at the cost of higher congestion.

For the configuration parameter, in most cases, or at least at the beginning of a connection, a default value can be used. Additionally, some applications might implement a respective mechanism to control the aggressiveness dynamically during a transmission. This is further discussed in Section 3.4.3 However, a detailed study of such a higher-layer control loop is not part of this work as it strongly depends on the specific application.

Regarding the second question, we decided to increase the congestion window linearly during one congestion epoch in the Linear Increment phase. Alternatively, e.g., one could increase based on a concave curve as TCP Cubic or convex as H-TCP.

TCP Cubic decided for a concave increase behavior as this quickly reallocates bandwidth after the decrease. As TCP SIAD always tries to keep the link fully utilized even after a window reduction, there is no need to increase quickly afterwards.

A convex behavior, instead, keeps the queue at a lower fill level for a longer time during each congestion epoch and therefore reduces the average queuing delay. This goes in-line with our design goals. Unfortunately, to generate a loss-(/ECN-) based feedback signal we need to fill the queue respectively and therefore ramp up even more quickly afterwards in case of the convex increase behavior. This causes a larger overshoot for each congestion event, meaning a convex increase behavior induces several losses within the one RTT where the queue is overloaded. However, one loss in fact is sufficient to notify the sender's congestion control algorithm.

A more complex approach would be, e.g., to first increase very slowly, then ramp up quickly to a point just below the Linear Increment Threshold, and finally again increase slowly to the final value. This could potentially fulfill both staying as long as possible at a low queue fill level and avoiding a large overshoot. We assume that network operators will configure smaller buffers in future, which also means smaller sawtoothing in case of TCP SIAD. Therefore, the average the queuing delay might still not be much smaller than when using a linear increase instead. In fact, if the queue is small, the shape of the increase curve no longer has a big influence on the average queuing delay. To maintain the trade-off between low average delay and a small overshoot as well as in the favor of simplicity, we decided to stay with the linear increase behavior.

Adaptive Decrease

To maintain high utilization the congestion control scheme must always maintain a high enough sending rate to fill the bottleneck link, even after a decrease. In the best case, the congestion window should only be reduced to exactly the sending rate at which the buffer gets empty and therefore can drain to zero over the next RTT while the link stays full. Of course, network operators cannot adapt their buffering to the decrease behavior and BDP of each and every single flow sharing the current bottleneck. Therefore, the congestion control algorithm in the end host should adapt its decrease behavior to the available buffer to avoid underutilization as well as a standing queue.

As soon as the bottleneck queue fills, end-to-end delay and thus RTT increases. In contrast, the minimum RTT RTT_{min} can be observed only when the queue is empty. This means until the buffer starts filling the measured RTT stays constant even though the congestion window is growing (disregarding short term bursts as, e.g., can be observed in Slow Start). The queuing delay, itself, can be calculated by subtracting the minimum from the currently measured RTT. If the sender is not limited elsewise, the congestion window indicates the current number of packets in flight; which are the packets on the link as well as in the buffer. Therefore, the ratio of the queuing delay $qdelay$ to the current RTT $RTT(t)$ equals the ratio of the number of packets in the queue q to the current congestion window value (assuming a single bottleneck queue).

$$\frac{qdelay}{RTT(t)} = \frac{RTT(t) - RTT_{min}}{RTT(t)} = \frac{q}{cwnd} \tag{3.1}$$

This relation was also used by TCP Vegas, TCP Africa and Compound TCP (see Eg. 2.36) to estimate the number of back-logged packets in the queue q and, in their cases, compare it with a threshold.

$$q = \frac{RTT(t) - RTT_{min}}{RTT(t)} \cdot cwnd \tag{3.2}$$

TCP SIAD instead uses this information to calculate a proper decrease factor. H-TCP [93] was the first proposal that uses this estimate to dynamically calculate the decrease factor on congestion notification. In case of loss-based congestion control, the end-to-end delay reaches its maximum when the queue overflows and subsequently the congestion control algorithm in the end host is notified that the link is congested. Therefore, the new congestion window should be

$$cwnd \leftarrow cwnd - q_{max} = (1 - \frac{RTT_{max} - RTT_{min}}{RTT_{max}}) \cdot cwnd = \frac{RTT_{min}}{RTT_{max}} \cdot cwnd \qquad (3.3)$$

where RTT_{max} is the measured RTT when the queue is full. Note that H-TCP [93] only allows a decrease of $0.3 \cdot cwnd$ up to $0.5 \cdot cwnd$ and therefore only adapts to smaller buffers/buffer fill levels (than one BDP) but might still cause a standing queue for large buffers. In TCP SIAD, we do not restrict the decrease at all, as a decease that is too large is partly compensated by Scalable Increase and a decrease that is too small by the Additional Decrease approach. The decrease factor could, e.g., be wrongly estimated due to measurement errors or the influence of competing cross traffic on the bottleneck link as further explained below.

Additional Decrease

In fact, Eq. 3.2 only gives the queue length when the flow is alone on the bottleneck link. However, if cross traffic is sharing the bottleneck link, only the share of the queue of this flow is estimated. This is because Eq. 3.2 relates to the congestion window of this flow which also (just) reflects the current bandwidth share of this flow. This means using the decrease function as given above in Eq. 3.3 only empties the queue if either the flow is alone on the link or all competing flows on the bottleneck are synchronized. All flows being synchronized means that they all get a congestion notification in the same RTT and consequently perform their window reduction within the same RTT. That often does not happen in reality, e.g., because of the influence of delayed ACKs, as later shown in evaluation. Still, this case needs to be considered, especially if mechanisms to compensate for delayed ACKs are used, as with TCP SIAD.

Unfortunately, at least for larger aggregates, flows are usually not fully synchronized. Therefore, the correct decrease factor to empty the queue but not underutilize the link depends on the number of synchronized flows and the total queue length, which are both unknown to a single flow. TCP SIAD still performs the Adaptive Decrease on congestion notification as described above. But if the queue does not empty because all flows were not synchronized, TCP SIAD performs one or more Additional Decreases. More precisely, an Additional Decrease is performed when the minimum RTT cannot be measured after a regular, adaptive decrease.

For the concrete implementation of Additional Decrease two questions have to be addressed:

1. How many additional decreases can be allowed per congestion epoch?
2. How large should each additional decrease be?

We can only perform one additional decrease per RTT as the resultant changes in queuing delay can only be measured with a feedback delay of one RTT. Therefore, the maximum number of decreases that could be performed depends on the configured congestion feedback frequency. As Scalable Increase still aims to maintain the respective feedback frequency, the increase rate is recalculated after each Additional Decrease. This means there must be at least one RTT left to reach the respective target value.

Moreover, in general we only allow an increase rate that at maximum doubles the congestion window, as in Slow Start. If the newly calculated increase rate per RTT becomes larger than the current congestion window value, no additional decreases are performed anymore. This usually only happens if the currently congestion window is already very small and/or only a few RTTs are left to reach the target value. We also do not perform any further decreases if the congestion window is set to the minimum value by Additional Decrease. In these cases, if the queue is still not empty and the minimum delay still cannot be measured, this single flow cannot reduce the queuing delay any further.

As in case of Additional Decrease the currently measured delay is still larger than the minimum delay, we can still apply the decrease function as in Eq. 3.3. But this again only reduces by the share of this flow in the queue and might not empty the queue completely. To decrease by the correct amount we would need to know the total current queue length and how many flows are reducing their sending rate synchronized and by how much. One could estimate the total queue length based on the bottleneck rate C that could be derived from the ACK inter-arrival time and RTT measurements by

$$q_{total} = \frac{RTT_{max} - RTT_{min}}{C}. \tag{3.4}$$

Decreasing by this amount would empty the queue but in many cases also underutilize the link as we do not know what the competing traffic is doing. Further, an estimation based on the ACK inter-arrival time might not be very accurate. Therefore, we will first apply Eq. 3.3 for each additional decrease and then additionally decrease such that we can reach the minimum congestion window with the maximum number of decreases that are possible with the current configuration. E.g., if the configured number of RTTs for a congestion event is 20, we can at maximum perform 19 additional decreases and for the first additional decrease reduces at least additionally by 1/19 of the current congestion window.

Moreover, we have to at least decrease by the new α number of packets, as this is the number of increases that we will perform during the next RTT. During this RTT we would like to keep the queue empty to be able to measure the minimum RTT correctly. Note that while one flow performs additional decreases, other competing flows might still increase their rate. Therefore, it can be hard to measure the minimum correctly. In the worst case, Additional Decrease reduces the congestion window to the minimum allowed congestion window value (over several steps). This is not a problem with respect to the desired capacity sharing, as the increase is even faster after these Additional Decreases due to the Scalable Increase that still maintains the same time between congestion two congestion events. Finally, it could even be the case that we cannot measure the minimum RTT at all due to competing traffic that always keeps the queue full or at least not long enough to be sure that the queue is empty.

To summarize, we basically have two cases: If the buffer size is larger than one BDP, we additionally decrease by α; if buffer size is smaller, the decrease depends on the configured time for a congestion epoch. Further, we have also considered different decrease schemes, e.g. halving. As there is always a tradeoff between decreasing too much and thereby underutilizing the link and decreasing too few and thereby causing several decreases in a row that cause a larger increase rate later, we found this a good compromise.

Further, note that Additional Decrease also helps to adapt quickly to changes in the non-queuing base delay. Note, if the end-to-end delay decreases this is measured and updated automatically, but not if it increases. By ensuring that the queue gets empty to the extent possible we are also able to measure the minimum RTT in every congestion epoch. If the base delay has increased in the mean but we can measure the same delay within subsequent RTTs, even though we were already increasing our sending rate, we assume that the queue is empty and update our minimum RTT value. Therefore, we can also quickly adapt to increases in the minimum RTT, e.g. due to route changes, while most delay-based scheme have to implement separate mechanism to trace such changes, as e.g. explained in Sec 2.2.2.1 for LEDBAT.

Fast Increase above Linear Increment Threshold

During a transmission, congestion control has to handle two cases: either the network conditions are stable (all competing flows have converged to their steady state and maintain current link utilization) or the network conditions are changing (e.g. due to starting or stopping flows) and all competing flows have to converge to a new equilibrium. To be able to detect these changes, the AIMD scheme frequently probes for new capacity and frequently reduces the sending rate to provide space in the queue for new staring flows.

While in steady state the target rate at which overload is expected is known, it is hard to determine how much and how quickly the rate should be increased to allocate newly available capacity. Therefore, TCP SIAD maintains two different increase phases, Linear Increment and Fast Increase. In Linear Increment, TCP SIAD increases the sending rate linearly to the target value as discussed above. During transmission start-up or above the target value, or more precisely above the Linear Increment threshold, when new bandwidth becomes available, no target is known. Therefore, TCP SIAD increases the sending rate faster in Fast Increase to be able to quickly allocate new bandwidth and consequently quickly converges, even in high-speed networks. TCP Cubic also addresses this problem by implementing a cubic curve that increases the sending rate carefully around the target but more aggressively afterwards. However, as we show in Chapter 4 the implementation as in TCP Cubic spends a long time probing around the target before allocating new capacity in high-speed networks.

We designed the increase behavior in Fast Increase to be similar to traditional Slow Start. In the actual start-up phase we set our increase rate to the current congestion window, and double both the congestion window as well as the increase rate once per RTT. When we enter Fast Increase, TCP SIAD implements the same behavior of increasing the increase rate but starts with the smallest possible initial increase rate to carefully probe around the target.

Increasing the increase rate in Fast Increase might cause a large overshoot when the maximum capacity is reached and therefore causes several unnecessary packet losses at once. There is a general trade-off between responsiveness/convergence speed and smoothness of the oscillation/size of the overshoot as discussed in Section 2.5.3. We handle this trade-off by limiting the maximum sending rate in Fast Increase to not increase the congestion window more than 1.5 times in one RTT.

The second case where flows need to converge but the capacity is currently fully utilized, e.g., in case of new starting flows, the convergence time is not only be determined by the increase rate but also by how quickly the already running flows can release capacity. Unfortunately, when the buffer size is small, the adaptive, multiplicative decrease factor is small as well. We do not further address this aspect by a separate mechanism as a potentially larger decrease would make our algorithm sensitive to short traffic bursts of flows that start in TCP Slow Start and therefore push away long-lasting flows but do not exist long enough to use the freed capacity. Further, note that only decreasing more does not help the problem, as Scalable Increase will subsequently calculate a higher increase factor to reach the target value in the configured time. Therefore, implementing a larger decrease, if e.g. the target value could not be reached before congestion occurred, only causes link underutilization but does not help convergence. Instead, one would need to also adapt the target value, similar as to proposed trend calculation that is described next. We did not further investigate a respective mechanism to address this case for the sake of simplicity and it is more important for us that the link stays utilized as long as we have a mechanism that ensure convergence as the trend calculation that is described next.

Trend Calculation for Convergence

AIMD was shown to be able to converge to equal capacity sharing for competing flows with the same aggressiveness from any starting point. This is because, even though it implements the same increase rate for all flows, the multiplicative decrease leads to a smaller decrease regarding the total number of packets for the flow(s) with the currently smaller sending rate. If now all flows use the same increase rate, the smaller rate flows end up at a higher congestion window at the next congestion event than at the previous congestion event. With Scalable Increase, unfortunately, the flow with the smaller decrease also calculates a smaller increase because it targets the same maximum congestion window value as before. Therefore, Scalable Increase in fact revokes this convergence principle.

Consider two competing flows using the basic SIAD scheme, configured to achieve the same congestion feedback rate, but for some reason they currently send with different rates. Adaptive Decrease calculates the same multiplicative decrease factor for both flows as based on the same measured RTT values. The same decrease factor leads to a larger absolute decrease for the flow with the higher rate, similar to the multiplicative decrease behavior of TCP NewReno with a fixed decrease factor. But at the same time the flow with the higher rate also chooses a larger linear increase factor to reach the previous maximum congestion window in the same time as the competing flow (with the lower rate). Therefore, both flows get the next congestion notification when reaching the same window size as before.

To reach convergence, the flow with the lower rate needs to increase more quickly than the flow with the higher rate. Choosing the increase rate indirectly proportional to the sending rate is not a good idea either, as this means that the flow with the large sending rate increases more slowly and therefore this does not scale well with high-speed, large-BDP networks.

To address this convergence problem, TCP SIAD additionally calculates a trend. We in fact try to detect when the network situation is stable or not by comparing the maximum congestion window value at the previous congestion event to the current congestion window value at the current congestion event. If the congestion window at the previous congestion event was higher, we increase slightly slower, as we expect cross traffic tries to allocate capacity. If the window was smaller, we increase faster to grab new capacity.

For the trend calculation we only consider the congestion window value of the previous congestion event. The trend could also be calculated over a longer history. This might smooth oscillation but potentially also slows down convergence. Again, that is a tradeoff between responsiveness and smoothness, so we decided to only take the most recent history into account.

Another extension would be to observe the development of the trend of time and respectively weaken or amplifying the trend even more. In the sake of simplicity we did not further evaluate this approach.

3.2 Algorithm Design

As already described above, Scalable Increase and Adaptive Decrease re-calculate the increase rate of α packets per RTT and, respectively, the decrease factor β for each congestion epoch on congestion notification. Moreover, α is re-calculated after an Additional Decrease and dynamically adapted in Fast Increase.

3.2.1 Scalable Increase

To implement a constant feedback frequency, the linear increase factor α needs to be recalculated after every decrease. α gives the number of packets that the congestion window can be increased per RTT. Therefore, α is simply calculated by dividing the total number of packets to reach the target congestion window value by the desired/configured number of RTTs of one congestion epoch. Further, the total number of increases in packets during one congestion epoch can be calculated based on the congestion window after the congestion window reduction that is set to *ssthresh* and the desired target value *incthresh* that should be reached at the end of the congestion epoch. The number of RTTs during one congestion epoch is configured by the configuration parameter Num_{RTT} or, alternatively, estimated based on a configured absolute time interval and the expected average RTT for the next congestion epoch. With *ssthresh* being the congestion window after the window reduction and *incthresh* the target value, Scalable Increase calculates the increase α of packets per RTT as

$$\alpha = \frac{incthresh - ssthresh}{Num_{RTT}}, \quad 1 < \alpha < ssthresh$$

In the next congestion epoch we linearly increase by the standard Additive Increase function

$$cwnd \leftarrow cwnd + \frac{\alpha}{cwnd} \quad \text{[per ACK]}.$$

We limit α to a minimum increase of 1 packet per RTT as TCP Reno in Congestion Avoidance and allow at maximum an exponential increase as in Slow Start.

3.2.2 Linear Increment Threshold and Trend

Before determining the new α value, we have update the Linear Increment threshold appropriately. Therefore, we calculate *incthresh* based on *trend* by

$$incthresh = cwnd_{max} + trend, \quad incthresh \geq ssthresh$$

where $cwnd_{max}$ is the estimated congestion window when the congestion occurred as described further below in Sec. 3.2.4. Note, $trend$ can be positive or negative but $incthresh$ cannot get smaller than the congestion window after the decrease $ssthresh$. $trend$ is calculated by

$$trend = cwnd_{max} - prev\_cwnd_{max}$$

where $prev\_cwnd_{max}$ is the estimated maximum congestion window at the previous congestion event.

3.2.3 Fast Increase

If in Congestion Avoidance, the congestion window grows above $incthresh$, TCP SIAD enters *Fast Increase* phase. In Fast Increase there is no target value as new capacity has probably become available. To quickly speed up, we adjust the increase step size α dynamically over time within the congestion epoch. As soon as the current congestion window reaches the $incthresh$ in Linear Increment, we reset α to 1. Above $incthresh$ (and below the Slow Start threshold $ssthresh$) we double the increase step size α per RTT. This means we adapt α by

$$\alpha \leftarrow \alpha + \frac{\alpha}{cwnd} \quad \text{[per ACK]}, \quad \alpha \leq \frac{cwnd}{2}$$

which in fact would be an increase by 1 for each congestion window increase. In Fast Increase when $cwnd$ is above $incthresh$, we limit α to $\frac{cwnd}{2}$. This means we can only increase the congestion window by a factor of 1.5 per RTT. This avoids too large oscillations and therefore achieves a more stable behavior.

On initialization, $incthresh$, α as well as $cwnd_{max}$ are set to the initial congestion window value. This initial setting leads to an exponential increase as in Slow Start.

When the Slow Start threshold $ssthresh$ is passed and there is no $incthresh$ (that is larger than $ssthresh$), we reset α to 1 and enter Fast Increase directly. If there is an $incthresh$ value that is larger than $ssthresh$, we recalculate α as in Eq. 3.2.1 and enter Linear Increment.

3.2.4 Adaptive Decrease

As the congestion feedback has a delay of one RTT, the congestion window is further increased during that time period until the congestion notification is received. This means, we need to consider the congestion window value $cwnd_{max}$ that was used one RTT ago at the time where the congestion actually occurred. Therefore, $cwnd_{max}$ is calculated based on the current congestion window value $cwnd$ (before the decrease) minus the number of increases that were performed during the last RTT. In Linear Increment, the number of increases is usually α. But as we change the α in Fast Increase, we have to consider several different cases leading to the following adaptation

$$cwnd_{max} = cwnd - \begin{cases} \frac{cwnd}{2} & \text{if } \alpha \leq cwnd \vee cwnd \leq ssthresh \\ \frac{cwnd}{3} & \text{if } cwnd > incthresh \wedge \alpha = \frac{cwnd}{2} \\ \frac{incthresh-ssthresh}{Num_{RTT}} & \text{if } cwnd \geq incthresh \wedge \alpha = 1 \\ \frac{\alpha}{2} & \text{if } cwnd > incthresh \\ \alpha & \text{otherwise.} \end{cases}$$

As mentioned above for the regular case in Linear Increment we can reduce the current window value simply by α (see bottom line). If we are in Slow Start or (far enough) above the Linear Increment threshold, we reduce the current window usually by $\alpha/2$ as the increase rate was doubled during the last RTT. Further, if the Linear Increment threshold was just surpassed and α has been reset to 1, we still have to apply the increase value that was used before in Linear Increment and can be recalculated based on Linear Increment threshold *incthresh* and Slow Start threshold *ssthresh*. For simplification, we neglect the rare case where Slow Start threshold was just surpassed. However, we also have to consider the case when the maximum increase rate of α was reached. In Fast Increase this is $\alpha = \frac{cwnd}{2}$, and therefore we reduce the congestion window by one third in this case. Otherwise the maximum value is reached if $\alpha = cwnd$. In this case we have to half, respectively.

We do not apply this adjustment right after a decrease if the congestion window has not been increased yet. According to the function given above this would wrongly lead to window halving, even though our increase rate is only α packets per RTT, as the congestion window still equals *ssthresh*. In fact, as we did not increase at all yet, no adjustment is needed. This case can actually occur if congestion is still signaled but the congestion state is already left (e.g. due to distributed losses over a whole RTT). For simplification we decrease by α as soon as the congestion window has been increased once. This can lead to a larger adjustment than needed but usually this is not a problem because it is better to be conservative when receiving a congestion notification in first RTT after decrease.

Adaptive Decrease aims to exactly empty the queue without causing underutilization or a standing queue. Therefore, the decrease factor β is calculated based on RTT measurements of the minimum and current RTT, RTT_{min} and RTT_{curr}, as already explained above in Section 3.1, as

$$\beta = \frac{RTT_{min}}{RTT_{curr}} \tag{3.5}$$

and the congestion window is decreased by

$$cwnd = \beta \cdot cwnd_{max} - 1 \text{ [on congestion event]} \tag{3.6}$$

We additionally decrease by one, as it is important for us to empty the queue completely. Additionally, the congestion window is cropped to a minimum value of $MIN\_CWND = 2$. Further, RTT_{curr} is filtered to remove single outliers by using the minimum of the last two RTT measurement samples before the congestion event. Note the RTT at the moment when the congestion notification is reached corresponds to the current queue length. In case of loss (without AQM), the queue is filled to the maximum in this case. Further, note, if no valid RTT information are available, e.g., if the congestion notification is received right after congestion start, the congestion is halved.

H-TCP [93] only allows a decrease of 0.3 up to $0.5 \cdot cwnd$ and therefore might still cause a standing queue for large buffers. In TCP SIAD, however, we do not restrict the decrease factor, as a too large decrease is partly compensated by Scalable Increase and a too small decrease by Additional Decrease as explained next.

3.2.5 Additional Decrease

Unfortunately, the above described decrease behavior only drains the queue if all competing flows are synchronized, which means they must perform their decrease in the same RTT. As TCP SIAD aims to empty the queue within every congestion epoch, TCP SIAD performs an additional decrease in the Linear Increment phase if it cannot measure the minimum RTT about one RTT after the first regular, adaptive decrease. More explicitly, if we cannot observe either an RTT measurement that is equal to or smaller than the total minimum delay RTT_{min} or the same delay samples in subsequent RTTs, we perform an additional decrease. Note if the (minimum) RTT stays constant even though the sending rate was increased, this indicated that the queue is empty.

Additional Decrease is performed immediately within the current congestion epoch without further congestion notification. We decrease the congestion window first to

$$cwnd = \frac{RTT_{min}}{RTT_{curr}} ssthresh - 1$$

as the current RTT is still larger than the minimum and therefore this approach leads to the at the least needed reduction. Note that we use *ssthresh* here as this is the congestion window value about one RTT ago, which corresponds to our current RTT measurement. Moreover, we afterwards also decrease the congestion window the following

$$cwnd \leftarrow cwnd - max(red, \alpha_{new})$$

where *red* is a reduction factor that depends on the remaining number of RTT in this congestion epoch and α_{new} is the new α that we would need apply after the reduction. *red* is calculated such that congestion window can be reduced to its minimum value within at maximum $Num_{RTT} - 1$ RTTs if further Additional Decreases would be applied.

$$red = cwnd \cdot \frac{1}{Num_{RTT} - dec\_cnt},$$

where *dec_cnt* is the number of Additional Decreases already performed during this congestion epoch (including the current one). Further, we have to at least decrease by α_{new} number of packets

$$\alpha_{new} = \frac{incthresh - cwnd}{Num_{RTT} - dec\_cnt - 1}$$

as this is the number of increases that we will perform during the next RTT and we would like to keep the queue empty during this RTT to be able to measure the minimum RTT correctly.

Note that while one flow performs Additional Decreases, other competing flows might still increase their rate. Therefore, it can be hard to measure the minimum correctly. Further, note, we perform at maximum $Num_{RTT} - 1$ Additional Decreases and also stop when the congestion window is decreased down to MIN_CWND as this is the maximum one flow can contribute to keep the delay low.

Of course, Scalable Increase still aims to reach $incthresh$ within the remaining RTTs. Therefore, a larger increase factor is recalculated after each Additional decrease. Only if red is larger than α_{new}, we have to calculate α again by

$$\alpha = \frac{incthresh - cwnd}{Num_{RTT} - dec\_cnt}$$

otherwise we set $\alpha = \alpha_{new}$. Note, even if α is larger than the current value $cwnd$, we at maximum double the congestion window per RTT (until a large enough $cwnd$ value is reached and α is smaller again). Therefore, we also do not performed any further Additionally Decreases if the new α is larger than the new congestion window value. Finally, we set $ssthresh$ to $cwnd - 1$.

To estimate the right value for the minimum RTT we store both an absolute minimum RTT_{min} and the current minimum RTT_{min_curr} that we have seen so far in this congestion epoch. If the same RTT_{min_curr} value can be measured over several RTTs, even though we have increase the congestion window in the meantime, we can assume that the queue was empty and update RTT_{min} to RTT_{min_curr}. Otherwise if the queue would not have been empty, the delay should immediately increase with every increase in sending rate. As TCP SIAD aims to empty the queue in every congestion epoch, we should also be able to update the minimum in every congestion epoch, e.g., if the base delay has increased due to routing changes.

Moreover, we remember the last three values of the minimum RTT at decrease. If we observe that these values are monotonic increasing, we assume an estimation error, e.g., when the queue was not emptied completely but no increase in delay was observed, e.g., due to a too low time stamp resolution. If this is the case, we reset the minimum to the oldest, stored minimum value. With this mechanism, we are still not able to decrease the queue completely when estimation error occur but at least we can avoid an growing standing queue due to an growing, wrong minimum RTT estimation.

3.3 Implementation

We implemented TCP SIAD in the TCP stack of the Linux kernel version 3.5.7 as congestion control kernel module as explained in Sec. 2.4. The complete source code can be found in Appendix A. More precisely, we implemented the following four methods:

`void tcp_siad_init(struct sock *sk)`

This method is used to initialize our congestion control state; mainly the increase rate to behave similar to Slow Start at connection start-up.

`void tcp_siad_cwnd_event(struct sock *sk, enum tcp_ca_event event)`
This method is used to reset the state before the next increase period when the congestion state is left. Therefore, we only take action if the `CA_EVENT_COMPLETE_CWR` event is passed.

`void tcp_siad_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)`
Whenever an ACK is received, this method is called and performs the following steps:

1. estimate the current delay sample and potentially reset the minimum delay,
2. filter the current delay by taking the minimum of the current and the previous sample, and
3. increase or decrease (in case of Additional Decrease) the congestion window.

`u32 tcp_siad_ssthresh(struct sock *sk)`
When the congestion recovery state is entered, this method is called and performs the following steps:

1. estimate the maximum congestion window $cwnd_{max}$ when congestion occurred,
2. calculate the new Slow Start threshold $ssthresh$, and
3. calculate a new Linear Increment threshold $incthresh$ and a new increase per RTT α.

This method returns the new Slow Start threshold $ssthresh$.

This means `tcp_siad_cong_avoid()` and `tcp_siad_ssthresh()` implement the increase and, respectively, decrease behavior of TCP SIAD as described above. Note that the current congestion window `snd_cwnd` as well as the Slow Start threshold `snd_ssthresh`, can be derived from the socket memory which is passed to each of method (as the input parameter `sk`). When a congestion event occurs, `tcp_siad_ssthresh()` is called to set a new Slow Start threshold $ssthresh$. Subsequently, the congestion window is reduced stepwise to this new Slow Start threshold until the status reset in `tcp_siad_cwnd_event()` and the congestion event is over. In the kernel version used, the PRR [103] algorithm is implemented which guarantees that the congestion window is set to exactly the calculated value of $ssthresh$ at the end of the recovery phase (about one RTT after the first congestion notification).

When loading kernel modules default parameters can be passed. Our implementation has two parameters, namely

- *num_rtt* which maintains the desired number of RTTs between two congestion events (if the resulting time interval is larger than the configured number of milliseconds *num_ms*) and
- *num_ms* which maintains the desired milliseconds between two congestion events (if larger than the resulting time interval for the configured number of RTTs *num_rtt*).

If these parameters are not initialized at module loading time, they are set to 20 and 0, respectively. Therefore, the length of a congestion epoch is by default targeted to be 20 RTTs.

Additionally, the congestion epoch length can be influenced after loading by the kernel parameter setting using two new sysctl interfaces which can be accessed using the following parameters: *net.ipv4.tcp_siad_num_rtt* and *net.ipv4.tcp_siad_num_ms*. These sysctl settings are only considered at connection set-up. To influence the congestion epoch during the transmission, a new TCP socket option called *TCP_SIAD_NUM_RTT* is available. With this socket option the configuration can basically be changed with every ACK (in Congestion Avoidance). Only if this option is reset to zero, the congestion epoch length is updated at the next congestion event based on the default values, as only at this point an average RTT value can easily be estimated. Further, at the beginning of a connection *num_rtt* or, if configured, *net.ipv4.tcp_siad_num_rtt* is always used as no average RTT for a congestion epoch could be estimated yet to use the value provide by *num_ms* or *net.ipv4.tcp_siad_num_ms*. The patches that implement this new sysctl parameters and socket option are shown in the appendix A.2.

In the following sections we show in detail which state information need to be maintained, as well as how the per-packet delay is estimated and minimum delay is updated. Further, we explain implementation specific details on the increase and decrease process.

3.3.1 Maintaining State Information

Listing 3.1 shows the c struct used by the TCP SIAD implementation to access the reserved memory space for the congestion control module. This struct holds all state information needed for SIAD processing, using the available 64 Bytes/512 bits that are allocated for the congestion control state for each socket (anyway).

Listing 3.1: SIAD struct.

```c
struct siad {
  int config_num_rtt;      // configured Num_RTT by TCP_SIAD_NUM_RTT
  u32 default_num_rtt;     // default Num_RTT value
  u32 default_num_ms;      // default Num_ms value
  u32 curr_num_rtt;        // current calculated Num_RTT

  u32 increase;            // = alpha * curr_num_rtt
  u32 prev_max_cwnd;       // estimated maximum cwnd
  u32 incthresh;           // Linear Increment threshold

  u32 prior_snd_una;       // ACK number of previous ACK

  u32 prev_delay;          // delay value of previous sample
  u32 curr_delay;          // filtered current delay value
  u32 min_delay;           // absolute minimum delay
  u32 curr_min_delay;      // minimum delay since last congestion event
  u32 dec_cnt;             // number of additional decreases
  u8  min_delay_seen;      // state variable
  u8  increase_performed;  // state variable
  u16 prev_min_delay1,     // previous min_delay values if
      prev_min_delay2,     // monotonously increasing values
      prev_min_delay3;     // due to measurement errors
};
```

`config_num_rtt` must be the first element in this struct as we directly overwrite this part of the memory space when the *TCP_SIAD_NUM_RTT* socket option is set and SIAD is configured to be used. If `config_num_rtt` is set (larger than zero), `curr_num_rtt` is equal to `config_num_rtt`. Otherwise we directly set `curr_num_rtt` to `default_num_rtt` or a respectively calculated value based on `default_num_ms` and an average RTT for the last congestion epoch, if larger than `default_num_rtt`. Both default values `default_num_rtt` and `default_num_ms` are set at connection set-up to either the module default values *num_rtt* and *num_ms* or based on the sysctl parameter values if configured. This means if the socket option is not set but `default_num_ms` is larger than zero, `curr_num_rtt` will be dynamically calculated at each congestion epoch. Note `default_num_rtt` cannot be smaller than 2.

We do not store the actual value of α directly but a variable `increase`. In Linear Increment `increase` matches the number of packets between *incthresh* and *ssthresh*. Therefore, α is

$$\alpha = \frac{\texttt{increase}}{\texttt{curr\_num\_rtt}}. \tag{3.7}$$

This means `curr_num_rtt` determines the resolution of α which is sensible as we always want to increase by a full number of packets (`increase` = *incthresh* - *ssthresh*) per congestion epoch. When increasing on ACK receipt by $\alpha/cwnd$, we perform the respective calculation based on `increase`, `curr_num_rtt`, and the current congestion window value *cwnd* for each increase as further explained below in Section 3.3.3 Thereby α is automatically adapted in case `curr_num_rtt` changes. As a side remark, if `curr_num_rtt` could be restricted to a power of two, the implementation complexity could be largely reduced to a shift operator of `increase`. On initialization in `tcp_siad_init()` we set `increase` to `snd_cwnd`·`curr_num_rtt` and `incthresh` to `snd_cwnd` to a achieve an increase behavior similar to Slow Start as shown in Appendix A.1.

Further, we need to store the maximum estimated congestion window at the congestion event $cwnd_{max}$ in `prev_max_cwnd` to be used as $prev\_cwnd_{max}$ for the trend calculation at the next congestion event. `prev_max_cwnd` is set to `snd_cwnd` at connection initialization.

To compensate for delayed ACKs we estimate the number of acknowledged packets as described in Section 3.3.3 below. Therefore, we remember the acknowledgement number of the previously received ACK `prior_snd_una` in Congestion Avoidance. `prior_snd_una` is initialized as well as reset after recovery to `snd_una`, a variable in the TCP socket memory indicating the first unacknowledged byte.

`prev_delay` is the unfiltered delay sample of the previous ACK. The minimum between this value and the current delay sample is the current (filtered) delay `curr_delay` that is used for the decrease calculation. Note that TSOpt provides one RTT sample per ACK. If TSOpt is not supported, Linux will sample RTT measurements which leads to a certain inaccuracy but still allows TCP SIAD to work.

`min_delay` is the minimum delay of the total connection (not only this congestion epoch). Note that `min_delay` is updated if either a smaller sample was recognized or the delay stays constant for several RTTs. In contrast `curr_min_delay` is the minimum delay that was observed within this congestion epoch and will be reset after every congestion event.

`min_delay_seen` is a state variable to decide if Additional Decrease should be performed. `min_delay_seen` is set to zero after every decrease. Only if `min_delay` is updated by a new minimum value or if the same `curr_min_delay` value was observed in subsequent RTTs within one congestion epoch, `min_delay_seen` is set to one. Otherwise if `min_delay_seen` is still zero about one RTT after the decrease, Additional Decreased is performed. To avoid timer handling, we compare the RTT when the congestion window reaches $ssthresh + \alpha + 1$, which is about one RTT after the window reduction, to the so far seen `curr_min_delay` value and respectively set `min_delay_seen`. As soon as the congestion window grows by one more packet ($ssthresh + \alpha + 2$) we decide about performing Additional Decrease based on the state of `min_delay_seen`.

Further, we need to remember the number of additional decreases which have been performed already during the current congestion epoch `dec_cnt` to recalculate α or limit the maximum number of additional decreases. To avoid unnecessary window halving if another congestion notification is received right after a congestion event, we require the congestion window to be increased at least once first. This state information is stored in `increase_performed`. Moreover, the last three samples of the minimum RTT estimate are stored in `prev_min_delay1`, `prev_min_delay2`, and `prev_min_delay3` as the value needs to be reset if a monotonous increase is detected. Otherwise these values are set to zero as on initialization.

We initialize `curr_delay`, `dec_cnt`, `min_delay_seen`, and `increase_performed` to zero and set initially `min_delay`, `curr_min_delay`, as well as `curr_delay` to the maximum value `INT_MAX`, which is the maximum value of an unsigned 32 bit variable. Of course `curr_min_delay`, `dec_cnt`, `min_delay_seen`, and `increase_performed` have to be reset to either `INT_MAX` or zero after each recovery phase. The initialization method `tcp_siad_init(..)` as well as `tcp_siad_cwnd_event(..)` perform these actions as shown in Appendix A.

3.3.2 Delay Estimation and Filtering

`tcp_siad_cong_avoid()` is called for each ACK that is received in Congestion Avoidance and for each call we retrieve one RTT measurement sample. Note the larger the increase step size α and/or the smaller the congestion window, the fewer samples are available to observe the minimum RTT. However, only if TSOpt is used, a new RTT measurement can be accessed for each ACK. With TSOpt the ACK reflects a time stamp that was added to the data packet it acknowledges. By comparing this time stamp to the current system, the RTT can be estimated. Note that this estimation can include additional processing delays, e.g., if delayed ACKs are used or a duplicated ACK is received, the time stamp of the first packet is reflected. In case of delayed ACK if no second packet is received that could trigger the ACK, a timer is used to delay the ACK not more than 100 ms. This can lead to a single samples that indicates a too high RTT. Therefore, we always used for the current RTT measurement RTT_{curr} the minimum of the last two samples to filter out single outliers that might be caused by the used feedback strategy. If TSOpt, however, is not available, we used the smoothed RTT (SRTT) provided by the Linux TCP implementation, e.g., for RTO. SRTT is calculated by a EWMA as

$$SRTT \leftarrow (1-\alpha)\cdot SRTT + \alpha \cdot R' \tag{3.8}$$

where R' is the current sample and α a weighting factor of $\frac{1}{8}$ [115].

To decide if Additional Decrease should be performed, we check if the minimum RTT could be observed after the window reduction. If we measure the same or a smaller value as stored in `min_delay` during the subsequent RTT, we do not have to perform Additional Decrease. Further, if the delay did not increase during this first RTT after the end of recovery phase, even though we have increased the sending rate, we also do not perform Additional Decrease but update our minimum delay value. To avoid timer handling in our implementation we simply check if `curr_min_delay` or a smaller value could be observed again as soon as the congestion window has grown above `snd_ssthresh+(increase/curr_num_rtt)`. Only if `curr_min_delay` could not be observed again and the congestion window grows larger than `snd_ssthresh+(increase/curr_num_rtt)+2`, we perform an additional decrease.

Listing 3.2: Minimum delay estimation.

```
if (siad->min_delay == INT_MAX || delay <= siad->min_delay ) {
    // initialize total min delay or set to smaller value
    siad->min_delay = delay;
    siad->min_delay_seen = 1;
    siad->curr_min_delay = delay;
} else if (delay <= siad->curr_min_delay) {
    // update current minimum
    siad->curr_min_delay = delay;
    if (tp->snd_cwnd > tp->snd_ssthresh +
                (siad->increase/siad->curr_num_rtt) + 2) {
        // reset as same minimum over several RTTs
        siad->min_delay = delay;
        siad->min_delay_seen = 1;
   }
}
```

3.3.3 Linear Increase Calculation

Listing 3.3 shows the implementation of the increase function in `tcp_cong.c` as used by, e.g., TCP Reno. In this implementation `snd_cwnd_cnt` is increased by one for each ACK or the congestion window is increased by one if `snd_cwnd_cnt` has grown larger than the current congestion window value `snd_cwnd`. Note the congestion window is only increased if not application-limited (if smaller than *snd_cwnd_clamp*). Therefore, it is only increased if there is enough data in the socket buffer to fill the congestion window. If the congestion window is increased, `snd_cwnd_cnt` is reset to zero.

Listing 3.3: Increase behavior as implemented in tcp_cong.c.

```
if (tp->snd_cwnd_cnt >= tp->snd_cwnd) {
    if (tp->snd_cwnd < tp->snd_cwnd_clamp)
        tp->snd_cwnd++;
    tp->snd_cwnd_cnt = 0;
} else {
    tp->snd_cwnd_cnt++;
}
```

However, this implementation does only increase by one for each ACK triggers an increase in congestion window and does not use the acknowledged bytes to estimated how much packets where acknowledged by this ACK. Therefore, with delayed ACKs this implementation increases slower than defined in the originally TCP Reno increase algorithm. If only one acknowledgement is sent for two packets, this implementation increases the congestion window by half a packet per RTT or, more exactly, it increases by one packet every two RTTs as the window can only be increased by full packets. As with this implementation the window is only increased every second RTT, this actually leads to de-synchronization of competing TCP flows. This is because the congestion window increase itself often leads to loss of one packet, as after a window increase two packets will be sent at once: one for the received ACK and one for the increase itself. If only one flow increase in one RTT, only this one flows sees a loss in this RTT. Consequently, it reduces its congestion window and the congestion situation is already resolved before the competing flow increases its window the next time in the subsequent RTT.

To achieve the desired congestion epoch length in TCP SIAD, we, however, need to take the number of acknowledged packets into account. Note that while the congestion window is maintained in the Linux kernel in number of packets, a (delayed) ACK only provides the current acknowledgement number. Based on the previous and current acknowledgement number the number of acknowledged bytes can be calculated. Therefore, we store the previous acknowledgement number in `prior_snd_una`. `prior_snd_una` is updated with every ACK even if no increase is performed, e.g., due to Additional Decrease or if the transmission is application-limited. To estimate the number of acknowledged packets `acked_pkts`, however, we assume that full-sized packets were originally sent.

Listing 3.4: Estimation of the number of acknowledged packets.

```
u32 bytes_acked = ack - siad->prior_snd_una;
siad->prior_snd_una = ack;
u32 acked_pkts = bytes_acked / tp->mss_cache;
if (bytes_acked % tp->mss_cache || acked_pkts == 0)
    acked_pkts++;
```

Consequently, we do not increase `snd_cwnd_cnt` by one for each ACK, but by the estimated number of acknowledged packets `acked_pkts`. Further, we always account all acknowledged packet while in the implementation in Listing 3.3) an ACK that actually leads to an congestion window increase is (wrongly) not accounted at all. Therefore, `snd_cwnd_cnt` is increased before the `if` condition is entered, as shown below. The value `next` that `snd_cwnd_cnt` must reach to trigger a congestion window increase is dynamically calculated based on `increase` as our increase rate is not fixed (other than with TCP New Reno). The congestion window can be increased by more than one packet at once, e.g., if delayed ACKs are used where each ACK acknowledges several packets and TCP SIAD increases its sending rate with the maximum allowed increase rate as in Slow Start. Note even if the congestion window cannot be increased by the calculated number of packets `n`, e.g., due to application limitation, `snd_cwnd_cnt` is reduced by the desired increase $n \cdot$`next`.

Further, in `tcp_siad_cong_avoid()` in Fast Increase or Slow Start the increase rate is increased by one for each increase in congestion window. Note when entering Fast Increase the increase rate is reset to one and when leaving Slow Start α has to be recalculated if a valid Linear Increment threshold is given. Otherwise Fast Increase is entered directly. If Additional

Decrease is triggered, no increase processing is performed in `tcp_siad_cong_avoid()` as it can be seen in Appendix A.

Listing 3.5: Increase behavior as implemented in TCP SIAD.

```
tp->snd_cwnd_cnt += acked_pkts; // compensate for delayed ACKs
u32 next = max(1, tp->snd_cwnd * siad->curr_num_rtt / siad->increase);
if (tp->snd_cwnd_cnt >= next) {
    int n = tp->snd_cwnd_cnt / next;
    if (tp->snd_cwnd < tp->snd_cwnd_clamp)
        tp->snd_cwnd += min(acked_pkts,
                min(n, tp->snd_cwnd_clamp - tp->snd_cwnd));
    tp->snd_cwnd_cnt -= n * next;
}
```

The implementation of `tcp_siad_ssthresh()` is also shown in Appendix A. It implements, straight forward, the α calculation and the decrease as well as, if applicable, updates `curr_num_rtt` or resets `curr_min_delay`.

3.4 Summary and Discussion

TCP SIAD is designed to support high-speed networks as well as low latency requirements of emerging services based on the design goals as stated in Section 1.4. Therefore, five algorithms have be implemented, namely Scalable Increase, Adaptive Decrease, Fast Increase, Additional Decrease, and Trend calculation.

While Scalable Increase addresses the design goal of providing a fixed and configurable feedback rate and thereby provides scalability, Adaptive Decrease is designed to maintain high link utilization, even with smaller buffers. The average throughput of these two algorithms in isolation in steady state can be derived based on the response function for AIMD algorithm following the same approach as in [106]. Figure 3.3 illustrates the congestion window evolution of one TCP SIAD flow in steady state. Similar as every AIMD scheme it induces periodic network feedback (based on loss or ECN marks) that will trigger a window reduction at the end of each congestion epoch (assuming no additional source of irritation). Each congestion epoch therefore delivers $(cwnd_{min} \cdot Num_{RTT}) + \frac{1}{2}(cwnd_{max} - cwnd_{min}) \cdot Num_{RTT})$ packets. Further, assuming a constant probability p that a congestion event occurs in steady state, it follows

$$(cwnd_{min} \cdot Num_{RTT}) + \frac{1}{2}(cwnd_{max} - cwnd_{min}) \cdot Num_{RTT}) = \frac{1}{p}. \tag{3.9}$$

With $cwnd_{min} = \beta cwnd_{max}$ TCP SIAD achieves an average throughput $B(p)$ in packets per RTT of

$$B(p)[pkt/RTT] = \frac{2}{(\beta + 1)p}. \tag{3.10}$$

Note, the scalability problem of TCP NewReno is based in its response function $\sqrt{\frac{2}{3p}}$ where the feedback rate becomes lower with the square of the bandwidth. As also explicitly desired by, e.g,.E-TCP [57] and Relentless TCP [102], scalability requires a response function that is proportional to $\frac{1}{p}$. TCP SIAD therefore provides scalability and also is capable to provide high

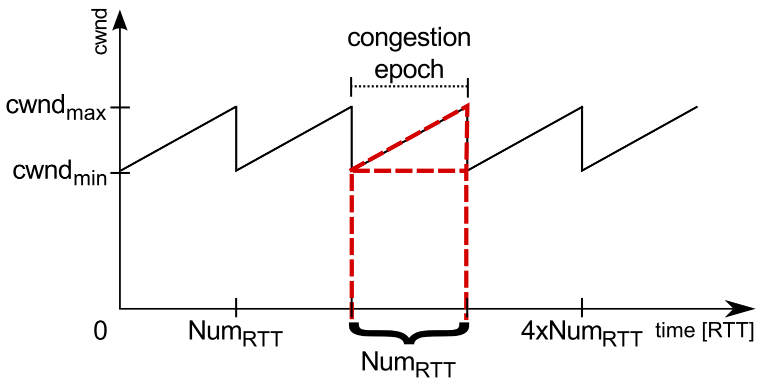


Figure 3.3: Congestion window evolution under periodic feedback.

throughput with small buffers; using a completely different approach than E-TCP or Relentless TCP.

For support of the low latency requirement of emerging services, operators must maintain low queue fill levels in the Internet. To still achieve high link utilization, congestion control needs to implement a smaller decrease factor than usually used today. To avoid a standing queue at the same time, the decrease factor needs to be calculated adaptively as by Adaptive Decrease in TCP SIAD. While Adaptive Decrease only empties the queue in the rare case where all competing flows are synchronized, Additional Decrease aims to empty the queue at every decrease. Therefore, only the implementation of both Adaptive Decrease and Additional Decrease avoids a standing queue that minimizes the average queuing delay.

Additionally, Fast Increase partly compensates eventual too large decreases. However, in general Fast Increase provides a fast allocation of newly available resources as needed in high-speed networks. We believe that any efficient high-speed scheme must be able to distinguish between stable state and changing network conditions and therefore needs to implement a separate Fast Increase phase.

Finally, Trend calculation supports convergence with Scalable Increase which, of course, is a general requirement for congestion control. In the following chapter, we further evaluate in simulations the convergence properties of each of the algorithms as used in TCP SIAD separately and in different combinations to obtain further insight on interactions of the different components.

To elaborate the space for further research and development based on TCP SIAD, in the following we summarize degrees of freedom in TCP SIAD's design as already laid-out above in Section 3.1. Subsequently we discuss implementation complexity and provide guidance for buffer sizing and parameterization with use of TCP SIAD.

3.4.1 Degrees of Freedom in Design

Even though TCP SIAD as previously presented is composed in this way to address the given design goals, each of the five introduced algorithms provides some degrees of freedom in its design as summarized next.

Length and strength of Trend In Trend calculation the length of the history that is taken into account as well as the strength of the trend itself can be varied to influence the convergence properties of TCP SIAD. Therefore, we also tested cases where the last two maximum congestion window values are taken into account or where the trend is only half as strong as in the proposed implementation. In general we found that the trend calculation influences the trade-off between convergence time and oscillation size. While the current Trend calculation can be slow in a scenario where a second flow starts while a first flow is already fully utilizing the link, faster approaches induced higher oscillations. As we do not explicitly target fast convergence in such a scenario as a required improvement to current state-of-the art mechanisms, we decided to not further optimize Trend calculation for this case. Moreover, we experimented with random (small) offsets added to the calculated Trend. This addition still provides convergence and is potentially more stable but also slower. Therefore, we chose the described setting as a good compromise.

Decrease size of Additional Decrease Additional Decrease aims to empty the queue. Determining the optimal decrease size requires knowing the number of synchronized competing flows. This is nearly impossible for a distributed approach as end-to-end congestion control. Therefore, different decrease strategies can be implemented for Additional Decrease. We decided to require that it must be possible to empty the buffer with the maximum number of Additional Decreases. Instead, one could, e.g., also try to minimize the number of Additional Decreases by estimating the total queue size (e.g., based on packet inter-arrival times). However, such a strategy might lead more often to link underutilization if other decreases are performed synchronously. Therefore, the design of Additional Decrease needs to maintain a trade-off between low average queuing delay and link utilization.

Minimum increase rate TCP SIAD implements a lower and upper limit for the minimum and, respectively, maximum increase rate per RTT. Both influence the convergence time as well as stability regarding oscillation size and frequency. We chose a minimum increase of one packet per RTT, similar to the increase rate of TCP NewReno. However, we expect future network configurations (regarding buffer size and feedback rate) that lead to an increase rate of more than one packet per RTT, e.g. 40 packets of buffering with a Num_{RTT} configuration of 20 would lead to an increase rate of two packets per RTT. On a 100 Mbit/s link 40 packets of buffering is equal to about 5 ms of maximum queuing delay. Note, whenever Num_{RTT} is too large or the buffer share too small, all competing flows increase their sending rate with the same (minimum) increase rate and therefore share resources equally among each other and also with TCP NewReno-like traffic.
Further, an increase rate of one packet per RTT is the minimum granularity of change that results in noticeable action. Only if the congestion window is increased by a full packet, more packets are actually sent out. This minimum sending rate therefore guarantees that in each control interval of our control loop a change in the input signal is applied and therefore a resulting change in the network output response would be measurable. We also tested a smaller minimum increase rate of $1/Num_{RTT}$ per RTT according to the minimum resolution in our implementation. Thereby, we could achieve a finer grained capacity sharing but this also leads to longer periods in de-converged states. Therefore, we chose one packet per RTT is a reasonable minimum rate that helps smoothness and convergence.

Increase rate in Fast Increase In Fast Increase we increase the increase rate. We decided to double the increase per RTT as we can replicate the Slow Start behavior with this increase function. While doubling the congestion window in Slow Start, we only allow a maximum increase rate of $1.5 \cdot cwnd$ in Fast Increase. Simulative evaluation still showed fast convergence but lower oscillation for this maximum increase rate than when doubling. In any case, the maximum increase rate should be a multiplicative of the current congestion window *cwnd* to address scalability.

RTT estimation Adaptive Decrease calculates the decrease factor based on the current RTT and base RTT. To filter out single outliers we use the minimum of the last two samples (before the congestion notification was received). However, RTT measurements are known to be error-prone due to noise on the end-to-end path (e.g. additional processing delays). If the underlying network induces high additional delay variations, a more sophisticated filter could be used. However, in case of only small variations TCP SIAD would either perform a slightly too large or too small decrease. Both cases are not problematic for the operation of TCP SIAD due to the addition of Fast Increase and Additional Decrease. Therefore, we argue that a simple minimum filter over the last two samples is sufficient for most Internet usage scenarios.
In high-speed networks the limited resolution of TSOpt can be too low to recognize small queuing delays and Additional Decrease might not be performed. While the timestamp resolution might improve in the future, an alternative would be to signal explicitly when the queue is empty. Both approaches negotiating the time stamp resolution [129] and changing the semantic of ECN [27] to signal further information are under consideration in standardization in the IETF.

3.4.2 Implementation Complexity

We have implemented TCP SIAD as Linux congestion control module. As TCP SIAD is still an AIMD scheme, this does not require any further modifications in the kernel. Only if the application also wants to use the introduced sysctls, namely *net.ipv4.tcp_siad_num_rtt* and *net.ipv4.tcp_siad_num_ms*, or the socket option *TCP_SIAD_NUM_RTT*, additionally kernel modifications are needed.

All computations are performed on ACK reception (or very rarely on time-outs). However, only a few ACKs actually trigger a congestion event. If a congestion notification is received, we re-calculate α and β. In this case, the sending rate will be reduced anyway and therefore processing time is not critical. Usually, we perform 2-3 multiplications and 2-3 divisions in `tcp_siad_ssthresh(...)` (or less otherwise). In Linear Increment and Fast Increase we perform 2-3 divisions and 1-3 multiplications. If instead Additional Decrease is performed, we have 2-6 divisions and 3-4 multiplications. However, this happens only a few times per congestion epoch, if at all, and again the sending rate is reduced in this case anyway.

As we decided to store `increase` instead of α, as explained above, we have to calculate $\alpha = \texttt{increase} \cdot Num_{RTT}$ for each ACK. If further computational optimization is needed, the number of divisions and multiplications could be reduced by only allowing powers of 2 for Num_{RTT} as the calculation of α accounts for most of the operations. However, similar as

TCP NewReno, we anyway have to calculate $\alpha/cwnd$ for each ACK which cannot be further optimized.

In comparison TCP Cubic performs 5-6 multiplications and 2-4 divisions per ACK in Congestion Avoidance and 1-3 multiplications and 1-2 divisions in `tcp_siad_ssthresh(...)`. Thus, the implementation of TCP SIAD is not more complex than of existing and deployed congestion control schemes and we therefore do not expect performance limitations due to processing overhead.

3.4.3 Buffer Sizing and Configuration

Finally, for the application of TCP SIAD in the future Internet there are two question on parameterization that need to be answered:

1. How to configure the network buffer size?
2. How to choose the right feedback rate?

While the parameter that influences the feedback rate, namely Num_{RTT} or Num_{ms}, is under control of the end-system that implements TCP SIAD, the buffer size is a configuration parameter that is set by the operator. Therefore, TCP SIAD is designed to work well with arbitrary buffer size, However, there are limits on minimum buffer size (that are valid for any congestion control). The configuration of the buffer size is to some extent independent of congestion control. The network buffer is needed to handle the simultaneous arrival of multiple packets in a multiplexed network. Therefore, the minimum buffer size depends on the multiplexing factor. In addition to avoid early window reductions for small flows, the buffer must cover at least the initial congestion window that was recently increased to 10 packets [35]. To avoid an early termination of Slow Start, $\frac{1}{2}\cdot$BDP of buffering would be needed [67]. However, if Slow Start is left early, TCP SIAD enters Fast Increase at the end of the first congestion epoch and therefore can allocate the remaining capacity relatively quickly. Especially for short flows it would still be beneficial to implement larger buffers but configure a shallow ECN marking threshold. By smoothing the feedback signal either in the network or in the end-host, a starting flow could stay longer in Slow Start without causing loss. However, if the maximum buffer size is large, this could also cause a high maximum end-to-end delay. Therefore, a network operator has to base its decision on the buffer configuration on the type of service (s)he wants to provide. When in future networks smaller buffers are configured to support emerging low latency services, TCP SIAD implementing an adaptive decrease and fast increase behavior will help to keep link utilization high.

When using TCP SIAD, Num_{RTT} can be set dynamically during a transmission by a higher-layer control loop to better cope with the requirements of a specific application. However, we expect that in most cases, and always at the beginning of a connection, a default value is used. This default value should be derived from typical Internet usage scenarios, e.g., with respect to the congestion feedback rate that TCP NewReno or TCP Cubic induces if maintaining, e.g., half the share of the current access link capacity. If then one TCP SIAD would compete on the access link with one TCP NewReno or TCP Cubic, they would share the available resources about equally. Moreover, the default value should not be chosen too large, if high responsiveness is desired. As Num_{RTT} determines length of a congestion epoch, it also influences strongly the maximum time needed to detect spare capacity.

Additionally, some applications might implement a mechanism to control the aggressiveness dynamically, e.g., for real-time video where the service needs a certain minimum rate to work at all. In this case the aggressiveness can be increased, at least for a certain time (until the congestion allowance at the policer, as described in the introduction in Section 1.1.3, is consumed) to grab a larger share than reached by the default parameterization. Note, if Num_{RTT} is selected too large and the bottleneck buffer is small, TCP SIAD operates at the minimum increase rate of 1 packet per RTT and therefore changes in Num_{RTT} do not have any effect on the capacity sharing until Num_{RTT} is small enough again.

To maintain stability the higher-layer control loop needs to work with a lower control frequency than the control frequency of the congestion control algorithm itself which is determined by the feedback frequency based on Num_{RTT} and therefore known by the application. A development of a higher-layer control loop is out of scope for this work as this would need to be integrated into the application processing and based on specific application requirements.

Note there is currently no mechanism that keeps single flows from using a more aggressive congestion control and thereby push away other traffic. In fact, TCP Cubic is already more aggressive than TCP NewReno and is the default configuration in Linux. Further, many applications open several simultaneous connections to achieve higher throughput. Also, competing TCP NewReno flows operating on different end-to-end delay cannot share the available resources equally. In Chapter 4 we show that equal sharing between competing TCP SIAD flows can be achieved, however, we do not expect that equal sharing is desired anymore in future as long as every flow is able to grab sufficient capacity to fulfill the application's requirements. Therefore, fairness should not be based on the instantaneous share of a single flow but by on a per-user basis over longer time scales. Such a definition of fairness, however, cannot be addressed solely by the congestion control in a highly distributed system such as the Internet but must to be addressed in the future Internet by (ingress) policing, e.g., based on the induced congestion rate (see introduction). Therefore, we argue that it is important to have a configuration possibility in congestion control as implemented by TCP SIAD providing the basis for the deployment of per-user congestion policing.

4 Evaluation

In this section we present an evaluation of TCP SIAD based on simulations integrating the Linux kernel and our Linux implementation of TCP SIAD, as described in the previous chapter. The simulations performed are aligned with the requirements as stated in the introduction in section 1.2: scalability, adaptivity, capacity sharing, and convergence. Further, we assess the robustness of TCP SIAD to (non-congestion) loss and dynamic network conditions. Even though this was not explicitly stated as a requirement for congestion control design, this is important for future deployment in the Internet.

In the following section we describe the used simulation setup and scenarios. For most simulations we use a simple dumbbell scenario and configure different conditions at the bottleneck link and queue, as they can be found in real networks, such as different link speed, queue size, or dynamics in speed and delay. Further, we can impact the traffic load by using different traffic models and vary the number of concurrent flows. Additionally, we evaluate one scenario with multiple bottlenecks. Further, we introduce and discuss the used metrics for evaluation of the stated requirements and design goals, as listed in section 1.4.

As a first stage of evaluation, we investigate the different components of TCP SIAD independent of each other or in combinations of two or three of them. We assess the contribution of each component to reach the desired design goals, and foremost discuss the influence on convergence of each component. This is important as the Scalable Increase approach itself does not provide convergence anymore. Further, the resulting capacity sharing of the combination of the five introduced algorithms cannot be assessed easily analytically, and therefore it is important to first understand the influence of each single component.

Section 4.3 focuses on single flow behavior and compares the steady state behavior of TCP SIAD to other state-of-the-art high-speed congestion control proposals. The evaluation of this static, single flow scenario provides initial insights on adaptivity and scalability properties of TCP SIAD. We show that TCP SIAD always reaches high link utilization and avoids a standing queue as well as induces a fixed feedback rate independent of the bottleneck bandwidth and therefore achieves the respective design goals in these first scenarios. Further, we assess the ability of a single TCP SIAD flow to handle delay variations or rate variations at the bottleneck link to evaluate the influence of these network effects independent of other traffic dynamics.

In Section 4.4, we demonstrate the viability of the SIAD principle in multiple flow scenarios and TCP SIAD's impact on capacity sharing depending on the configured feedback rate. We evaluate scenarios with competing flows that either implement the same or a different congestion control scheme and either operate based on the same or different network conditions, such as different end-to-end delays or a different number of bottlenecks that need to be passed by one of the flows in test. We demonstrate that the configuration of different Num_{RTT} values impacts capacity sharing as desired.

Subsequently, we assess the convergence properties of TCP SIAD in Section 4.5. There are basically two cases for evaluation, either where the available bandwidth increases or decreases due to changes in the network (e.g. re-routing) or dynamics in the traffic load by stopping or starting flows with adaptive or constant bandwidth allocation. Further, it is of interest to investigate the responsiveness of an existing flow in steady state, if a new flow starts. This means the bottleneck link is already under full load and the convergence time is strongly influenced by the ability of the first flow to release capacity. This scenario is not covered by one of our design goals but provides insights on TCP SIAD's behavior in comparison to the other schemes in test.

To prove the applicability of using TCP SIAD in the Internet we also show TCP SIAD's robustness in high congestion or high loss scenarios in Section 4.6 where TCP SIAD always reaches a high link utilization.

All in all, we show that only TCP SIAD is able to fulfill all design goals achieving high utilization with arbitrary buffer sizes and always avoiding a standing queue. Further, only TCP SIAD and Scalable TCP implement a fixed feedback rate independent of the link speed while Scalable TCP reaches this on the cost of inducing a standing queue and high loss rate. Moreover, we demonstrate the capacity sharing properties of SIAD depending on the configured feedback rate and show that TCP SIAD allocates newly available capacity quickly and converges reasonably fast. In addition, TCP SIAD provides a much higher resilience to non-congestion losses than all other schemes in test.

4.1 Simulation Setup

Our evaluation is based on simulations. We used the event-driven network simulator IKR SimLib [3] together with the IKR VMSimInt [146] extension which provides integration of virtual machines into the simulated network. Therefore, we can run unmodified Linux kernel code in simulation and apply our TCP SIAD Linux implementation. This setup provides realistic TCP behavior and a proof of concept of implementation feasibility. Further, we can compare TCP SIAD to the existing implementations of other schemes in Linux. Moreover, by using a simulated network instead of a real test-bed, we are able to fully control the network conditions and therefore eliminate unwanted sources of disturbance. Therefore, a controlled, simulated network environment provides best conditions for a detailed evaluation of the microscopic algorithm behavior and to show the robustness in a large set of well-defined (extreme) scenarios. IKR VMSimInt provides comparable results to the ns-3 simulation environment, a well-known network simulator for TCP evaluations, [146] and additionally allows an easy use of unmodified and recent kernel code.

We mostly use the default Linux configuration with Selective Acknowledgment (SACK) and TSOpt enabled. Note, in the used kernel version 3.5.7 an initial congestion window of 10 packets is implemented which of course as well is used for TCP SIAD. However, we implemented two new sysctl parameters (additionally to the two to configure Num_{RTT} and Num_{ms}) to dis-

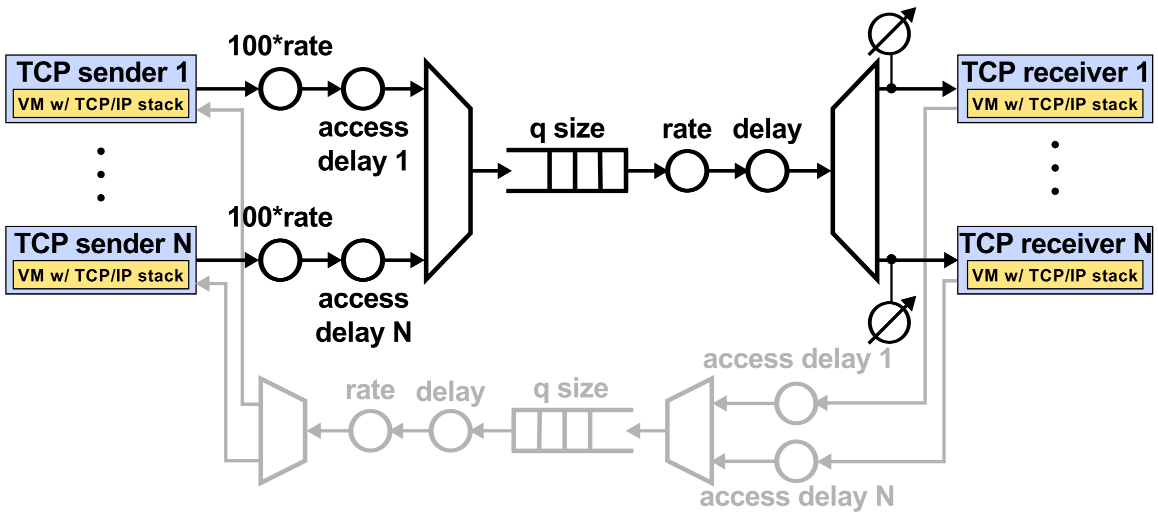


Figure 4.1: Simulation setup.

able delayed ACKs and set the Slow Start threshold as needed for certain evaluation scenarios. The respective source code for these Linux patches is shown in Appendix A.2. Moreover, to avoiding window limitation due to memory restrictions we explicitly set the respective sysctl parameters based on the configured memory for the virtual machine. The memory of the virtual machine is always set to $32 \cdot max(q, BDP) \cdot n + 24MB$ where q is maximum buffer size and n is the expected maximum number of flows. This configuration is chosen to provide enough space for the dummy data in user and in kernel space as well as for the operating system itself. Subsequently `net.core.wmem_max` and `net.core.rmem_max` are set to $16 \cdot max(q, BDP)$. For `net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem` we set the minimum, the default, and the maximum value to $16 \cdot max(q, BDP)$ as well. And for `net.ipv4.tcp_mem` we set the minimum and default to $16 \cdot max(q, BDP)$ but the maximum to $16 \cdot max(q, BDP) \cdot n$. Note, memory limitations can lead to slightly different behavior in similar scenarios as, e.g., early end of quick ACK phase at receiver side in Slow Start. However, those small variations in Slow Start do not invalidate our statistical evaluation.

4.1.1 Simulation Scenario and Traffic Generation

In all simulations we used a dumbbell topology as shown in Figure 4.1. In most scenarios we only used one or two sender/receiver host pairs where each sender and each receiver operates an own kernel images. Therefore, we can configure the congestion control scheme used on a per sender basis.

Only in the case of CBR traffic, the scenario is slightly different as we have one sender-receiver pair that does not generate TCP traffic but sends messages with a fixed rate.

Otherwise, each of the senders opens one or more connections to the respective receiver and sends dummy data. In our simulation the receiver does not send any payload data but only control data as ACKs. The dummy data is generated in the simulation tool and passed to a relay program in the virtual machine as described in Section 2.5.1. In the simple case we model a data-unlimited (greedy) source where there is always enough data available to fill the link.

Additionally, we can start and stop the data input for each connection at a certain time in the simulation. Note, if we stop a TCP connection, all data that is already in the VM waiting for processing is still sent before the connection is terminated. The length of the additional transmission time depends on the buffer configuration of the virtual machine. However, in our evaluation it either doesn't matter when exactly a transmission is terminated or the results are compared based on the same conditions for all schemes in test. Further, note that the convergence times we observed are in the order of seconds while the additional transmission time in our setup is a couple of RTTs.

To simulate short traffic bursts we generate data bursts based on configurable distributions of the burst size and the IAT. In this case we can either open an own TCP connection for each data burst or send all data over the same connection. If we open one connection per traffic burst, we always start the connection in Slow Start. In the latter case we only slow-start if the idle time between two bursts is large enough. In contrast if the IAT is too short, the traffic queues up and is sent successively. However, if the IAT is large enough both options implement the same behavior which is the case for all our simulations in Section 4.6.

Finally, all traffic passes a central bottleneck link. This link is modeled by a queue that can implement different AQM schemes, a rate limitation, and a link delay. The central link on the backward path is identically modeled and configured. As the traffic load is always smaller than on the forward path, this setup guarantees that the backwards path never is the bottleneck. If not stated differently, we use a DropTail scheme with a configurable maximum queue size and a link delay of 50 ms resulting in 100 ms base RTT (transmission time without queuing delay). The queue size is always represented as a multiple of the base BDP (when the queue is empty). We chose this delay configuration because typical RTTs in the Internet are between a few milliseconds to about 200 ms. We additionally performed further simulations with smaller RTT which are not shown in this report as it is sufficient to either vary the rate or the delay to demonstrate certain effects that occur with different link BDPs. Therefore, we decided to use a fixed RTT of 100 ms except for simulations where competing flows have different base RTTs.

Each sender is connected to the bottleneck link by an access link with a bandwidth limitation of 100 times the bottleneck bandwidth. Modeling the access link speed enables packet interleaving between traffic of different senders. There is no speed limitation on the receiver's access link as the sending rate and send-out timing is determined by the incoming traffic. Further, we can configure an additionally link delays on the access link. If set, the corresponding sender and receiver access link delays are always configured with the same additional delay value for both links. However, the access delay is always zero except for the simulation setup in Section 4.4.2 where the bottleneck link delay is set to zero instead.

As a basis for our evaluation we monitor the congestion window at each sender for each connection tracing each change at packet arrival or transmission. Further, we trace the queue length for each incoming packet and track if the packet is forwarded or dropped at the bottleneck. Additionally, we measure the throughput at receiver side with a resolution of $2 \cdot \mathrm{RTT}$. If not stated differently all simulations ran for 600 seconds. For statistical evaluations the first 20 seconds were not considered, as we only evaluate steady state behavior. In scenarios with long convergence times, we ensured that the simulation ran long enough to achieve convergence. This start-up phase has been chosen deliberately long as we have to ensure that it is long enough to exclude Slow Start and the initial convergence phase.

4.1.2 Metrics

We define the following metrics which are calculated based on the trace data output of each simulation. As these metrics directly refer to our design goals, they are listed in the same order as the respective design goals in Section 1.4 that aim for the implementation of high link utilization, minimized average queuing delay, a fixed feedback rate, quick capacity allocation, and a configurable aggressiveness.

Average link utilization The average link utilization is calculated based on the time that the link was occupied in relation to the time that the link was not occupied. A basic requirement for congestion control is efficiency; therefore we have an explicit design goal on high throughput. Only if the link utilization is close to 100 %, the throughput is maximized.

Average loss rate The average loss rate is calculated based on the number of bytes that were dropped at the queue in relation to the total number of bytes that were received at the queue. To finally assess high throughput one must not only consider link utilization but has to also assess the (average) loss as dropped packets will keep link utilization high but do not provide any useful communication. Note, that we also require a certain feedback rate where the feedback is provided by loss signals. While it is in general desirable to minimize the loss rate, we explicitly have a design goal that requires a certain (minimum) loss rate, as further detailed below.

Minimum and average queue fill fraction The minimum and average queue fill fraction is the minimum or average queue size in bytes normalized by the maximum configured queue size in bytes. The average queue size is calculated based on samples of the current queue length multiplied by the elapsed time since the last update and is updated with every en- or de-queue. The average queue length is compared in different scenarios and for different congestion control schemes to assess the influence of the used congestion control scheme on low latency services. We explicitly stated a design goal of minimizing the average queuing delay. Therefore, we do not explicitly evaluate the jitter that is induced which often is also important for low latency services. However, for all loss-based congestion control schemes the maximum queuing delay is determined by the queue length and therefore only the average can be influenced. The minimum queuing delay is used to rate if a standing queue was induced. If the minimum is never zero, there is a standing queue.

Average loss event distance and standard deviation The loss event distance is the time between the last loss of the previous congestion event and the first loss of the current congestion event where all losses that occur within $2 \cdot \text{RTT}$ are accounted for the same congestion event. The loss event distance can be estimated for only one flow or an aggregate of flows at one bottleneck queue. The loss event distance correlates to the feedback rate (in case of loss-based congestion control) which we aim to be constant (as configured) independent of the available bandwidth. We evaluate this metric in simple scenarios with different BDPs.

Average, minimum, and maximum convergence time We define the convergence time as the time that is elapsed until a flow reaches a given sending rate. This metric is also derived based on periodic sample rate values and therefore its resolution is limited by the sample interval. For convergence between two competing flows we regard the time until 80% or 95% of the equal sharing sending rate is reached. Based on the convergence time we will evaluate how quickly a flow can grab available bandwidth, as stated in our design goals. Further, the convergence time in a scenario with multiple competing flows also allows us to assess the ability of a flow to release capacity to other starting flows. To quickly converge in such a situation is not an explicit design goal, as already explained in the previous chapter, however, for comparison with existing proposals we also evaluate these kind of scenarios.

Average sending rate The average sending rate is the mean value of rate values samples that are calculated based on the number of received bytes in a fixed time internal. The average sending rate can be measured on a per flow basis or for a traffic aggregate at one point on the path. In addition, we calculate the ratio between two average sending rates to evaluate which capacity sharing is achieved. While we often state the average sending rate for various scenarios, we are mostly interested in the ratio between two (or multiple) flows as this assesses the achieved capacity sharing in certain scenarios.

Jain's Fairness Index As introduced in Section 2.5.3, Jain's fairness index is a measure for the equal distribution of a certain resource. For those scenarios where equal sharing is desired we calculate Jain's fairness index based on the average sending rate of a flow or a group of flows. If we assess equal capacity sharing of different groups of flows, we calculate the mean of the average sending rates of all flows in the group. Mostly we show that the aggressiveness configuration parameter can be tuned in different scenarios to achieve a high value of Jain's Fairness index and therefore demonstrate the ability of TCP SIAD to influence capacity sharing as stated in our design goals.

Average oscillation size and standard deviation For the oscillation size we isolate the minima and maxima from the congestion window trace. Subsequently we calculate the size of each decrease and each increase period by subtracting each minimum from the previous and subsequent maximum. This metric does not directly relate to a certain design goals but is used to assess smoothness and responsiveness in scenarios where multiple flows share a bottleneck. While smoothness might provide higher stability, a reasonable high responsiveness is needed to quickly reach convergence and allocate newly available bandwidth. We only discuss the oscillation in comparison to other schemes in test and in relation to additional metrics. E.g., a larger oscillation size is (only) bad if also the utilization is lower and/or the loss rate is higher than for the scheme in comparison.

Additional to the statistical metrics, we show congestion window and queue length traces over time to demonstrate the different characteristics of the congestion control schemes in evaluation.

Table B.1 in the Appendix B gives an overview about where which metrics are used to evaluate which requirement. In addition, Table B.2 summarizes the network and traffic parameters used in the discussed evaluation scenarios. These tables do not include Section 4.2 as this section is not part of the actual evaluation of the TCP SIAD algorithm but discusses an initial, preliminary assessment to provide a basic understanding of the characteristics of the individual algorithm components introduced in the previous chapter.

4.2 Stability and Convergence of Individual Algorithm Components

This section provides an independent, preliminary assessment of characteristics of the individual algorithms, namely Scalable Increase, Adaptive Decrease, Additional Decrease, Fast Increase, and the use of Trend. Based on the implementation of TCP SIAD as described in the previous section, we extracted or replaced the code elements of the individual algorithms or a combination of two or three of them and derived different implementation variants as described below. These variants are not supposed to be used as a stand-alone congestion control scheme but are only used for the evaluation in this section to assess a) if the respective (stand-alone) algorithm can achieve the desired design goal that it is aiming for and b) to demonstrate its convergence and capacity sharing properties. As convergence and the achieved capacity sharing ratio of one or more flows using the total TCP SIAD scheme cannot be easily assessed analytically the preliminary evaluation in the section helps to better understand the evaluation results of TCP SIAD on convergence and capacity sharing as presented in Sections 4.4 and 4.5.

More explicitly, we demonstrate the following observations in this section:

Scalable Increase

1. Flows using Scalable Increase induce the same loss event distance on different bandwidth links and therefore Scalable Increase provides a fixed feedback rate independent of the link capacity (scalability).
2. The sharing ratio of the link capacity of two competing flows using Scalable Increase in convergence state is random and therefore Scalable Increase does not converge to equal sharing.

Adaptive Decrease
Two competing and synchronized flows using Adaptive Decrease always just empty the queue on decrease on network paths with different queue sizes (adaptivity).

Additional Decrease
Two competing flows using Adaptive Decrease and Additional Decrease can also empty the queue even when the flows are not synchronized.

Trend
The introduction of Trend calculation re-introduces convergence to equal capacity sharing of two competing flows that use Scalable Increase.

We evaluate Fast Increase only in combination with Trend calculation. Without Trend when two competing flows operate in a stable state, Fast Increase has no influence and therefore these combinations are not demonstrated in this section. However, the Trend calculation introduces additional dynamics. In this case we demonstrate the influence of Fast Increase on the convergence speed. We do not demonstrate in this section the ability of Fast Increase to provide fast bandwidth allocation in situations with changing network conditions where the available bandwidth increases abruptly due to, e.g., route changes or stopping traffic. We further evaluate these kind of scenarios in Section 4.5 as this can nicely be demonstrated for the total TCP SIAD scheme and therefore does not need to be evaluated separately in this section.

We implemented and evaluate the following variants. The source code of all implemented variants can be found in Appendix A.3.

- **SI_only**: This variant only implements Scalable Increase with a minimum increase rate of 1 packet per RTT and a fixed decrease factor of 0.5.
- **Fixed_Increase**: To evaluate capacity sharing with different increases rates, we also implemented a variant with a (configurable) fixed increase rate and a (non-configurable) fixed decrease factor of 0.5.
- **SI_trend**: This variant implements the Trend calculation additionally to Scalable Increase as in SI_only.
- **SI_trend_fastIncrease**: This variant implements Scalable Increase, Trend, as well as Fast Increase. As increase factor can grow quite large with the use of Fast Increase, we also need to adapt the congestion window based on the most recently used increase factor before applying Adaptive Decrease. Note, we do not implement this additional adaptation with any other variant evaluated in this section for the sake of simplicity.
- **AD_only**: This variant only implements Adaptive Decrease with a minimum congestion window of $MIN\_CWND = 2$ and a fixed increase rate of 1 packet per RTT. If a congestion notification is received in Slow Start, this implementation still halves the window (as the increase rate was doubled in the last RTT of feedback delay).
- **Fixed_Decrease**: To evaluate an alternative decrease behavior as it could be used if the total queue size would be known, we implemented a variant with a fixed subtractive decrease of a configurable number of packets and a (non-configurable) fixed increase rate of 1 packet per RTT.
- **SIAD_only**: This variant performs Scalable Increase and Adaptive Decrease but without any additional mechanisms as Fast Increase, Trend, and Additional Decrease. As in AD_only we halve the congestion window in Slow Start before the regular, adaptive decrease is applied.
- **SIAD_addDecrease:** For this variant Scalable Increase and Adaptive Decrease, as explained above, together with Additional Decrease is implemented.
- **SIAD_trend:** This variant implements Scalable Increase and Adaptive Decrease as well as the Trend calculation; but neither Additional Decrease, nor Fast Increase.

Note, for all implemented variants that use Adaptive Decrease we always use the total minimum delay seen during the connection and never update to a larger value. This is a simplification that can be done in simulations as we know that the base delay does not change and therefore gives always the correct value. Furthermore, the delay used to calculate the decrease factor is the minimum of the last two samples before the congestion notification was received to filter single outliers.

To avoid the need to estimate the number of acknowledged packets we deactivated delayed ACK for the simulations in this section. This allows the implemented variants to achieve the desired increase rate and a minimum increase rate of 1 packet per RTT without additional complexity.

For all simulations in this section, to avoid unnecessary complexity, we used one simple scenario with four hosts and in total two TCP connections that have always enough data to send (greedy) as described above in 4.1. There are either two competing flows that start simultaneously or the second flow starts with a defined delay of some milliseconds compared to the first flow. Further, we always use a (symmetric) OWD of 50 ms. If not stated differently, we use a basic configuration of 10 Mbit/s link speed for the bottleneck and a buffer size of 0.5·BDP which are 62500 Bytes and therefore about 41 full-sized packets (of 1514 Bytes incl. Ethernet header). Note, as we always have data to send in these simulations we only have full sized data packets. If not noted differently, we configured a Num_{RTT} value of 20, if applicable (that means for Scalable Increase variants only). In addition, we show one simulation where $Num_{RTT} = 30$ to demonstrate that different feedback rates can be configured with Scalable Increase and one simulation where the two flows use different Num_{RTT} values

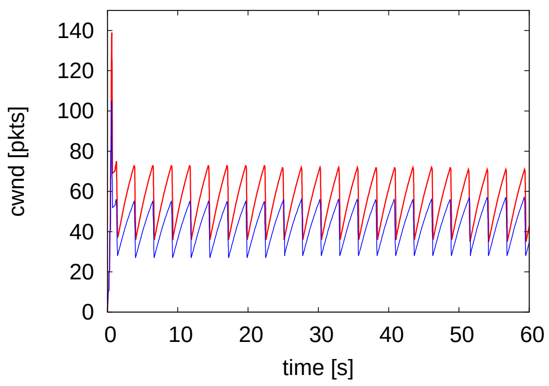
4.2.1 Fixed Feedback Rate and Convergence with the Use of Scalable Increase

In this section we evaluate the four variants that only modify the increase behavior but halve the congestion window on loss: SI_only, Fixed_Increase, SI_trend, and SI_trend_fastIncrease. We demonstrate the ability of Scalable Increase to induce a fixed feedback rate independent of the link bandwidth and show various scenarios where two competing flows converge to different capacity sharing ratios with and without the use of the Trend calculation. Further, we demonstrate the behavior of two competing flows with a fixed, non-adaptive increase rate. This implementation does not aim for a fixed feedback rate, but the achieved capacity sharing ratio of two competing flows operating with different fixed aggressiveness can be derived analytically. Therefore, this behavior provides the theoretical basis for our design goal that aims to provide a configurable aggressiveness that can be used to influence the capacity sharing.

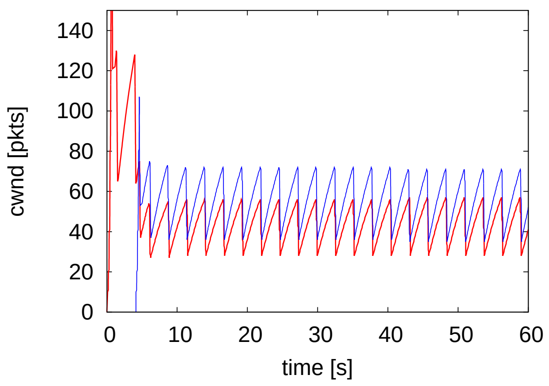
Scalable Increase

Figure 4.2 shows traces of the congestion window over time of the two competing flows, in this case for the SI_only variant. In Figure 4.2a both flows start simultaneously. Even though they are configured with the same Num_{RTT} value of 20, they do not converge to equal share. This is because SI will calculate different effective increase rates for both flows to maintain the same length of the congestion epoch and therefore reach the same target value as before in the configured time frame. The initial target value that determines the sharing ratio depends on the capacity sharing in the start-up phase. In fact we can achieve a random sharing just dependent on the start time of the second flow. E.g., Figure 4.2b shows a scenario where the second flow starts 4 seconds after the first. In this case the second flow obtains the larger share as the queue was nearly empty at starting time and therefore the second flow could grab a larger share due to increasing aggressively in Slow Start.

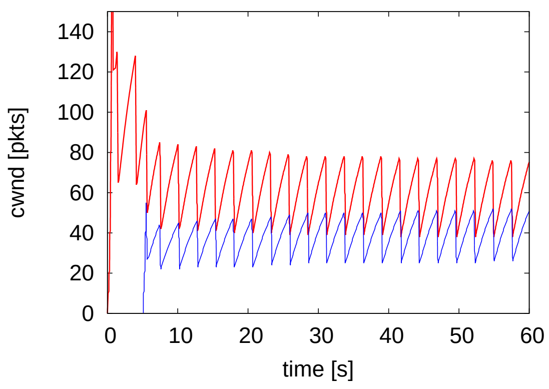
In Figures 4.2c and 4.2d, again the second flow gets the smaller share starting at 5 and 6 seconds. In case of the latter, the convergence period (until a stable state is reached) is quite long as the queue was already nearly filled by the first flow when the second flow was starting. Initially they converge to a certain share (until about 20 seconds) because of the implemented minimum increase rate of 1 packet per RTT.



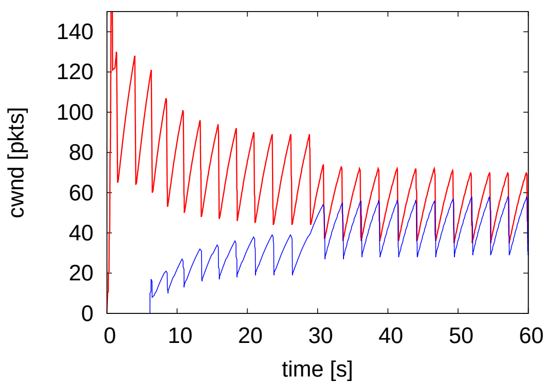
(a) Both flows start simultaneously.



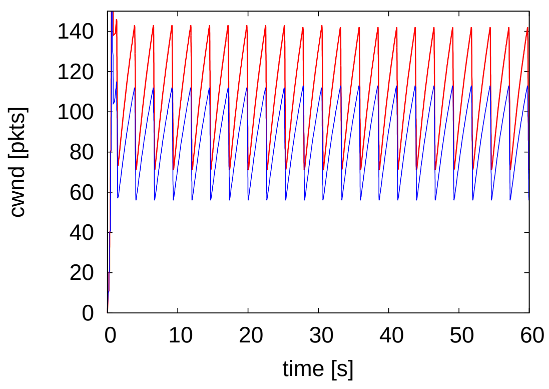
(b) Second flow starts at 4 seconds.



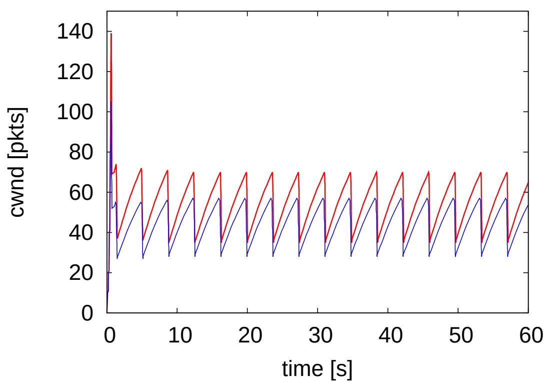
(c) Second flow starts at 5 seconds.



(d) Second flow starts at 6 seconds.



(e) Scenario with higher rate of 20 Mbit/s.



(f) Two flows with $Num_{RTT} = 30$.

Figure 4.2: Two competing SI_only flows.

Moreover, this trace shows one more effect: at about 30 seconds only one of the flows randomly gets a congestion notification and therefore the sharing changes. Taking a closer look at both figures it seems that the flows actually converge. This is only because of the computational inaccuracy that results from the fact that we only send out full packets and therefore can also only increase the sending rate by full packets. Still both flows never completely converge to equal sharing when running longer simulations.

Figure 4.2e illustrates the desired property of Scalable Increase to implement a fixed but configurable feedback rate (see design goals in Section 1.4). The congestion epoch time stays constant (as configured) even though the bottleneck rate is increased (from previously 10 Mbit/s) to 20 Mbit/s. Or, if the configuration is changed to $Num_{RTT} = 30$, the congestion epoch increases respectively, as shown in Figure 4.2f.

Note that if we use different Num_{RTT} values for both flows, this leads to a situation where the flow with the larger value reduces its sending rate more and more until the minimum sending rate of 1 packet per RTT is reached. This is because the flow with the larger Num_{RTT} value does not only calculate a smaller increase rate but also adapts its target to a lower value each time a congestion event occurs and the target was not reached yet. In the given setup, this happens every congestion epoch as the smallest configured values dominates the feedback frequency. This leads to a smaller and smaller target value and therefore smaller and smaller rate calculation in each congestion epoch until the minimum is reached.

This behavior can also be found in our final TCP SIAD approach but in combination with Trend and Fast Increase, the flow with the smaller share does not stay at the minimum rate but tries to grab additional capacity from time to time. Moreover, Additional Decrease also helps the flows with the larger share to release capacity quickly. Due to this interaction of the different mechanisms it is hard to calculate a resulting sharing ratio but we will show that the capacity sharing can still be influenced by the configuration parameter Num_{RTT} as desired.

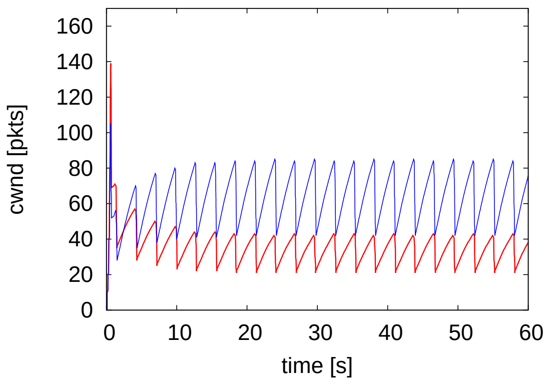
Fixed Increase

Additionally, we demonstrate how a certain sharing ratio is reached if different fixed increase rates are used by the competing flows. Therefore, we evaluate the Fixed_Increase variant. Figure 4.3a shows two flows, one with an increase rate of 1 packet per RTT (similar as TCP NewReno) and the other one with a rate of 2 packets per RTT. While the flow with the smaller increase rate reaches an average rate of 3.28 Mbit/s, the other flow gets about twice the capacity with 6.36 Mbit/s. Note in this case the link could not be fully utilized due to the buffer configuration of 0.5·BDP and the fixed decrease factor of 0.5. In Figure 4.3b we see two flows with the same increase rate as above of 1 and 2 packets per RTT but on a 20 Mbit/s link. The sharing ratio stays about the same with 6.51 Mbit/s and 12.78 Mbit/s, respectively. A different ratio is reached in Figure 4.3c where the two flows are configured with an increase rate of 1 and 3 packets per RTT, resulting in 2.64 Mbit/s and 7.02 Mbit/s. One of the flows gets about two times the share than the other as it can be calculated by resolving the TCP response function as given in Eq. 2.7 to the loss probability

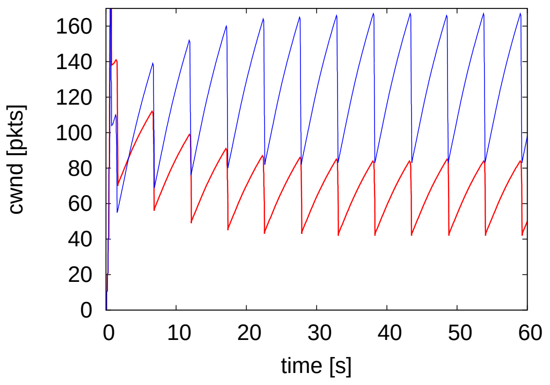
$$p = \frac{1}{RTT^2} \frac{\alpha(2-\beta)}{2\beta \cdot B^2} \tag{4.1}$$

Both competing flows, of course, have the same loss distance but the number of losses per congestion event depends on the increase rate per RTT, therefore

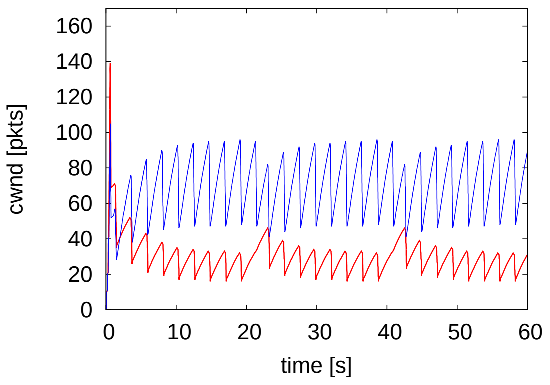
$$\alpha_1 \cdot \frac{\alpha_1(2-\beta)}{2\beta \cdot B_1^2} \overset{!}{=} \alpha_2 \cdot \frac{\alpha_2(2-\beta)}{2\beta \cdot B_2^2} \tag{4.2}$$



(a) Two flows with increase rate of 1 and 2 packets per RTT on 10 Mbit/s.



(b) Two flows with increase rate of 1 and 2 packets per RTT on 20 Mbit/s.



(c) Two flows with increase rate of 1 and 3 packets per RTT on 10 Mbit/s.

Figure 4.3: Two competing Fixed_Increase flows with 0.5·BDP of buffering.

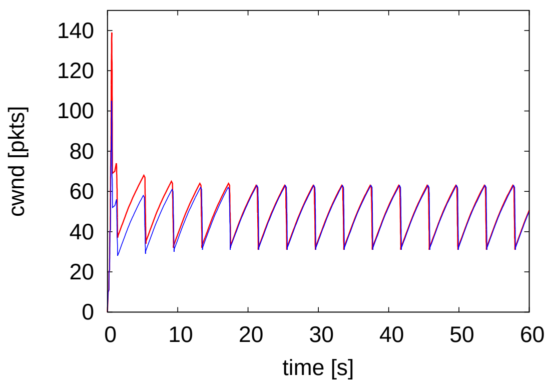
resulting in

$$\frac{\alpha_1}{\alpha_2} \overset{!}{=} \frac{B_1}{B_2}. \tag{4.3}$$

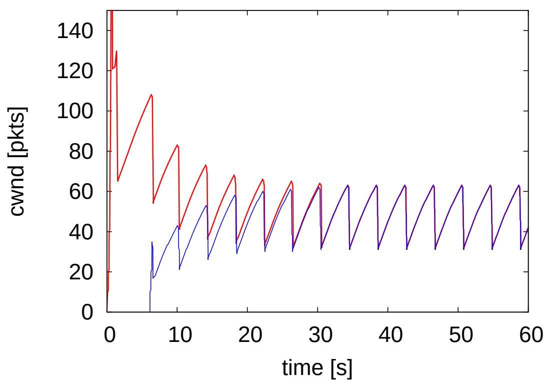
This demonstrates the capability to achieve certain capacity sharing ratio between competing flows that use different increase rates as a basis for our design goal to provide a respective configuration parameter to influence the capacity sharing.

Scalable Increase and Trend

To achieve convergence that is not reached in the SI_only case we evaluate the variant SI_trend that additionally implements the trend calculation to adapt the target value. Figure 4.4 shows two scenarios with different start times. It can be seen that both flows converge quickly in both cases, when they start simultaneously. Even when the second flow starts 6 seconds later which was the worst case in the sample scenario shown above for SI_only convergence is reached after about 15 s.

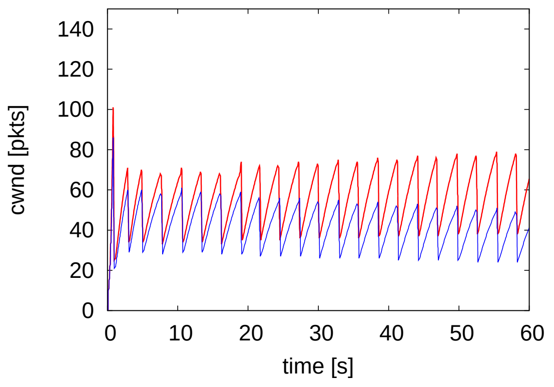


(a) Both flows start simultaneously.

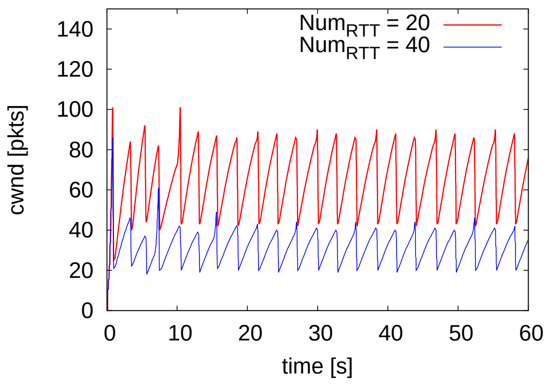


(b) Second flow starts at 6 seconds.

Figure 4.4: Two competing SI_trend flows.



(a) Both flows start simultaneously.

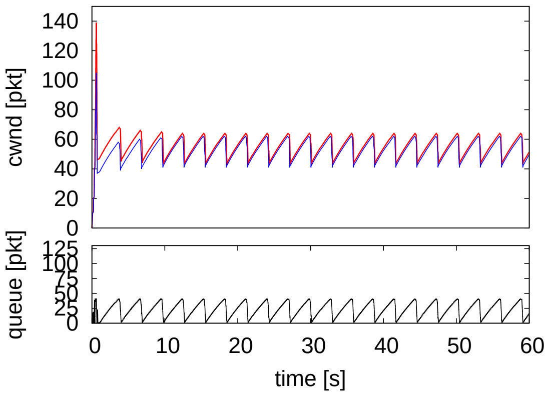


(b) Different Num_{RTT} values.

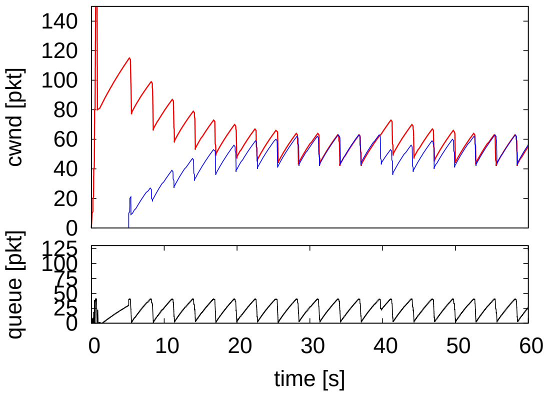
Figure 4.5: Two competing SI_trend_fastIncrease flows.

Unfortunately, if we take a look at the SI_trend_fastIncrease variant where trend and Fast Increase is implemented, convergence is worse. In Figure 4.5a first both flows converge quickly but then seem to de-converge again. Note that only 60 s of simulation time are shown here. Later on the flows will converge again. In fact over a longer simulation time of 580 s both flows share the link about equally with a rate of 4.96 Mbit/s and 4.6 Mbit/s. This means alternating one or the other flow has a slightly larger share for a period of several seconds. This effect occurs mainly because Fast Increase will introduce higher dynamics which disturbs the convergence state. Further, with the use of Fast Increase we also need to adapt the congestion window correctly based on the current increase rate before decreasing. This actually stabilizes the behavior but also leads to longer convergence periods when both flows do not share the link equally.

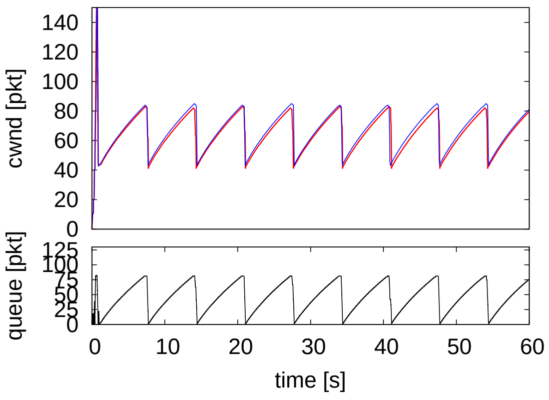
Figure 4.5b shows two flows with different Num_{RTT} values of 20 and 40. Fast Increase introduces more dynamics but not enough to really influence the achieved capacity sharing ratio. Therefore, with this implementation and the use of different Num_{RTT} values, one of the flows always operates at the minimum increase of 1 packet per RTT while the other (with the smaller Num_{RTT} value) operates at the configured rate. This means using a different Num_{RTT} than 40



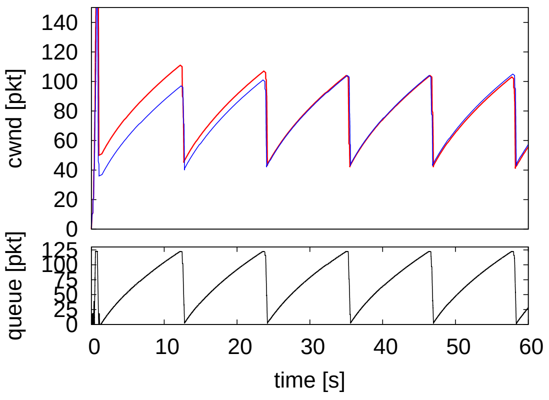
(a) Two flows with buffer of 0.5·BDP.



(b) Second flow starts at 5 seconds.



(c) Two flows with buffer of 1·BDP.



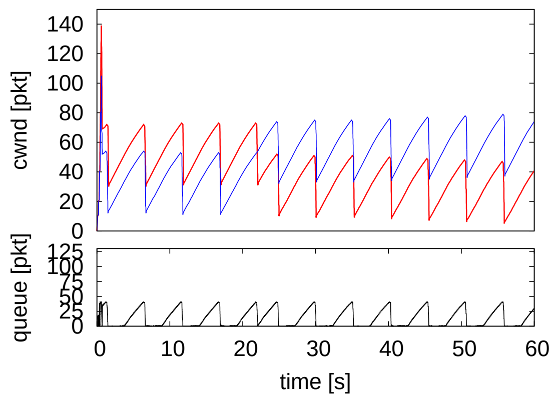
(d) Two flows with buffer of 1.5·BDP.

Figure 4.6: Two competing AD_only flows.

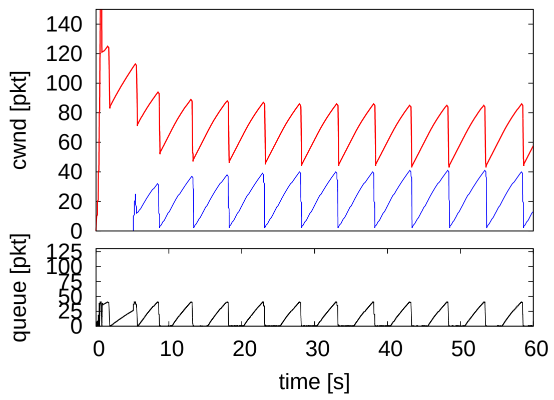
(that is still larger than 20 as used by the competing flow) will not change the sharing ratio. We will show later that a configurable capacity sharing can be reached in our final TCP SIAD approach (when also Additional Decrease is used).

4.2.2 Queue Development and Convergence with the Use of Adaptive Decrease

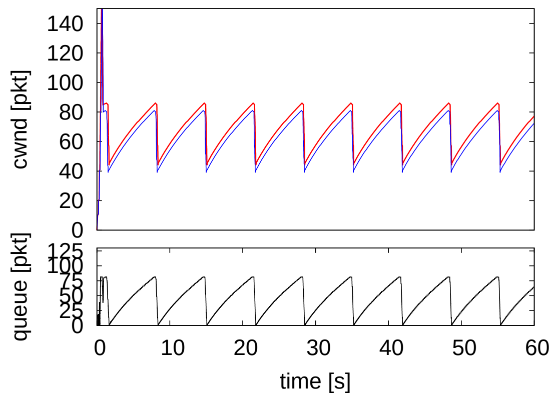
In this section we evaluate AD_only and Fixed_Decrease. While Adaptive Decrease only estimates the number of backlogged packets in the queue of the current flow, Fixed Decrease is a simple sample scheme that could be used if the total queue length would be known instead. However, both approaches can or cannot fulfill the design goals depending on flow synchronization; meaning that two of flows will reduce their congestion synchronously in the some RTT. We demonstrate that Adaptive Decrease can only empty the queue if the two competing flows are synchronized. In contrast, a reduction of number of packets that reflects the total queue size underutilizes the link in the case of synchronization. Subsequently, in the following section we further demonstrate that the chosen approach of using Adaptive Decrease in combination with Additional Decrease is, however, able to achieve our design goals by just emptying the queue no matter if the flows are synchronized or not.



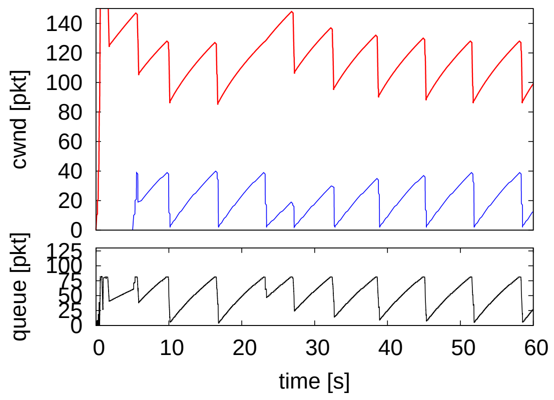
(a) Two flows with buffer of 0.5·BDP.



(b) Second flow starts at 5 seconds.



(c) Two flows with buffer of 1·BDP.



(d) Second flow starts at 5 seconds.

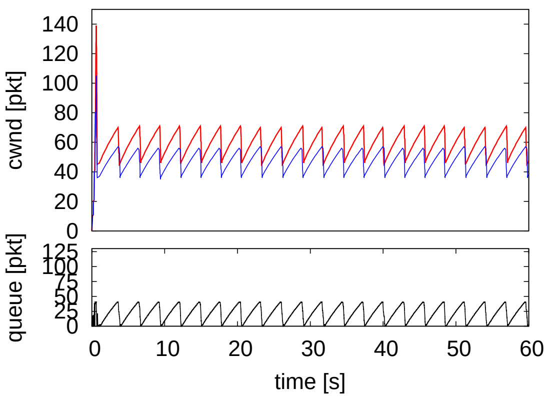
Figure 4.7: Two competing Fixed_Decrease flows.

Adaptive Decrease

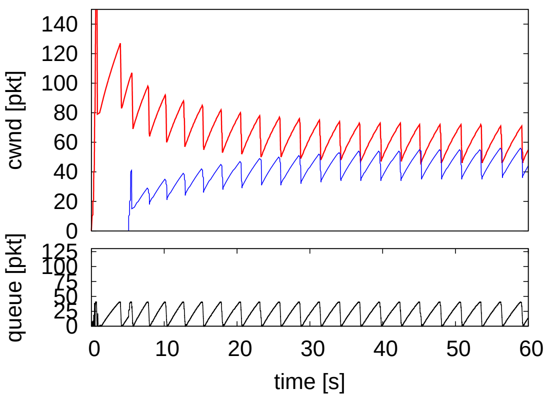
To demonstrate the desired adaptivity of Adaptive Decrease to the buffer size we evaluated the AD_only variant in three scenarios with buffer sizes of 0.5, 1.0, and 1.5 times the base BDP. Figure 4.6 shows additionally to the congestion window over time also the queue length over time. It can be seen that in all cases the buffer is emptied at every decrease and therefore a standing queue can be avoided. Further, the queue starts immediately to grow again after a decrease which keeps the link utilization high. Figure 4.6b also demonstrates that Adaptive Decrease still maintains the same convergence properties than traditional AIMD as it is still a multiplicative decrease scheme.

Fixed Decrease

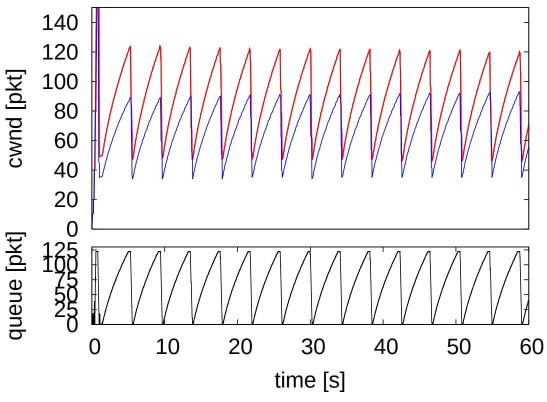
As an alternative we demonstrate a variant with the fixed subtractive decrease, named Fixed_Decrease, which does not use a multiplicative decrease scheme anymore. Figure 4.7 shows simulation results where a fixed decrease of 42 packets on congestion notification is configured. In Figures 4.7a and 4.7b the queue size is configured to 0.5·BDP which translates to 41 full-sized packets. As we have two (synchronized) flows in this scenario and therefore in total decrease by 84 packets, this configuration underutilizes the link. Only one case can be observed at around



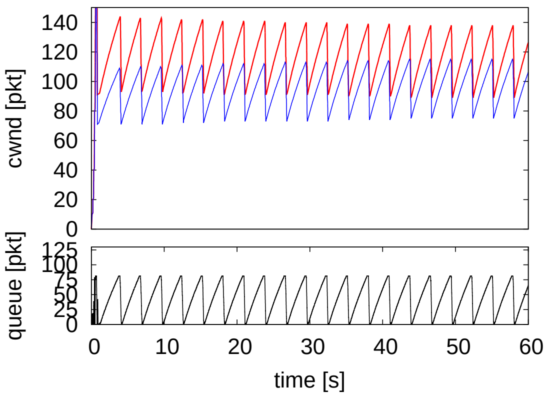
(a) Two flows with buffer of 0.5·BDP.



(b) Second flow starts at 5 seconds.



(c) Two flows with buffer of 1.5·BDP.



(d) Two flows at 20 Mbit/s.

Figure 4.8: Two competing SIAD_only flows.

22 seconds where the decrease just empties the buffer as only one of the flows got a congestion notification by chance. Further, it can be seen, especially in Figure 4.7b where the second flow starts later, that the subtractive decrease behavior does not provide convergence. Even worse, the flow that starts later when the link is already utilized, still always decreases by the same fixed value as the competing flow. Therefore, it only reaches a maximum congestion window of 42 packets within one congestion epoch and then decreases again to *MIN_CWND*. Note in Figure 4.7c both flows by chance achieve a nearly equal sharing (but still never converge fully). In this scenario, the buffer is configured to a size of 1.0·BDP and respectively 82 full packets while the flows still decrease by 42 packets (only). As both flows are synchronized the buffer is just emptied with every decrease. However, if only one flow decreases or one of the flows is not able to decrease by the total of 42 packets, this causes a standing queue and therefore unnecessarily increases the delay as it can be observed clearly in Figure 4.7d. Moreover, in Figure 4.7d the same effect as before can be still observed when one of the flows starts later.

4.2.3 Fixed Feedback Rate, Queue Development, and Convergence with the Use of Scalable Increase and Adaptive Decrease

In this section, we show that the demonstrated abilities of Scalable Increase and Adaptive Decrease still hold if both algorithms are used in combination based on the SIAD_only variant.

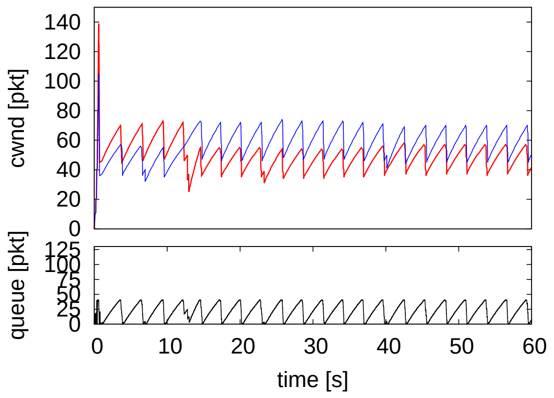


Figure 4.9: Two competing SAID_addDecrease flows.

Further, we demonstrate that the use Additional Decrease together with Scalable Increase Adaptive Decrease in the SIAD_addDecrease variant can also the empty the queue in those cases where the decrease is not synchronized. Finally, we demonstrate that the Trend calculation still re-introduces convergence to equal capacity sharing of two competing flows if used with Scalable Increase Adaptive Decrease in the SIAD_trend variant.

Scalable Increase Adaptive Decrease

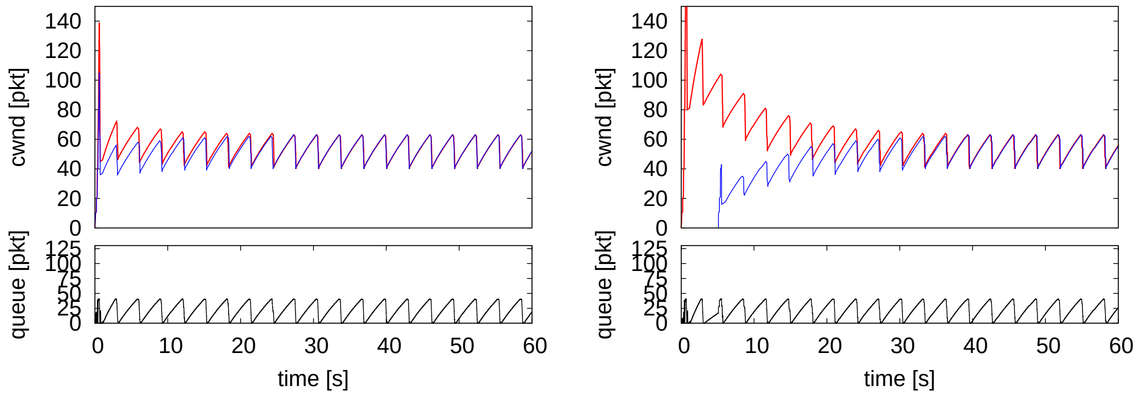
Figure 4.8 reveals the same problem to convergence for SIAD_only as for SI_only. Further, SIAD_only needs a longer time to converge to a stable state in case of one of the flows starting later as shown in Figure 4.8b. But Figure 4.8c and 4.8d show the adaptivity to different queue sizes and, respectively, the scalability due to the fixed feedback rate independent of the link bandwidth.

Scalable Increase Adaptive Decrease and Additional Decrease

By evaluating SIAD_addDecrease we show the ability of the Additional Decrease algorithm to always empty the queue even when not synchronized, e.g., as it can bee seen around 12 seconds in Figure 4.9. In this moment by chance only one of the flows gets a congestion notification and therefore additional Decrease is subsequently performed by the same flow until the queue is sensed empty.

Scalable Increase Adaptive Decrease and Trend

As previously seen for SI_trend and now illustrated for SIAD_trend in Figure 4.10, the trend calculation provides convergence to equal sharing when two flows with the same NUM_{RTT} configuration compete independent of the starting times of the flows. The convergence speed is about similar as previously demonstrated for SI_trend as the addition of Adaptive Decrease does not have any influence on convergence of two synchronized, competing flows.



(a) Both flows start simultaneously. (b) Second flow starts at 5 seconds.

Figure 4.10: Two competing SIAD_trend flows.

4.2.4 Conclusion

In this section we have shown that the general principle of Scalable Increase can provide a fixed feedback rate independent of the link bandwidth as desired for scalability. Further, Adaptive Decrease is able to empty the queue on each congestion event if all flows are synchronized. We demonstrated the ability of Additional Decrease to also cope with situations where flows are not synchronized. As a drawback, any implementation of Scalable Increase does not provide convergence to an equally sharing ratio anymore even though that is a known property of AIMD. Therefore, we have shown that the introduction of a Trend calculation can re-introduce convergence to equally sharing if Scalable Increase is used. Not shown in this section but clearly displayed later in more dynamic scenarios is that Fast Increase is needed to provide quick allocation for newly available bandwidth. However, the introduction of Fast Increase comes with higher dynamics and therefore is disadvantageous for convergence as well. Still, over a longer time scale Trend calculation can still provide convergence to equal sharing and therefore is a valuable and important component of the final TCP SIAD algorithm.

Having shown that each of the single algorithms provides a component that is needed to fulfill the design goals as stated in Section 1.4 and having demonstrated the implications on convergence and capacity sharing with the use of Scalable Increase and well of the introduction of Trend, we subsequently evaluate the complete TCP SIAD algorithm, as described in the previous chapter, in the following sections.

4.3 Adaptivity and Scalability based on Single-Flow Behavior

In this section we concentrate on a scenario with only one TCP connection from one data-unlimited sender to one receiver. We use this simple scenario to evaluate the required adaptivity to different buffer sizes and scalability with link speed of the proposed TCP SIAD scheme and in comparison to other high-speed congestion control schemes. Again, all simulation runs had an OWD of 50 ms, thus an RTT of 100 ms. At first, we show the congestion window development of TCP SIAD and six other current and in Linux implemented (high-speed) schemes, namely TCP NewReno, TCP Cubic, Scalable TCP, High Speed TCP, H-TCP and TCP Illinois. Further, we will compare the mean link utilization and average queue length to show that a standing queue can be avoided while the link is still always highly utilized with TCP SIAD. This addresses the adaptivity requirement as stated in Section 1.2. We, moreover, show the loss distance and total loss rate to proof TCP SIAD's scalability to the link speed.

4.3.1 Statistical Evaluation in Steady State

Figure 4.11a shows the congestion window of one TCP NewReno flow and the queue length over time for different link speeds (10 Mbit/s and 20 Mbit/s) and buffer sizes (1.5·, 1.0·, and 0.5·BDP). It can be seen that the feedback rate strongly depends on the average BDP and therefore also the maximum queue size. Moreover, the queue and so the link run empty if the configured buffer size is smaller than the base BDP.

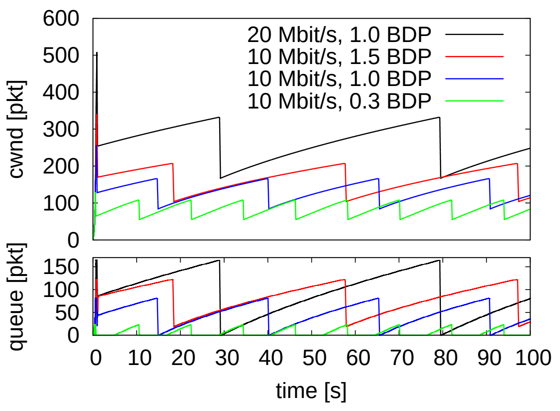
TCP Illinois has a very similar behavior as TCP NewReno. Only the increase is faster at the beginning of each congestion epoch as it can be seen in Figure 4.11b. This helps to better utilize the link in case of small bottleneck buffer configurations but also causes a higher average queue length and therefore high (and unnecessary) end-to-end delays. However, as it used the same decrease scheme, TCP Illinois does not improve the scalability.

As shown in Figure 4.11c TCP Cubic can utilize bottlenecks with smaller buffers (down to half the BDP) as it only reduces the sending rate by 0.3. In return it causes a larger standing queue already with smaller buffer sizes than TCP NewReno. Further, a typically bi-frequent behavior of TCP Cubic's congestion window can be seen. As designed for high-speed networks TCP Cubic's feedback rate is higher than TCP NewReno's. However, the congestion event distance still gets larger when the link capacity increases and therefore TCP Cubic scales better than TCP NewReno but does not solve the scalability problem fully.

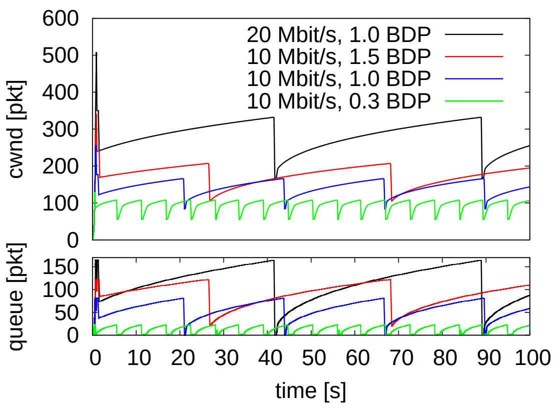
For High Speed TCP as in Figure 4.11d the increase rate gets higher the larger the link bandwidth is. Therefore, for the selected scenarios with 10 Mbit/s and 20 Mbit/s the feedback rate seems to be similar but in fact still depends on the link speed (as we show more clearly next with the statistical assessment of the congestion event distance). Moreover, as already explained in Section 2.2.2, High Speed TCP selects the decrease factor such that it is fair to TCP NewReno cross traffic on link speed links and therefore still causes either link underutilization or a standing queue.

In contrast, Scalable TCP provides the same feedback rate in all scenarios and therefore fully scales with all link speeds as expected for a MIMD scheme. Unfortunately, it most often causes a large standing queue as shown in Figure 4.11e. Further, as we assess next in more detail, the feedback rate is high and therefore also the loss rate is very high.

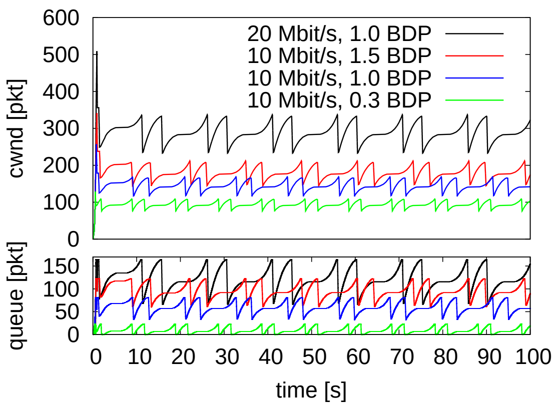
Figures 4.11f displays the congestion window and queue length of H-TCP. Similar to TCP SIAD in Figure 4.12, H-TCP adapts its decrease factor to the queue size. As H-TCP only implements a decrease between 0.3 and 0.5, it still causes a standing queue when the buffer is larger than the base BDP. Comparing the increase rate of H-TCP and TCP SIAD based on the shown congestion window plots, we expect a higher loss rate for H-TCP which we will as we further investigate below.



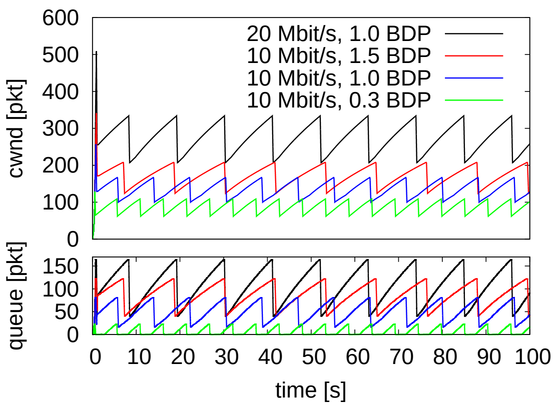
(a) TCP NewReno.



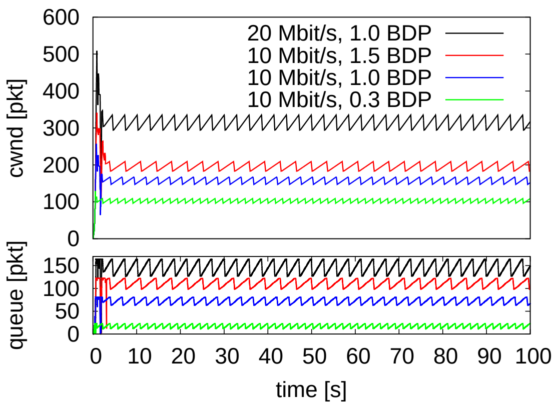
(b) TCP Illinois.



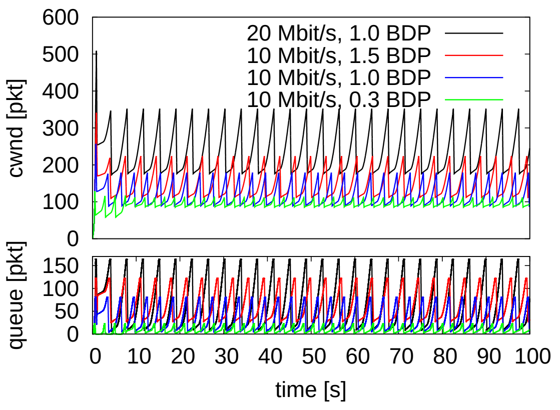
(c) TCP Cubic.



(d) High Speed TCP.



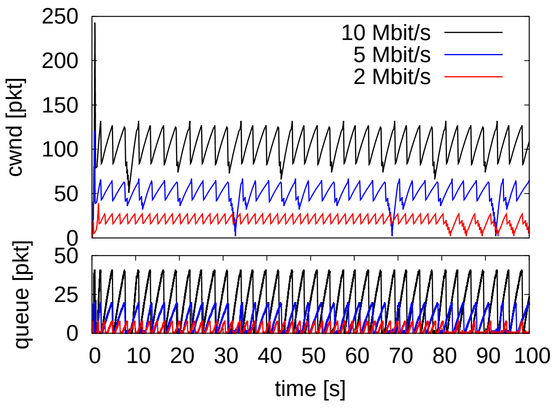
(e) Scalable TCP.



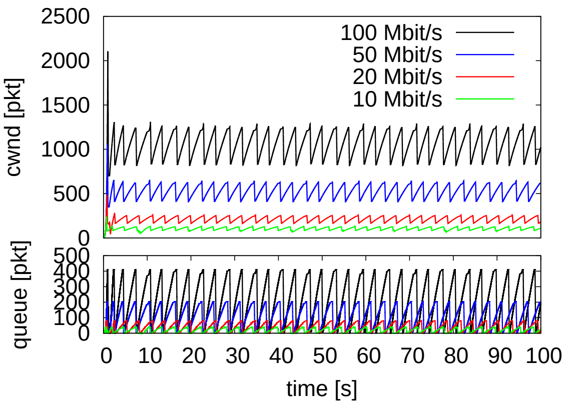
(f) H-TCP.

Figure 4.11: Congestion window and queue length over time for a single flow.

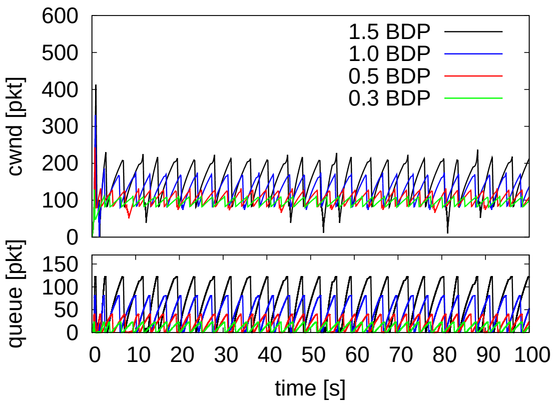
In TCP SIAD we removed this limitation because Scalable Increase partly compensates a too large decrease as it can be seen, e.g., in Figure 4.12a where Additional Decreases are performed from time to time (negative spikes) which, however, are recovered quickly as the increase rate is respectively larger afterwards. Figures 4.12a and 4.12b illustrate simulation results for different link speeds but with the same buffer size of 0.5·BDP. For all scenarios, TCP SIAD operates with the configured (or a higher) feedback rate and therefore scales. Note, if the buffer size (in packets) is smaller than the configured Num_{RTT} value (in RTTs), TCP SIAD respectively operates



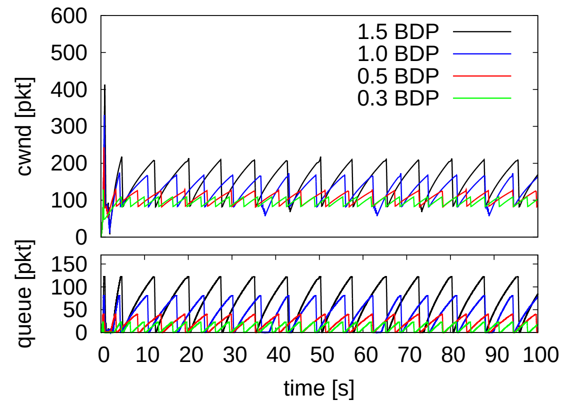
(a) TCP SAID with rates 2 to 10 M/bits.



(b) TCP SAID with rates 10 to 100 M/bits.



(c) TCP SAID with $Num_{RTT} = 20$.



(d) TCP SAID with $Num_{RTT} = 40$.

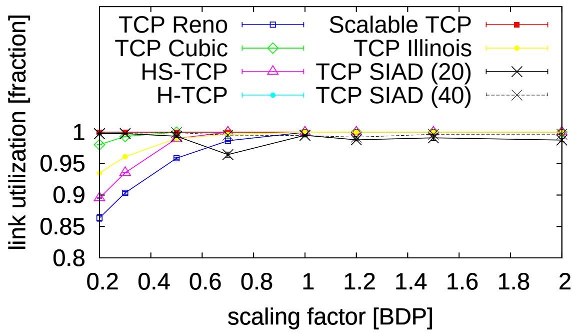
Figure 4.12: Congestion window and queue length over time for a single TCP SIAD flow.

at the minimum increase rate of 1 packet per RTT instead of the desired rate and consequently the feedback rate is also higher than configured. Figures 4.12c and 4.12d show simulation runs with various buffer size configurations on a 10 Mbit/s link for Num_{RTT} values of 20 and 40. It can be seen that the feedback rate is controlled as desired based on the Num_{RTT} configuration. Further, in all cases there is never a standing queue as Adaptive Decrease correctly calculates the decrease factor for all buffer configurations.

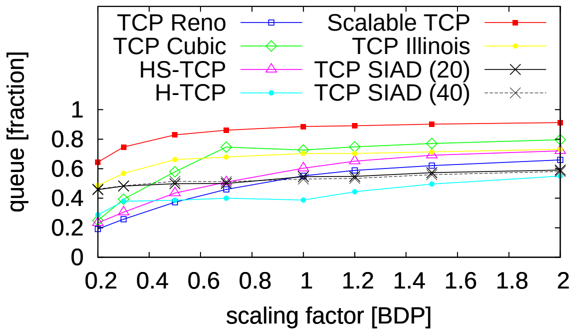
After assessing the different characteristics of the regarded schemes, we will now compare statistical properties for all schemes in a larger set of scenarios with link speeds from 1 to 100 Mbit/s and buffer size configurations from 0.1 to 2.0·BDP. All figures below show two configurations for TCP SIAD with Num_{RTT} values of 20 and 40.

In Figures 4.13a, 4.13b, and 4.13c, the average link utilization, the average queue fill level as well as the minimum queue fill level are shown for simulation runs with a bandwidth of 10 Mbit/s but different buffer sizes.

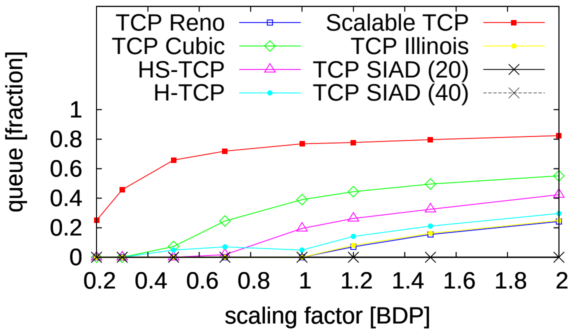
TCP SIAD is designed to always fill the queue (as every loss-based scheme). We made the decision to design a loss-based scheme to be able to compete with loss-based schemes which are predominant in the Internet. However, TCP SIAD is also designed to always fully empty the queue on congestion. This goal was reached in all simulations as a minimum queue fill level of



(a) Average link utilization.



(b) Average queue fill level.

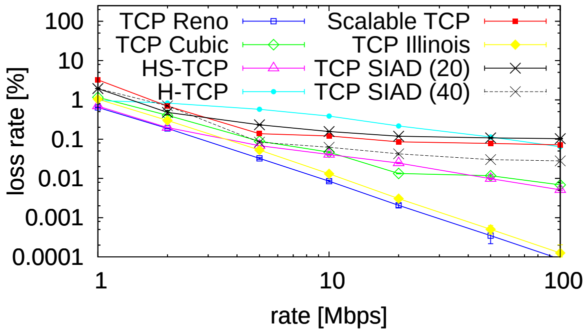


(c) Minimum queue fill level.

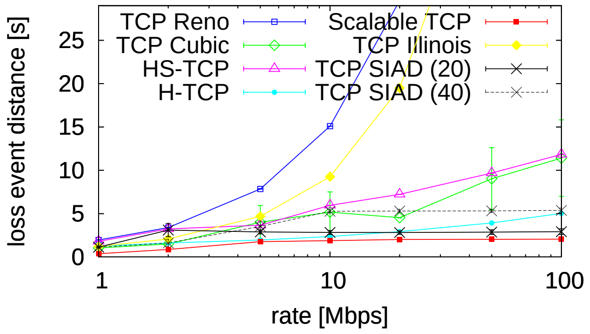
Figure 4.13: Single flow at 10 Mbit/s.

zero indicates in Figure 4.13c. Further, as TCP SIAD implements linear increase it targets an average queue fill of 0.5 of the maximum configured buffer size. This can be observed in simulations in Figure 4.13b. Unfortunately, varying between queue length of zero and the maximum configured queue length also induces the maximum delay variation, as the queue fill level can directly be translated into end-to-end queuing delay. As small queuing delays and low variations are needed for latency sensitive applications, small queues or at least low ECN marking or dropping thresholds in case of AQM are desired with the use of TCP SIAD. While small queues or low thresholds lead to link underutilization with most traditional schemes, TCP SIAD allows network operators to configure small queues and still reach high network utilization as indicated in Figure 4.13a. It can be seen that TCP SIAD always reached a high utilization. In one of the simulation scenarios (0.7 BDP), TCP SIAD performs various Additional Decreases, similar as shown previously, and therefore only reaches a slightly lower, but still high utilization. Note that the use of AQM provides free queue space for small traffic burst without inducing burst loss. Further, the use of ECN avoids congestion signaling losses. TCP SIAD would benefit from the use of ECN as a configured, high feedback rate (to reach high responsiveness) can also lead to an increased loss rate compared to other high-speed schemes as shown further below. However, the smaller the queue, the smaller is also the per-RTT increase step of TCP SIAD which again lowers the overshoot and therefore the loss rate.

As expected only Scalable TCP, H-TCP and TCP SIAD can utilize the link fully with very small buffer sizes as it can be seen in Figure 4.13b. Of these H-TCP maintains the lowest average queue fill level, as shown in Figure 4.13b, but it also introduces a standing queue as the minimum queue length that larger than zero indicates in Figure 4.13c. Only TCP SIAD is able



(a) Average loss rate.



(b) Average loss event distance.

Figure 4.14: Single flow with queue size of 0.5·BDP

to always empty the queue completely in all scenarios. Even though H-TCP does not always empty the queue fully, it increases the sending rate more than linearly and therefore stays for a longer time at a lower queue fill level. Therefore, the average queue fill is most often lower than for TCP SIAD. However, the drawback of this increase function is the large overshoot when the queue is finally filled resulting in a larger loss rate as can be seen in Figure 4.14a. Note, we also considered choosing a different increase function for TCP SIAD but with the expectation of smaller queues in future networks the gain in average latency does not justify the additional complexity as already discussed in Section 3.1 in the previous chapter. Further, while TCP SIAD always maintains an average queue fill level of about 50%, the average queue fill of H-TCP increases with the queue size as it induces a larger the standing queue. While H-TCP is designed to only address scenarios with small buffer sizes, TCP SIAD can even avoid a standing queue in case of large buffer configurations.

For all other schemes the average as well as the minimum queue fill level also increase with the configured queue size as they also build up a standing queue. This introduces permanent additional and unnecessary delay. Of course whenever there is a permanent standing queue, the link is fully utilized. This is the case for all other schemes other than TCP SIAD when the queue is larger than the BDP as the minimum queue fill level indicates in Figure 4.13c. TCP SIAD is designed to rather slightly underutilize the link but drain the queue with every decrease. Therefore, e.g., in the scenario with a maximum queue size of 0.7·BDP, TCP SIAD only achieves a slightly lower utilization (but still 96.4%). In this scenario TCP SIAD performs frequently Additional Decreases as the minimum delay is not measured correctly.

Appendix C shows the same diagrams for a link speed of 20 Mbit/s. It can be seen that similar results can be achieved also with a higher link speed.

While one goal of TCP SIAD is high utilization even with small queues, another goal is to maintain a given feedback rate and therefore scale with any network bandwidth. In Figures 4.14a and 4.14b we see the average loss rate and the average time between two loss events which corresponds to the feedback rate. In this setup the maximum queue size is configured to 0.5·BDP while the bandwidth is varied.

Even though all high-speed proposals scale better than TCP Reno and TCP Illinois, only Scalable TCP and TCP SIAD maintain a feedback rate independent of the bandwidth. Scalable TCP

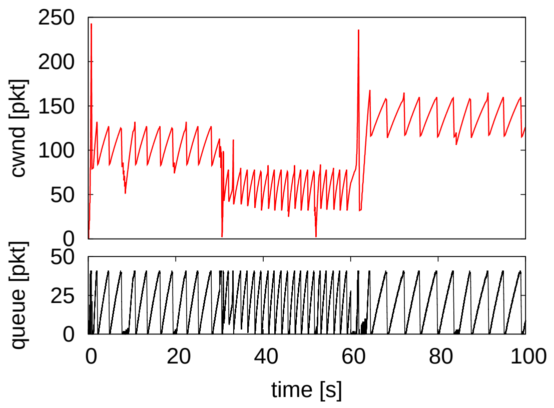


Figure 4.15: Single flow with changing RTTs.

provides a fixed and quite high loss rate at the cost of high queuing delay. In contrast TCP SIAD can be configured to maintain a given congestion event distance which also influences the loss rate as it is shown for the two example configurations where Num_{RTT} equals 20 or 40. For small BDPs the targeted feedback rate in Figure 4.14b is exceeded, as the maximum congestion window is very small (less than 40 or less than 20 packets) and TCP SIAD implements a minimum increase rate of 1 packet per RTT.

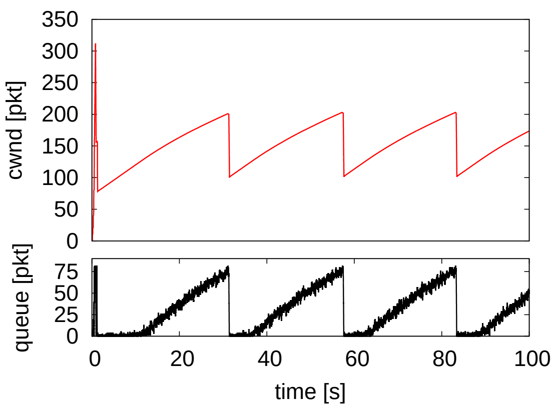
Note that the error bar shown is the standard deviation of the loss event distance. The standard deviation is especially high for TCP Cubic due to the bi-frequent behavior of the congestion window as shown earlier.

Appendix C show the same diagrams for a buffer size of 1.0·BDP which allows a fairer comparison with TCP Reno but provides the same conclusion.

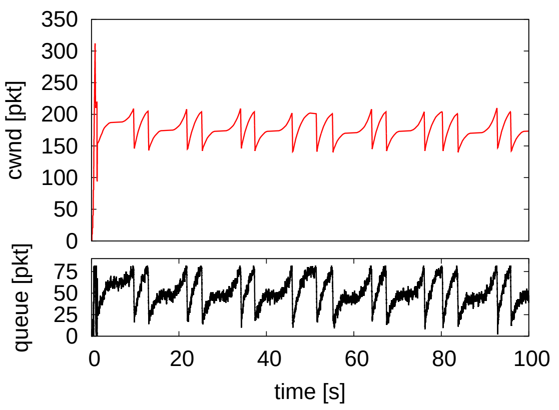
4.3.2 Assessment of the Vulnerability to Delay Estimation Errors

In this section we evaluate the ability of TCP SIAD to adapt to permanent delay changes, e.g., due to route changes, or to high frequency delay variations. Further, we assess the influence of CBR cross traffic on the base delay estimation. Note, most hybrid schemes use the total minimum RTT seen during a connection and do not update in any case. As TCP SIAD aims to empty the queue in every congestion epoch, it fortunately is also able to measure (and update) the base RTT in every epoch. Of course, there are situations where the base RTT cannot be measured, e.g., if the time stamp resolution is too small or due too strong variation of the delay samples. In these cases we have to rely on previous measurements.

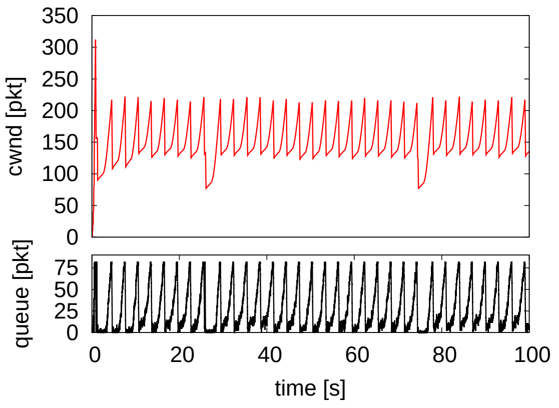
Figure 4.15 shows one single TCP SIAD flow on a 10 Mbit/s link with a bottleneck buffer size of 0.5 BDP. At the beginning of the simulation the RTT is 100 ms, after 30 s of simulation time the RTT is changed to 40 ms, and at 60 s of simulation time the RTT is increased to 140 ms. It can be seen that it takes a few overshoots before TCP SIAD correctly adapts to a smaller RTT. This is because even though the smaller base RTT can be measured immediately after a decrease, also the Linear Increment threshold *incthresh* and the trend calculation have to adapt to a new maximum congestion window which is also smaller now. The adaptation to a larger value is completed as soon as one decrease with the old value is performed. Of course, the



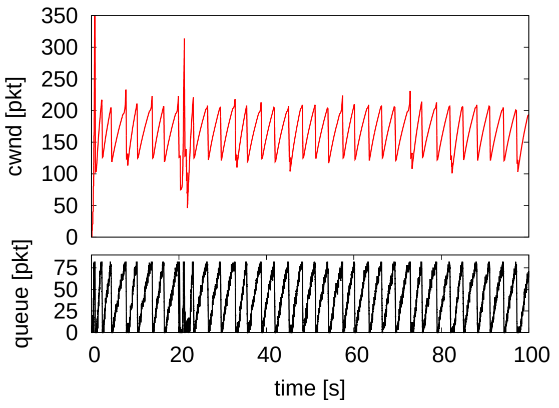
(a) TCP NewReno.



(b) TCP Cubic.



(c) H-TCP.



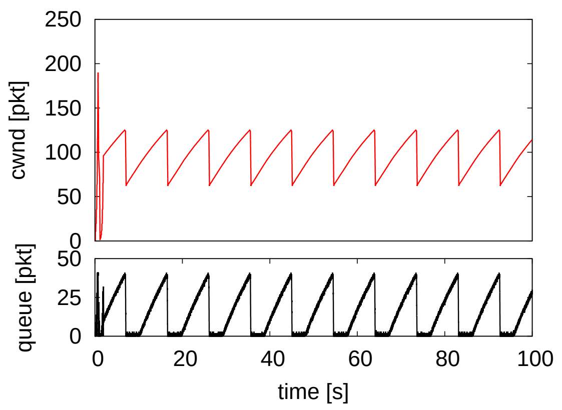
(d) TCP SIAD with $Num_{RTT} = 20$.

Figure 4.16: Congestion window and queue length over time with per-packet rates between 10 Mbit/s and 30 Mbit/s.

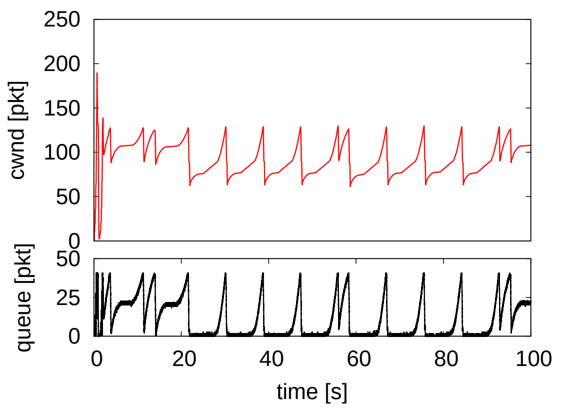
old decrease value provokes a too large decrease. But due to this too large decrease the link gets underutilized for a short time and the minimum delay can be updated. Subsequently, Fast Increase provides quick re-allocation of the link resources.

In the following, we will evaluate two extreme cases where we vary either the link serving rate or the link transmission base delay on a per packet basis. In wireless networks, the rate and delay can change quickly but usually there are still a couple of packets served with the same rate. Note, while in wireless network today the lower layer already provides in-order delivery (at the cost of slightly higher delays), in our case of varying per-packet delay, one packet can overtake another. Therefore, these scenarios can be rarely found in real (wireless) network but still provide extreme cases for evaluation.

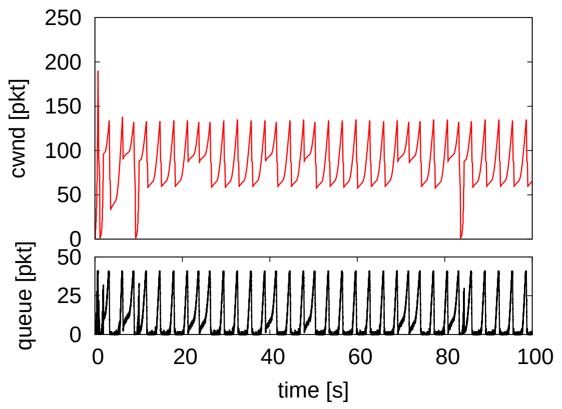
Figure 4.16 shows the congestion window of one TCP NewReno, one TCP Cubic, on H-TCP, or one TCP SIAD flow as well as the queue size over time. The link in this simulations has a per-packet serving rate between 10 Mbit/s and 30 Mbit/s. This leads to delay variations but in-order delivery. The base RTT is 100 ms. The buffer size is configured to 1.0·BDP based on the minimum rate of 10 Mbit/s.



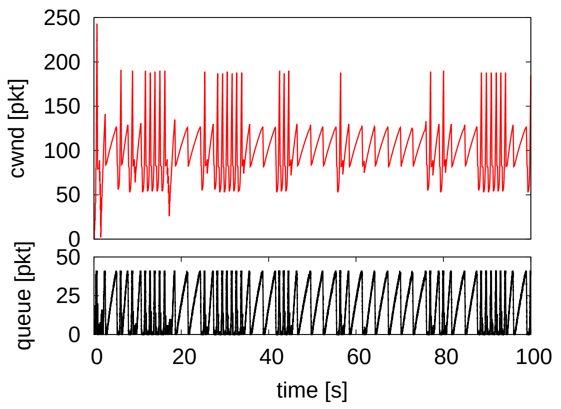
(a) TCP NewReno.



(b) TCP Cubic.



(c) H-TCP.



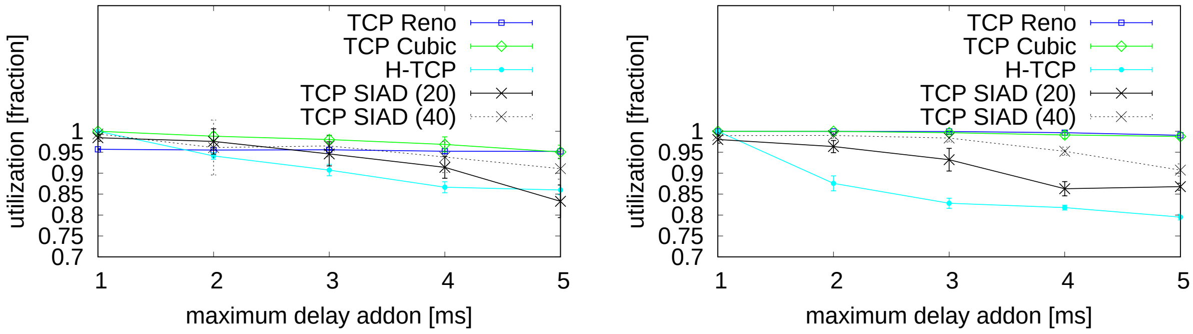
(d) TCP SAID with $Num_{RTT} = 20$.

Figure 4.17: Congestion window and queue length over time with per-packet RTT variations between 100 ms and 103 ms.

It can be seen that the congestion window development of TCP NewReno and TCP Cubic does not seem to get influenced by these small delay variations. However, as the buffer size is configured based on the smallest serving rate, TCP NewReno cannot always keep the link full but still reaches a link utilization of 97.8%. Both H-TCP and TCP SIAD are visibly influenced by the delay variations as from time to time a too large decrease factor is calculated. Still both schemes reach a high link utilization of 97.9% and 98.3%, respectively. TCP SIAD reaches a slightly higher utilization due to the ability of Scalable Increase to calculate a higher increase rate.

Figure 4.17 shows again the congestion window and queue length over time for TCP Reno, TCP Cubic, H-TCP and TCP SIAD. The buffer size is 0.5·BDP based on a base RTT of 100 ms. This time the link serving rate is fixed to 10 Mbit/s but the per-packet delay is varied between 100 ms and 103 ms as one example for potential realistic delay variations. Note that per-packet delay variations lead to re-ordering that is wrongly interpreted by TCP as loss and therefore can lead to unnecessary sending rate reductions. Of course, the higher the delay variation, the more re-ordering happens.

With this buffer configuration TCP NewReno can anyway not fully utilize the link but is otherwise not further influenced by the delay variation. In contrast TCP Cubic, that should be able to



(a) With 0.5·BDP of buffering.

(b) With 1·BDP of buffering.

Figure 4.18: Link utilization with additional per-packet delay between 1 ms and up to 5 ms and a base RTT of 100 ms.

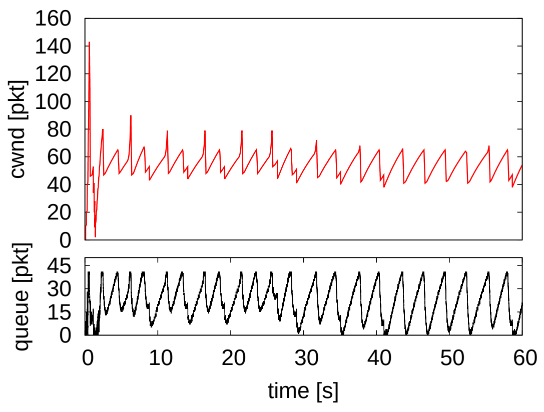
fully utilize a link with 0.5·BDP of buffering, experiences additional congestion events due to re-ordering. This is because TCP Cubic is increasing its sending rate faster with a higher rates than TCP NewReno. In this scenario TCP Cubic reaches only a utilization of 98.0%. H-TCP and TCP SIAD are even more influenced by the delay variation as both are hybrid schemes. H-TCP can only reach a utilization of 90.7% and TCP SIAD of 83.2% in this scenario. The strong oscillations that can be seen for TCP SIAD are due to Scalable Increase that is directly followed by a Additional Decrease phase. This happens more likely when TCP SIAD's estimations have been wrong, in this case due to link delay variations.

Figure 4.18 shows the link utilization for different amplitudes of delay variations. While TCP NewReno has a nearly constant high utilization in all scenarios, the utilization for TCP Cubic decreases slightly with increasing delay variations. H-TCP and TCP SIAD, both, are strongly affected by these variations. But note this is an extreme case that can rarely be found in real networks. For TCP SIAD, the utilization decreases the more, the smaller the Num_{RTT} value is as a smaller value leads to a larger per-RTT increase rate.

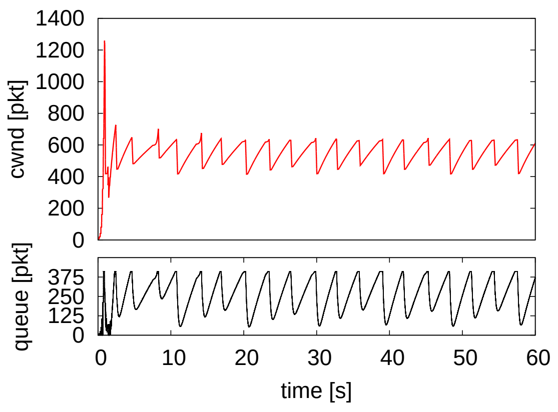
In the following we demonstrate a case where TCP SIAD is not able to empty the buffer on decrease because of non-adaptive cross traffic.

Figure 4.19a shows the congestion window and queue length of one TCP SIAD flow on a 10 Mbit/s link with 100 ms RTT and a buffer size of 0.5·BDP. Additionally to the TCP SIAD flow, there is CBR traffic sending with a rate of 5 Mbit/s using equal sized packets of 1514 Bytes. As explained earlier, Adaptive Decreases calculates its decrease factor based on its own share of the bandwidth assuming flow synchronization. If there is CBR traffic on the same link, this traffic of course does not decrease at all. Consequently TCP SIAD performs Additional Decreases from time to time. Therefore, as the queue length over time shows, TCP SIAD is often not able to fully empty the buffer.

In Figure 4.19b a 100 Mbit/s link with 50 Mbit/s CBR cross traffic is shown. In this case the time stamp resolution of TSOpt is too small (1 ms in Linux) to detect increases in delay right after a decrease. Therefore, the minimum delay will be wrongly updated in every congestion epoch to a higher value. It can be seen that TCP SIAD detects three increases of the minimum delay in a row and resets the minimum delay to the first value every fourth congestion events.

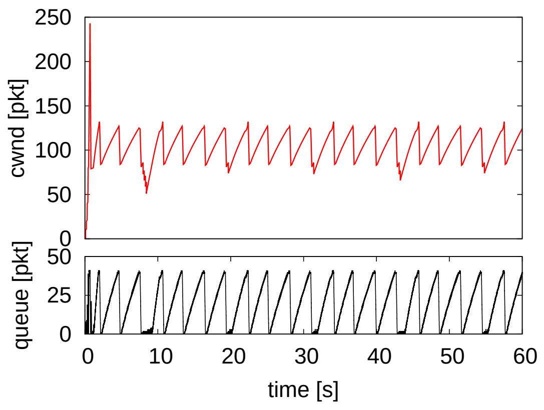


(a) On 10 Mbit/s link with 5 Mbit/s CBR traffic.

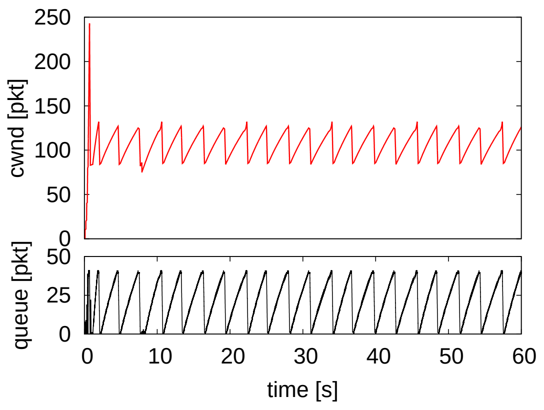


(b) On 100 Mbit/s link with 50 Mbit/s CBR traffic.

Figure 4.19: One TCP SIAD flow with CBR cross traffic.



(a) On 10 Mbit/s link with TSOpt.



(b) On 10 Mbit/s link without TSOpt.

Figure 4.20: One TCP SIAD flow w/ and w/o TCP Timestamp Option (TSOpt).

Even though the minimum delay does not further increase by using this approach, it is never measured correctly and therefore this situation causes a standing queue.

Finally, we demonstrate how TCP SIAD works if TSOpt is not supported by the receiver side. In this case the Linux kernel samples RTT measurements and calculates a smoothed average as described in Section 3.3.2. Figures 4.20a and 4.20b show one TCP SIAD on a 10 Mbit/s link with an RTT of 100 ms and buffer size of 0.5·BDP with and without TSOpt enabled. It can be seen that the flow without TSOpt support performs less Additional Decreases because of the smoothed measurement.

4.3.3 Conclusion

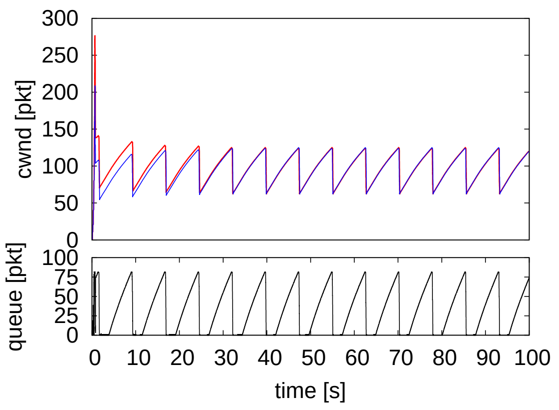
We demonstrated in a steady state single flow scenario the ability of TCP SIAD to utilize the bottleneck and to avoid a standing queue with arbitrary buffer sizes. This fulfills the requirement on adaptivity as stated in Section 1.2. Further, we have shown that the configured feedback rate can be reached under various network conditions and therefore TCP SIAD provides scalability as required for high-speed networks. None of the schemes that we evaluated in comparison, including H-TCP, were able to achieve high link utilization and avoid a standing queue for all scenarios. In case of H-TCP, this is due to the limitation of the decrease factor range which we do not have in TCP SIAD. Further, only Scalable TCP and TCP SIAD provide the same congestion feedback rate for all scenarios independent of the link bandwidth. While Scalable TCP induces a fixed feedback rate most often leading to high loss rates, the feedback rate of TCP SIAD is configurable.

Additionally, still focusing on evaluations with only a single congestion controlled flow, we assessed the vulnerability of TCP SIAD on delay estimation errors due to changes in the base delay, delay variation, or the influence of CBR cross traffic. We demonstrated that TCP SIAD can quickly adapt to a larger or smaller base delay while other delay-based or hybrid schemes usually do not adapt at all but use the absolute minimum seen during the connection as the base delay. While both hybrid schemes in evaluation, TCP SIAD and H-TCP, are more sensible to delay variations than others, they can still achieve reasonable link utilization in extreme scenarios. Of course, the higher the aggressiveness of TCP SIAD is configured the better is the link utilization. In the presence of CBR cross traffic, Adaptive Decrease is not able anymore to fully empty the queue. However, if the base delay can be estimated correctly, TCP SIAD performs Additional Decreases. Unfortunately, if the link bandwidth is very high, the current time resolution of TSOpt is not sufficient to estimate the delay correctly. For this case, TCP SIAD implements a mechanism to detect monotonous increases in base delay and subsequently resets the minimum delay to a previous value (as previously described in Section 3.2.5). Originally TSOpt was not designed to be used for congestion control but only for RTT estimation which usually is in the range of 10 to 100 ms. To address new use cases, there is standardization activity in the IETF [129]. If the TSOpt is not available and therefore only smoothed RTT samples, TCP SIAD is still able to operate as desired but might occasionally not be able to fully empty the queue.

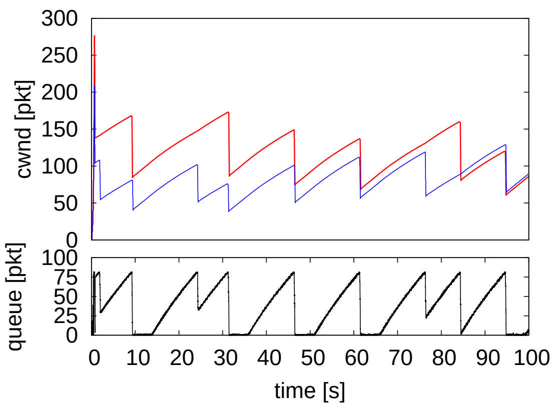
Since we have shown that our design goals have been achieved in these basics scenarios, we in the following evaluate the convergence and capacity sharing properties of TCP SIAD as well as its robustness. Therefore, we evaluate TCP SIAD in more dynamic scenarios with two or more flows using the same or different congestion control schemes in the following sections.

4.4 Capacity Sharing with Multiple Competing Flows

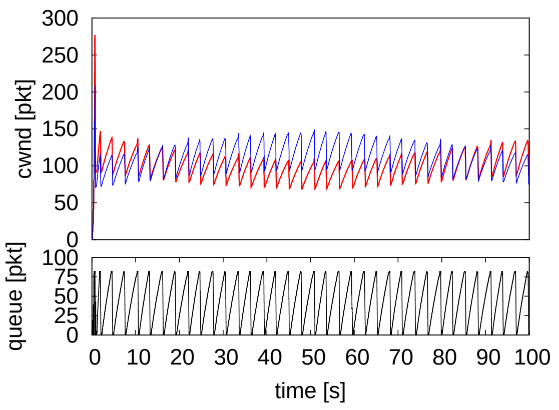
So far, we have analyzed the characteristics of a single TCP SIAD flow and shown that only TCP SIAD fulfills the stated requirements on adaptivity and scalability as well as the resulting desired design goals. Therefore, in this section we focus on capacity sharing between multiple flows. We first investigate the intra-protocol capacity sharing analyzing a scenario with two competing flows using the same congestion control scheme and the same or different Num_{RTT} values, including the case where both flows have different RTTs. Further, we evaluate a scenario where one flow has multiple bottlenecks and finally investigate inter-protocol scenarios.



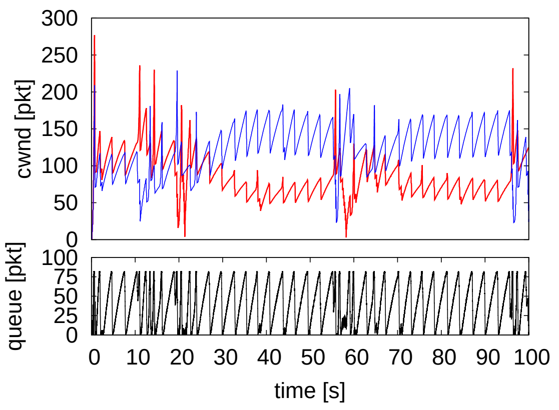
(a) Two TCP NewReno flows without delayed ACKs.



(b) Two TCP NewReno flows with delayed ACKs.



(c) Two TCP SIAD ($Num_{RTT} = 20$) without delayed ACKs.



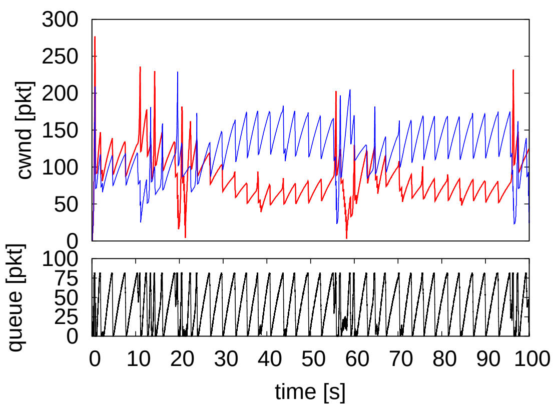
(d) Two TCP SIAD ($Num_{RTT} = 20$) with delayed ACKs.

Figure 4.21: Two flows on 20 Mbit/s link with and without the use of delayed ACKs.

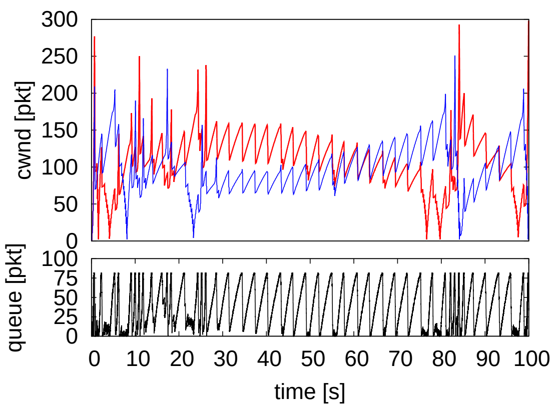
4.4.1 Statistical Evaluation with two Competing Flows

To evaluate scenarios with multiple flows, we perform the same statistical evaluations as in the previous section but based on scenarios with two flows. In this section we use a dumbbell scenario, as described above, with two senders and two receivers. Each of the senders maintains one data-unlimited connection to one of receivers on a 20 Mbit/s bottleneck link with 0.5·BDP buffering. Both flows have a base RTT of 100 ms.

Before we discuss the statistical evaluation, we have to demonstrate de-synchronization that can be observed due to delayed ACKs. Figures 4.21b and 4.21a show two TCP NewReno flows with and without delayed ACKs. It can be seen that both flows are fully synchronized without the use of delayed ACKs. If delayed ACKs are used, however, the TCP NewReno implementation in Linux has only an increase rate of 1/2 packet per RTT. Therefore, both flows together which increase with a rate of about 1 packet per RTT and consequently there is also only one loss per RTT. This means usually only one of the flows gets a congestion notification per congestion event and the flows are not synchronized anymore. While for other schemes with higher increases rates like TCP Cubic the congestion window dynamics are less affected, the influence of delayed ACK is clearly visible for TCP SIAD in Figures 4.21d and 4.21c.



(a) On 20 Mbit/s link with TSOpt.



(b) On 20 Mbit/s link without TSOpt.

Figure 4.22: Two TCP SIAD flows with and without TCP Timestamp Option (TSOpt).

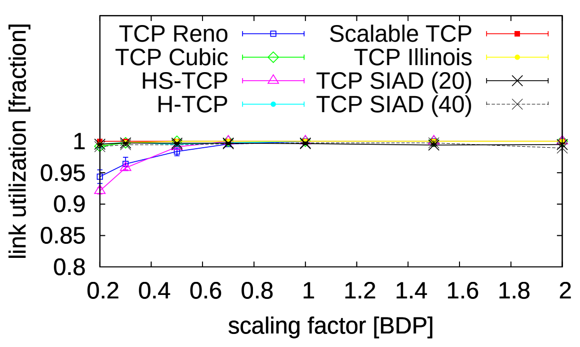
TCP SIAD compensates delayed ACKs based on an estimation of the number of acknowledged packets as described in Section 3.3.3. Subsequently, the traffic is more bursty as always multiple packets are sent out at once. This burstiness leads to the observable higher oscillations in the congestion window development of TCP SIAD with the use of delayed ACKs. While this figure only shows 100 ms of the simulation, we would like to note, that there are over the whole simulation these periods where both flows do not share the capacity equally but there is no periodicity. The length of this period is random and, e.g., influenced by the packet scheduling.

Further, similar as in the previous section, we briefly assess the capacity sharing of TCP SIAD if TSOpt is not available. In Figure 4.22b it can be seen that two competing flows without use of TSOpt on average share the capacity equally. Compared to Figure 4.22a where TSOpt is used the dynamics look very similar. However, there are a few more cases where the queue was not emptied completely due to the smoothed delay estimation.

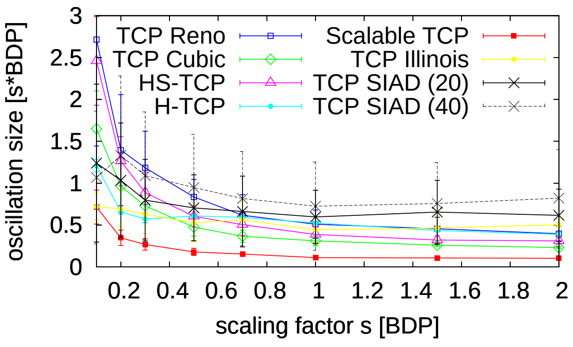
We perform all further investigation with the use of TSOpt. Further, we also use delayed ACKs if not stated differently. In Section 4.4.3 we deactivate delayed ACK to demonstrate equal capacity sharing between TCP Reno and TCP SIAD. While TCP SIAD implements a minimum increase rate of 1 packet per RTT, The Linux implementation of TCP NewReno only increases by 1/2 packet per RTT if delayed ACKs are used. In this case TCP SIAD would always get a larger share than TCP NewReno.

In the following we evaluate TCP SIAD in comparison to other high-speed schemes in the two flow scenario displaying the same statistics as in the previous section. Due to de-synchronization all schemes can achieve a higher link utilization than in the one flow scenario as shown in Figure 4.23. As shown for the one flow scenario, TCP SIAD still maintains high link utilization independent of the configured buffer size.

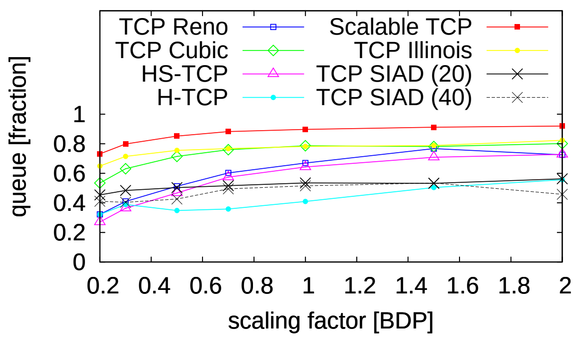
In addition, we also show the oscillation size normalized by the buffer size for this two flow scenario, as oscillations are expected to be irregular and therefore potentially high. In a fully synchronized case for TCP SIAD, we would expect that two competing flows have an average oscillation size of 0.5 of the buffer size. As it can be seen in Figure 4.23b the average oscillation size is higher and also the standard deviation is large. This is partly due to de-synchronization



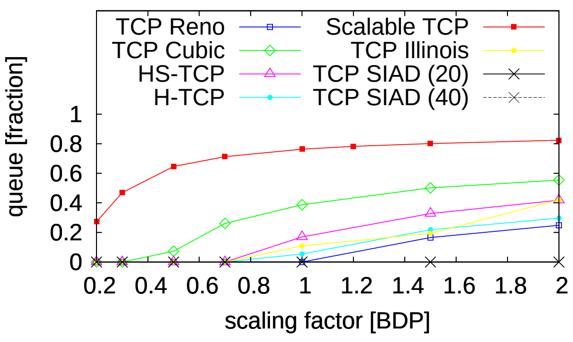
(a) Average link utilization.



(b) Average oscillation size.



(c) Average queue fill level.



(d) Minimum queue fill level.

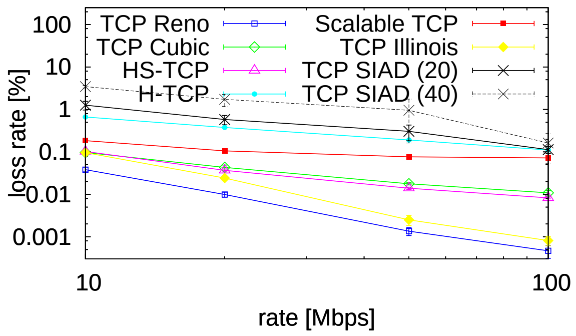
Figure 4.23: Two flows on 20 Mbit/s link.

but is also further amplified due to Fast Increase and Additional Decrease. In case of unsynchronized decreases, Additional Decrease aims to empty the buffer while Fast Increase might subsequently cause an overshoot. For all other schemes the normalized oscillation rate is decreasing with an increase in the provided buffer size as they do not adapt their decrease behavior to the buffer sizes but we normalized the oscillation rate by the buffer size. For small buffer sizes the standard deviation for the oscillation size of TCP NewReno (and High Speed TCP) is also very large. This is again due to de-synchronization.

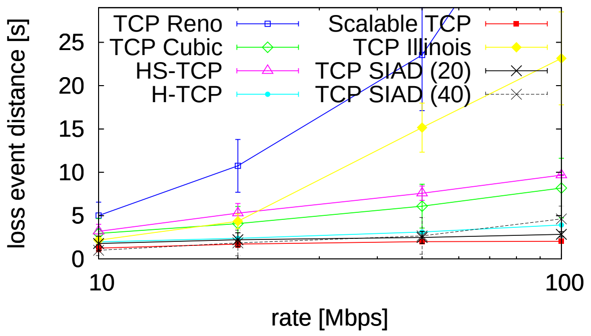
While TCP SIAD still maintains an average queue fill fraction of about 0.5, all other schemes, expect H-TCP, experience in this scenario higher queuing delays due to a larger standing queue in case of unsynchronized window reductions. However, this is not well reflected by the minimum size that is still similar to the results of the one flow scenarios as by chance synchronized decreases happen. Still with larger buffer sizes all schemes, except TCP SIAD, cause a standing queue.

Table 4.1: Mean loss event distance and standard deviation in seconds with buffer size of 0.5·BDP and $Num_{RTT} = 20$.

Mbit/s	mean total	mean flow 1	mean flow 2
10	1.7855 (±0.8446)	1.9235 (±0.8717)	1.9532 (±0.871)
20	2.2191 (±0.8011)	2.3867 (±0.7551)	2.3438 (±0.7646)
50	2.4823 (±0.6931)	2.6175 (±0.6293)	2.5615 (±0.6789)
100	2.84 (±0.3067)	2.8634 (±0.3083)	2.8621 (±0.3112)



(a) Average loss rate.



(b) Average loss event distance.

Figure 4.24: Two flows with 0.5·BDP of buffering.

Figure 4.24b shows that even with two competing flows TCP SIAD implements, as desired, a constant feedback rate according to its configuration. Figure 4.24b illustrates therefore the overall loss distance of both flows together between two congestion events as measured at the bottleneck queue.

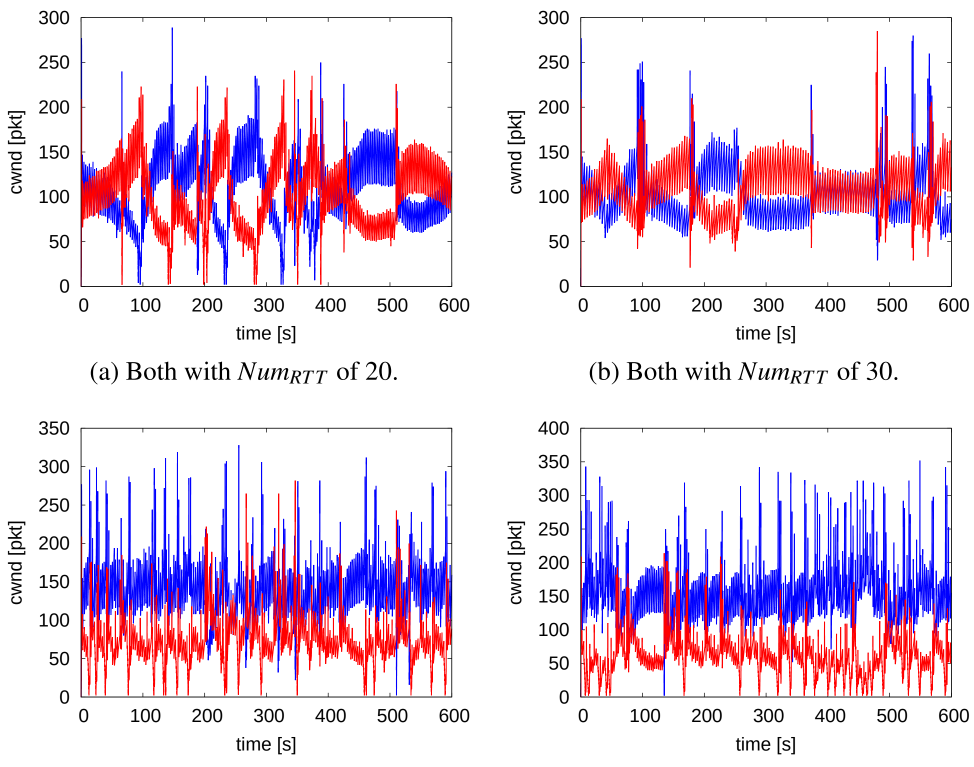
If the loss event distance is measured for each flow separately, as shown in Table 4.1, both flows receive feedback with about the same frequency. The total loss distance is always slightly smaller than the individually seen distances as the common congestion event is usually longer than the one seen by a single flow. Note that in case of a Num_{RTT} value of 40 with a link speed of 20 Mbit/s and below the loss event distance is smaller than configured due to the minimum increase rate of 1 packet per RTT. In Figure 4.24a the total loss rate is displayed. TCP SIAD induces the highest loss rate of all evaluated schemes. This is expected because most schemes also have a much lower feedback rate except Scalable TCP and H-TCP. Scalable TCP, however, maintains the queue at a very high fill level, as clearly visible in Figure 4.23c and 4.23d, which consequently leads to a smaller overshoot with every congestion event. H-TCP has a similar loss rate than TCP SIAD even though the feedback rate is slightly decreasing with higher link speeds.

The results show that the conclusions drawn from the basic one flow scenario in the previous section still hold and therefore only TCP SIAD fulfills all stated design goals, also in multi-flow scenarios.

4.4.2 Intra-Protocol Evaluation incl. different RTT Flows or Multiple Bottlenecks

In this section we focus on how different sharing ratios can be achieved based on TCP SIAD's configuration parameter Num_{RTT}. Further, we evaluate typical scenarios where congestion control schemes often cannot achieve equal rate sharing. Therefore, we investigate RTT-unfairness due to different RTTs of the competing flows and a scenario where one flow passes multiple bottlenecks. We perform multiple simulation runs with different Num_{RTT} configurations to show that Num_{RTT} can be adjusted to help this problem.

In Figure 4.25a and 4.25b we show, as an example, the congestion window and queue length over time for two TCP SIAD flows with the same Num_{RTT} configuration of 20 and, respectively,



(a) Both with Num_{RTT} of 20.

(b) Both with Num_{RTT} of 30.

(c) With Num_{RTT} of 20 and 30.

(d) With Num_{RTT} of 20 and 40.

Figure 4.25: Two TCP SIAD flows on 20 Mbit/s link and with 0.5·BDP of buffering.

30 on a 20 Mbit/s link with an buffer size of 0.5·BDP. In both figures the two flows share the link over the simulation time of 580 s (excluding 20 s start-up phase) in average about equally even though there are periods where one or the other get a larger share for several seconds (up to minutes). Note, this also happens with traditional congestion control, especially if the congestion epochs are very long, e.g., in high-speed networks. In case of a Num_{RTT} configuration of 20 the two flows achieve an average rate of 10.26 Mbit/s and 9.69 Mbit/s. The link utilization is 99.79 % and the average queue fill is 0.5 of the maximum queue length. With a Num_{RTT} configuration of 30 the flows reach an average rate of 9.61 Mbit/s and 10.26 Mbit/s and a link utilization of 99.85 % as well as an average queue fill of 0.51 of the maximum queue length.

Figures 4.25c and 4.25d further show two cases with different Num_{RTT} values for both flows. As desired the flows do not share the capacity equally anymore. In case of values of 20 and 30 one flow gets an average rate of 13.07 Mbit/s while the other one gets 6.86 Mbit/s with a link utilization of 99.66% and an average queue fill of 0.45; With 20 and 40 they reach an average rate of 14.24 Mbit/s and 5.7 Mbit/s with a link utilization of 99.7% and an average queue fill of 0.44.

Tables 4.2 and 4.3 display the ratio of the average rate of the first to the average rate of the second flow for different Num_{RTT} configurations. All values for the scenario depicted above with a link speed of 20 Mbit/s and a buffer size of 0.5·BDP can be found in Table 4.2. Table 4.3 provides the same evaluation with a higher link bandwidth of 100 Mbit/s and smaller queue size of 0.1·BDP buffer size. In both cases on the diagonal there are values about 1 (equal sharing) while in the left lower part all values are below 1 and the right upper part all are above. This

Table 4.2: Sharing ratio of two TCP SIAD flows on 20 Mbit/s link with 0.5·BDP of buffering.

flow 1 \ flow 2	10	20	30	40	50
10	0.95	3.83	4.72	6.25	6.49
20	0.29	1.06	1.91	2.5	2.54
30	0.17	0.47	0.93	1.48	1.49
40	0.19	0.38	0.68	0.98	1.09
50	0.14	0.4	0.67	0.97	1.05

Table 4.3: Sharing ratio of two TCP SIAD flows on 100 Mbit/s link with 0.1·BDP of buffering.

flow 1 \ flow 2	10	20	30	40	50
10	0.93	2.95	6.14	4.35	4.57
20	0.27	1.05	1.71	1.73	1.81
30	0.3	0.62	1.11	1.25	1.38
40	0.19	0.61	0.84	1.05	1.09
50	0.20	0.68	0.66	0.96	1.06

means, as expected, always the flow with the smaller Num_{RTT} gets the larger share. Note that, e.g., when competing with a flow with a Num_{RTT} of 10, an increase in aggressiveness from 50 to 40 might not lead to a higher share, as it can be seen in the tables. This is because in both cases the flow with larger Num_{RTT} value already operates mostly with the minimum increase rate of 1 packet per RTT; while the configuration would actually require a lower increase rate. Moreover, these values also show that within the simulation time of 600 s there was not always fully equal sharing achieved (values are between 0.93 and 1.11). This is due to the high window dynamics of TCP SIAD especially in cases with large buffers and subsequently long periods where the link is not shared equally. Still, as expected, in both tables the values are getting smaller to lower left corner and larger to the upper right corner. This demonstrates well that TCP SIAD's configuration parameter Num_{RTT} can be used to influence the capacity sharing between competing flows. Now, a higher-layer control loop could adjust Num_{RTT} during the transmission time to better cope with application requirements like a minimum sending rate.

In the following, we evaluate RTT-(un)fairness. We investigate two flows competing on a 20 Mbit/s link with one of the flows having an RTT of 100 ms while the other one has a twice as large RTT of 200 ms. The buffer is configured to one BDP based on a mean RTT of 150 ms. RTT-unfairness is a problem of many existing schemes. This means equal rate sharing cannot be achieved anymore if two competing flows operate based on different base RTTs. This can also be seen in Table 4.4 where we show the mean sending rate (as well as the confidence interval) for the two flows for all different schemes in test. Moreover, we calculated a fairness value based on Jain's Fairness Index, as explained in Section 2.5.3, where 1 stands for maximum fairness. Only TCP Cubic and H-TCP can achieve a high fairness value. Note, especially TCP Cubic was explicitly designed to address RTT-unfairness. TCP SIAD can only achieve a fairness value of 0.71 when both flows have the same Num_{RTT} configuration of 20. However, if we configure a value of 10 for the flow with larger RTT of 200 ms and a value of 40 for the other one with 100 ms RTT, TCP SIAD achieves a very high fairness value. As the right con-

Table 4.4: Two flows with different RTTs (100 ms and 200 ms) on 20 Mbit/s link.

	mean rate 1	mean rate 2	fairness
TCP NewReno	15.13 (±0.73)	4.87 (±0.73)	0.7917
TCP Cubic	12.73 (±0.48)	7.27 (±0.48)	0.9307
H-TCP	11.74 (±0.13)	8.26 (±0.13)	0.9704
High Speed TCP	16.92 (±0.18)	3.08 (±0.18)	0.6765
Scalable TCP	18.52 (±0.10)	1.48 (±0.10)	0.5894
TCP Illinois	13.52 (±1.18)	6.49 (±1.18)	0.8900
TCP SIAD (20/20)	15.63 (±0.49)	4.27 (±0.48)	0.7542
TCP SIAD (20/10)	8.84 (±.37)	10.95 (±1.33)	0.9888
TCP SIAD (40/10)	6.97 (±1.23)	12.74 (±1.21)	0.9212
TCP SIAD (40/20)	8.08 (±1.52)	11.72 (±1.46)	0.9673
TCP SIAD (Num_{MS}=4000)	10.93 (±2.2)	8.95 (±2.17)	0.9902
TCP SIAD (Num_{MS}=5000)	11.66 (±1.61)	8.11 (±1.53)	0.9687

figuration cannot be found by both flows in a distributed manner, it is actually more sensible to set the configuration not based on RTTs but based on absolute time if equal sharing is desired in this scenario. If we configure the desired congestion epoch time to 4000 ms for both flows, TCP SIAD achieves a fairness of 0.9902.

Another scenario where traditional congestion control schemes often do not achieve equal capacity sharing is a scenario where one flows crosses multiple bottlenecks while the competing flows on each bottleneck only has to cope with one bottleneck. We evaluate this scenario based on a parking lot topology as described in Section 2.5.2.2 and as also further detailed for our evaluation setup description above. Each bottleneck link as well as the access link at sender and receiver side for the cross traffic and the return link of each flow have an delay of 10 ms such that each flow experiences the same base RTT of 60 ms. The link bandwidth of each bottleneck link is 10 Mbit/s; therefore in the best case each flow should get a rate of about 5 Mbit/s. The buffer size is configured to 1·BDP. In such a case the flow crossing multiple bottlenecks receives more often congestion notifications (from the different bottlenecks) and consequently with traditional congestion control schemes it reduces its sending rate more often than the other flows. This effect is demonstrated for TCP NewReno, TCP Cubic and H-TCP in Table 4.5 where the average sending rate of flow 0 is much lower than for the other three crossing flows. In this case, the fairness is also calculated using Jain's Fairness Index but based on two groups of flows. One group is the flow that is passing multiple bottlenecks while the others that cross only one bottleneck represent the second group. Similar as above by adjusting the Num_{RTT} configuration of TCP SIAD we can again achieve an about equal sharing and therefore a high fairness value.

Finally as an example in Figure 4.26 we show ten TCP SIAD flows competing on a 100 Mbit/s link and a maximum buffer size of 0.1·BDP. As an example, the red line shows the congestion window development of the flow that puts the first packet on the link. While this flow can grab slightly more capacity at the beginning, it still has (only) an average rate of 10.67 Mbit/s over the total simulation time of 580s (without start-up phase). Therefore, competing TCP SIAD flows are able to share the capacity about equally as shown in detail for this ten flow scenario in appendix C in Table C.2.

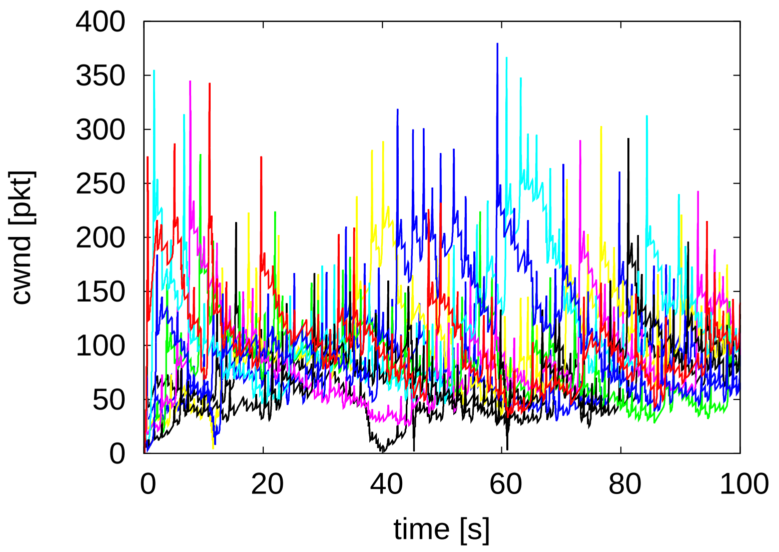


Figure 4.26: 10 TCP SIAD flows on 100 Mbit/s link and with 0.1·BDP of buffering.

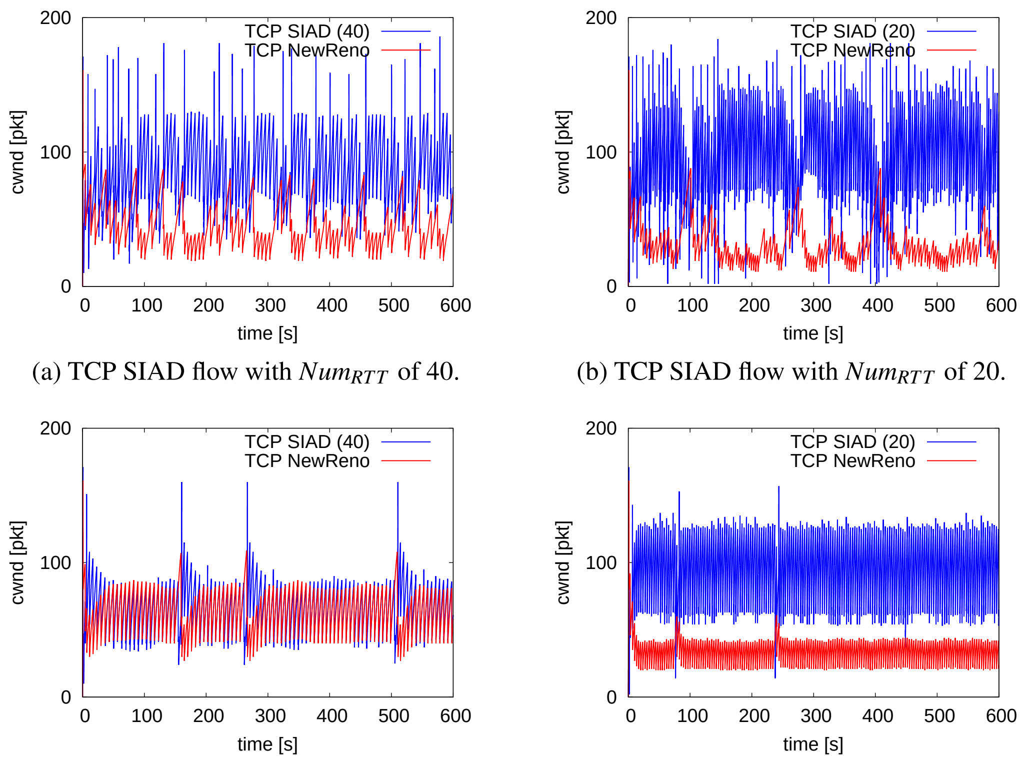
In this section we have shown that Num_{RTT} can be used to influence the capacity sharing between TCP SIAD flows. In the following we now analyze capacity sharing of TCP SIAD with non-SIAD cross traffic.

4.4.3 Inter-protocol Evaluation with Competing Non-SIAD flows

To evaluate inter-protocol capacity sharing we investigate scenarios where one TCP SIAD flow competes with TCP NewReno and TCP Cubic cross traffic as these congestion control schemes are the most common representatives that can be found in the current Internet [149]. While TCP SIAD is not designed to provide fair/equal capacity sharing with traditional TCP (NewReno) cross traffic, we at least would like to demonstrate that, when competing with TCP SIAD, the traditional schemes still gets a reasonable share. Further, we also show that TCP SIAD can be configured to share the bottleneck link equally with other schemes (if, e.g., some network and traffic conditions are known). Therefore, we choose the Num_{RTT} setting such that it has the about same feedback rate as a cross traffic flow induces in the scenario under evaluation when it would be allocating half of the link capacity. E.g., in case of an increase rate of 1 packet

Table 4.5: Flow 0 experiencing multiple bottlenecks with each having a link bandwidth of 10 Mbit/s.

	mean rate 0	mean rate 1	mean rate 2	mean rate 3	fairness
NewReno	3.14 (±0.57)	6.85 (±0.57)	6.85 (±0.57)	6.86 (±0.57)	0.8788
Cubic	2.46 (±1.03)	7.53 (±1.03)	7.53 (±1.03)	7.54 (±1.03)	0.7956
H-TCP	3.04 (±0.22)	6.91 (±0.23)	6.94 (±0.22)	6.95 (±0.22)	0.8682
SIAD (20)	1.67 (±0.17)	8.07 (±0.23)	8.13 (±0.19)	8.17 (±0.2)	0.697
SIAD (40)	1.83 (±0.19)	7.82 (±0.24)	7.93 (±0.23)	7.94 (±0.21)	0.7204
SIAD (10/40)	4.13 (±0.35)	5.15 (±0.42)	5.43 (±0.39)	5.55 (±0.35)	0.9831
SIAD (5/40)	4.74 (±0.25)	4.23 (±0.31)	4.68 (±0.27)	4.93 (±0.26)	0.9998



(a) TCP SIAD flow with Num_{RTT} of 40.

(b) TCP SIAD flow with Num_{RTT} of 20.

(c) TCP SIAD flow with Num_{RTT} of 40 and without delayed ACKs.

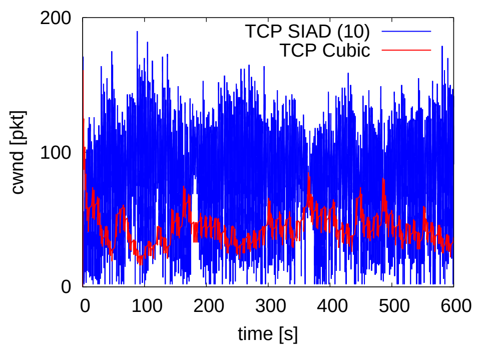
(d) TCP SIAD flow with Num_{RTT} of 20 and without delayed ACKs.

Figure 4.27: One TCP SIAD flow competing with one TCP NewReno flow on 10 Mbit/s link with 1·BDP of buffering.

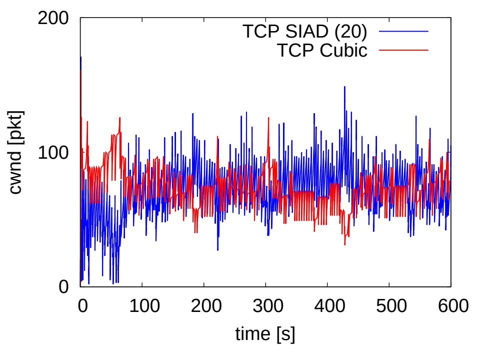
per RTT and halving on congestion as TCP NewReno would do, a congestion epoch length of about 41 packets would be achieved when allocating 5 Mit/s with 1·BDP of buffer before the bottleneck link and an RTT of 100 ms. Therefore, we expect about equal sharing for a TCP SIAD Num_{RTT} configuration of about 40.

In Figure 4.27 one TCP SIAD and one TCP NewReno flow share the same bottleneck with a link bandwidth of 10 Mbit/s and a maximum queue size of 1·BDP. If delayed ACKs are used, TCP SIAD gets always a larger share than TCP NewReno. This is because TCP SIAD implements a minimum increase rate of 1 packet per RTT while the Linux implementation of TCP NewReno effectively only has an increase rate of 1/2 a packet per RTT. Still, the share that TCP SIAD grabs can be influenced by the Num_{RTT} configuration. If TCP SIAD is configured with a Num_{RTT} value of 40, it achieves a share of 6.53 Mbit/s compared to 3.47 Mbit/s for the TCP NewReno flow which is about double the rate as expected. With a Num_{RTT} of 20 the TCP SIAD flow has an higher average rate of 7.47 Mbit/s and consequently the TCP NewReno flow only of 2.46 Mbit/s. Note, using a larger value than 40 would not help in this situation as TCP SIAD with $Num_{RTT} = 40$ already operates at the minimum increase rate of 1 packet per RTT.

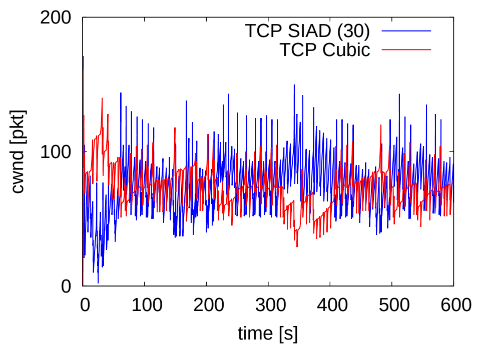
If we, instead, disable the delayed acknowledgements mechanism the Linux NewReno flow reaches an increase rate of 1 packet per RTT. In Figure 4.27c TCP SIAD is again configured with Num_{RTT} set to 40. Now it can be seen that both flows share the link about equally; more precisely the NewReno flow has a share of 5.06 Mbit/s and TCP SIAD of 4.93 Mbit/s. In Fig-



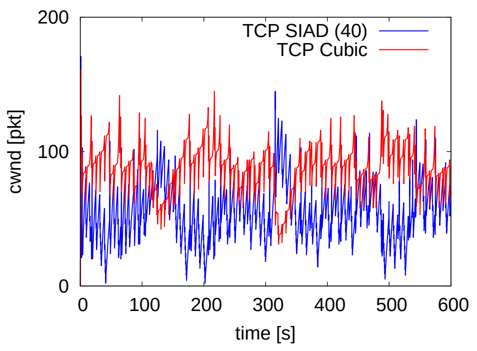
(a) TCP SIAD flow with Num_{RTT} of 10.



(b) TCP SIAD flow with Num_{RTT} of 20.



(c) TCP SIAD flow with Num_{RTT} of 30.



(d) TCP SIAD flow with Num_{RTT} of 40.

Figure 4.28: One TCP SIAD flow competing with one TCP Cubic flow on 10 Mbit/s link with 1·BDP of buffering.

ure 4.27d Num_{RTT} is set to 20. Here TCP SIAD gets again a larger share of 7.35 Mbit/s while the TCP NewReno flow has an average rate of 2.63 Mbit/s.

In Figure 4.28d the scenario is the same as before where roughly equal sharing was reached when competing with TCP NewReno; but this time TCP Cubic is used instead. We can see that TCP Cubic is more aggressive and therefore gets the larger share. While TCP SIAD with a Num_{RTT} value of 40 has an average rate of 3.77 Mbit/s only, the TCP Cubic flow has a throughput of 6.23 Mbit/s on average. If we set the Num_{RTT} to 30 in Figure 4.28c, we can again achieve about equal sharing with 4.98 Mbit/s for the TCP SIAD flow compared to 5.02 Mbit/s for the TCP Cubic flow. Or if we choose an even smaller value of 10, TCP SIAD achieves the larger share with 6.44 Mbit/s compare to 3.43 Mbit/s for the TCP Cubic flow. Further, results on capacity sharing with TCP NewReno and TCP Cubic are shown in the appendix in Table C.5, C.6, and C.7.

4.4.4 Conclusion

This section shows that TCP SIAD can still fulfill the desired design goals when multiple flows are competing for the same bottleneck even though TCP SIAD induces higher dynamics mainly due to Fast Increase and Additional Decrease. Further, we have demonstrated that the Num_{RTT} configuration parameter of TCP SIAD can be used to impact the capacity share between competing flows using the same or a different congestion control schemes. The configuration parameter Num_{RTT} either directly defines the feedback rate (when alone on the bottleneck link) or the share of the capacity between the competing flows. In case cross traffic imposes additional congestion with a higher frequency, the configured rate cannot be reached anymore. However, TCP SIAD is able to achieve a stable capacity share between competing flows (on the cost of higher congestion).

As discussed in Section 3.4.3, Num_{RTT} could therefore be used by a higher-layer control loop to better enforce application requirements while we anticipate ingress policing, e.g., based on congestion, to implement (per-user) fairness in the future Internet.

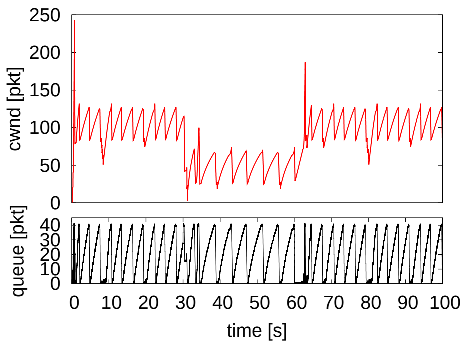
In the following section we evaluate more dynamic scenarios with either rate changes or starting and stopping of cross traffic. This scenario is used to assess our stated requirement on convergence.

4.5 Convergence and Responsiveness to Starting Traffic

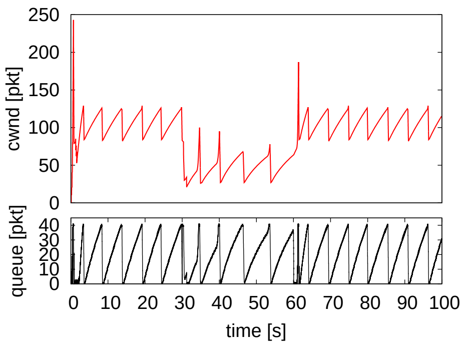
Having assessed efficiency regarding scalability and adaptivity as well as the capacity sharing capabilities of TCP SIAD, we now evaluate convergence properties in comparison to TCP NewReno, TCP Cubic, and H-TCP. We expect similar results for H-TCP and TCP SIAD, as both schemes implement (to some extend) the same decrease behavior and some kind of Fast Increase behavior. The increase behavior is most important when capacity resources become newly available that subsequently need to be allocated by the running flow(s), e.g., due to stopping cross traffic. In contrast, the decrease behavior is important when new starting cross traffic attempts to grab its share of the capacity. In this case, the convergence time depends strongly on how fast the running flow can release capacity to the new flow. We evaluate these transient effects based on three scenarios with either changing rate conditions, CBR cross traffic or one new starting flow that uses the same congestion control mechanism as the existing flow that is already fully utilizing the link resources at that point of time.

4.5.1 Transient Behavior due to Rate Changes or CBR Cross Traffic

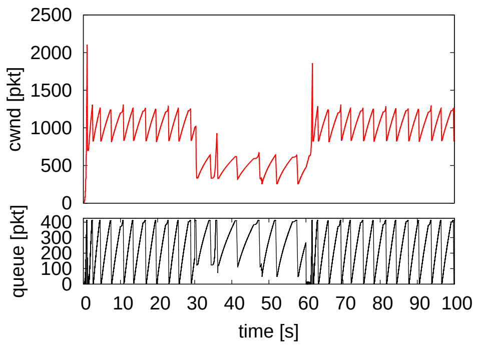
Our initial test simply changes the link rate while one flow is active. This can, e.g., happen due to route changes. Figures 4.29a and 4.29b show a scenario where the link has at the beginning a bandwidth of 10 Mbit/s, while it is changed to 3 Mbit/s after 30 s and back to 10 Mbit/s at a simulation time of 60 s. The buffer is sized to 0.5·BDP based on the initial bandwidth of 10 Mbit/s. Two cases are shown for TCP SIAD with a Num_{RTT} value of 20 and 40. In both cases TCP SIAD can quickly adapt to a lower rate. Due to Additional Decrease it is able to fully empty the queue and consequently immediately switch to a larger decrease factor. The convergence time in the case where new capacity is available depends on the point of time in the congestion epoch at which the rate change happens as well as on the Num_{RTT} value. In fact, as it can be seen by the length of the time period that the queue is empty after the rate adjustment in these simulation runs that the TCP SIAD configuration with a Num_{RTT} value of 40 converges faster. This is because rate adjustment happens right at the end of a congestion epoch where the congestion window is already maximized. Further, it can be seen that as soon as Fast Increase is entered, TCP SIAD quickly allocates the new capacity. Therefore, we



(a) With $Num_{RTT} = 20$ and rate changes from 10 Mbit/s to 3 Mbit/s and back.



(b) With $Num_{RTT} = 40$ and rate changes from 10 Mbit/s to 3 Mbit/s and back.



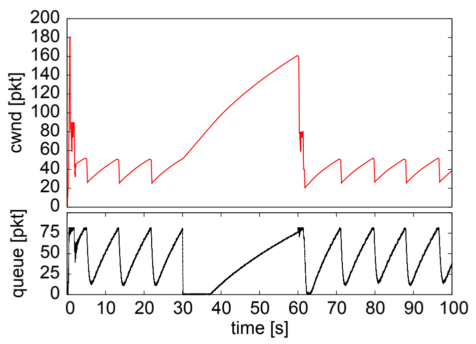
(c) With $Num_{RTT} = 20$ and rate changes from 100 Mbit/s to 25 Mbit/s and back.

Figure 4.29: One TCP SIAD flow with different Num_{RTT} configurations and rate changes.

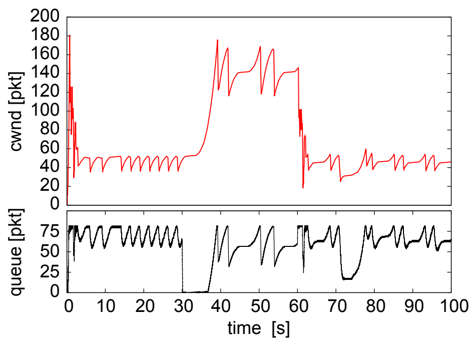
have shown that TCP SIAD can fulfill our design goals on fast bandwidth allocation in case of changing traffic conditions.

Similar to the previous scenario we also investigate a 100 Mbit/s link and 0.5·BDP buffer size with a rate change to 25 Mbit/s and back. In this case TCP SIAD cannot immediately adapt the decrease factor correctly because of the too low time stamp resolution. This means it can of course still utilize the link but not empty the queue completely as Additional Decreases are performed right at the beginning. Often a short overshoot due to Fast Increase can still trigger Additional Decrease later on. At 60 s, even though the rate has be increased by 75 Mbit, TCP SIAD can re-allocate the capacity quickly within a few seconds.

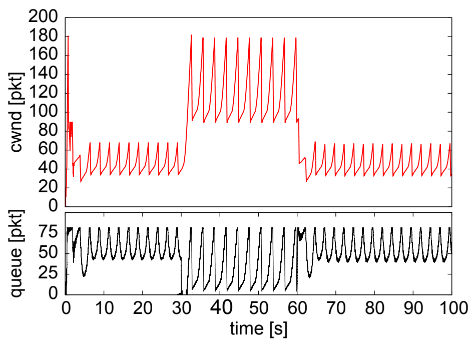
Inspired by the TCP Evaluation Suite [13, 63], we further have a look at a scenario where CBR cross traffic stops at a certain point in time and later starts sending again. While it is proposed to investigate a step decrease from 75 Mbit/s to 0 Mbit/s and the respective increase step back to 75 Mbit/s on a 100 Mbit/s link, we first examine a scenario with a lower bandwidth, similar to the scenario above, to be able to compare to TCP NewReno. Scalability in the sense of grabbing new available capacity in Congestion Avoidance is a well-known problem of TCP NewReno as depicted in the introduction in Section 1.1.1, therefore it would take TCP NewReno a very long time to converge on the 100 Mbit/s link. In contrast high-speed schemes as TCP Cubic and also H-TCP are developed to grab newly available capacity fast and therefore work better on, e.g., a 100 Mbit/s bottleneck link, as shown afterwards.



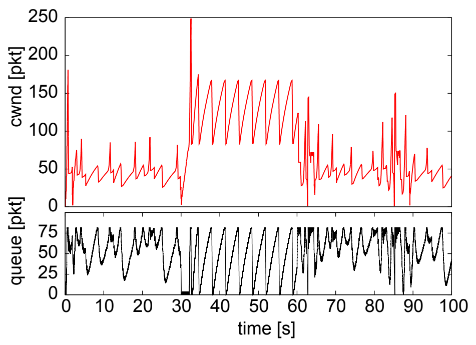
(a) TCP NewReno.



(b) TCP Cubic.



(c) H-TCP.

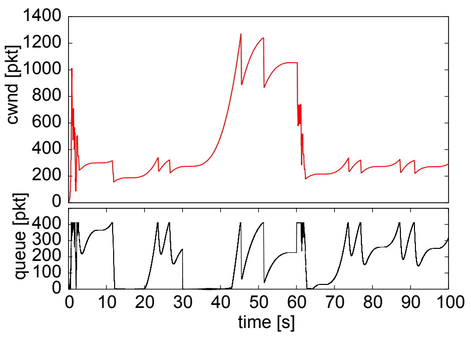


(d) TCP SIAD with $Num_{RTT} = 20$.

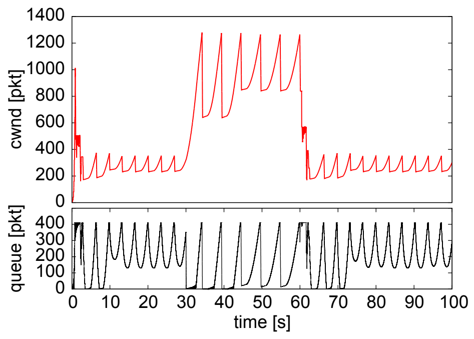
Figure 4.30: One Flow on 10 Mbit/s link with 7 Mbit/s CBR traffic till 30 s and from at 60 s on.

Figure 4.30 shows the congestion window of the congestion-controlled flow and bottleneck queue length on a 10 Mbit/s link with a buffer size of 1·BDP at an RTT of 100 ms. From the beginning of the simulation until 30 s of the simulation time a second flow sends CBR traffic with a rate of 7 Mbit/s and restarts at 60 s. These stop and start times just provide one sample measurement point for this scenario as the actual convergence time also depends, sometimes strongly, on the point of time in the congestion epoch at which the CBR traffic stops. It is not trivial to examine the convergence time objectively as various samples (equally) distributed over one congestion epoch would need to be taken. Unfortunately the length of a congestion epoch is very different for each scheme and sometimes also depend on the current network situation. For a first assessment, we consider the scenario described above and, for a start, discuss general convergence characteristics of the schemes in test.

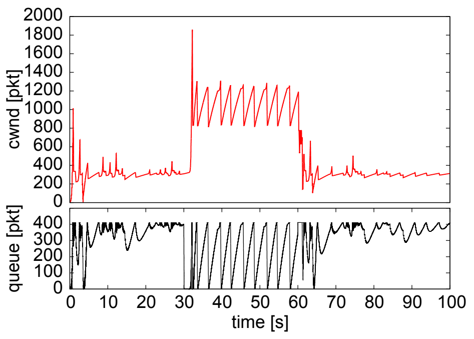
Both traditional schemes, TCP Cubic and TCP NewReno, cannot utilize the link after stopping of the CBR traffic for several seconds. This can be clearly seen in Figures 4.30a and 4.30b at 30 seconds simulation time, the queue length is zero for a quite long time (several seconds). TCP NewReno needs 6.2 s to reach a rate larger than 9.5 Mbit/s and TCP Cubic 6.6 s. In contrast, H-TCP and TCP SAID with a Num_{RTT} value of 20 need only 1.6 s and, respectively, 2.2 s. Note, this measurement has only a resolution of 200 ms as this is our measurement interval for the average rate. While the slow increase rate of TCP NewReno is a known problem, TCP Cubic is designed to scale better but unfortunately spends a long time at the plateau around the target congestion window value. However, even though TCP NewReno and TCP Cubic have about the same convergence time in this scenario, convergence is much worse for TCP NewReno with higher bandwidth links, as explained below.



(a) TCP Cubic.



(b) H-TCP.



(c) TCP SIAD flow with Num_{RTT} of 20.

Figure 4.31: One flow on 100 Mbit/s link with 75 Mbit/s CBR traffic till 30 s and from 60 s on.

In Figure 4.31 a 100 Mbit/s link and 0.5·BDP of buffer space are used. In this scenario TCP NewReno can not fully utilize the link anyway and can therefore only reach a maximum rate of 43.42 Mbit/s within the first 30 s where no CBR cross traffic is on the link. TCP Cubic has still a quite long convergence time (to reach a sending rate of more than 95 Mbit/s) of 12.8 s, while H-TCP and TCP SIAD ($Num_{RTT} = 20$) can maintain a convergence time in the same range than in the smaller bandwidth scenario of 2.6 s and, respectively, 2.2 s. Note, this time TCP SIAD converges faster but mainly, as explained above, because the TCP SIAD flow has by chance already a larger rate at the point of time when the CBR traffic stops. Appendix C.4 also shows traces for the same network scenarios but with two TCP flows using the same congestion control scheme that are competing with CBR cross traffic. We can conclude that Fast Increase provides quick bandwidth allocation as desired while especially TCP Cubic has a long convergence time in the shown scenario. Therefore, we see the concept of a separate Fast Increase phase similar in the case where no information about the available capacity is known more as general concept that could be used in new congestion control schemes.

In case of starting CBR traffic in [13, 63], it is proposed to evaluate the harm that is caused to the CBR traffic when a congestion control scheme decreases its sending rate too slowly. Therefore, it is proposed to count the number of losses seen by the CBR traffic in the first 100 s after starting. As it can be seen, e.g., for TCP Cubic and TCP SIAD in Figure 4.31, schemes designed for high-speed networks are usually able to adapt within the first few seconds. Therefore, we compare the number of losses seen by the CBR in the first 5 seconds. For the scenario described above, one TCP Cubic flow causes 1314 losses, one H-TCP flow 990 and one TCP SIAD flow 1476 to the CBR cross traffic. Additionally, in Table 4.6 we display the total loss rate for both together the TCP traffic and the CBR cross traffic for the first couple (up to 5) seconds and

Table 4.6: Loss rate when CBR traffic starts on 100 Mbit/s link.

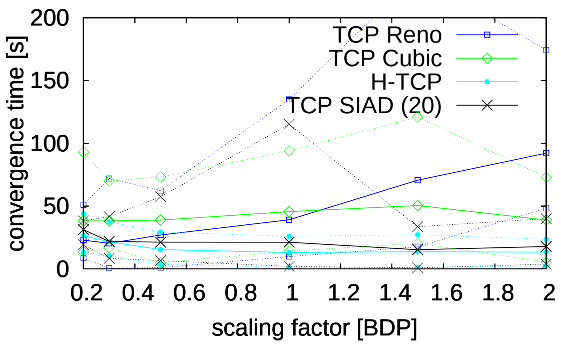
	·BDP	1 s	2 s	3 s	4 s	5 s
TCP Cubic	0.3	35.92%	33.01%	27.46%	22.49%	18.93%
	0.5	37.58%	30.66%	24.32%	20.11%	16.76%
	1.0	35.94%	31.99%	27.25%	23.13%	19.42%
	1.2	40.64%	35.26%	29.63%	24.09%	20.06%
H-TCP	0.3	30.41%	27.93%	24%	19.59%	16.34%
	0.5	33.89%	29.12%	23.01%	20.01%	16.56%
	1.0	23.49%	21.73%	15.78%	12.24%	9.97%
	1.2	16.56%	13.33%	9.87%	7.56%	6.04%
TCP SIAD (20)	0.3	37.03%	35.93%	30.25%	25.59%	22.74%
	0.5	39.96%	30.82%	24.11%	20.92%	17.48%
	1.0	35.53%	29.12%	21.38%	18.37%	17.14%
	1.2	31.01%	27.21%	20.63%	18.66%	16.08%
TCP SIAD (40)	0.3	36.98%	32.97%	26.98%	22.71%	19.32%
	0.5	31.42%	29.6%	24.44%	19.03%	17.33%
	1.0	32.78%	27.38%	22.32%	18.9%	15.89%
	1.2	26.98%	24.16%	18.12%	16.92%	14.42%

for different buffer configurations. Of course, the higher the Num_{RTT} configuration value is, the lower is the loss rate as this configuration makes TCP SIAD less aggressive. Further, TCP SIAD induces a lower loss rate, the larger the buffering is. This is because the decrease factor is also larger and therefore the increase step per RTT is smaller. For H-TCP this is only true in the range between 0.5 and 1·BDP of buffering. Further, as with the convergence time the size of the overshoot is also determined by the starting time of the CBR traffic. Appendix C.4 show the same table for a 10 Mbit/s link. In this table it can clearly be seen that, e.g., the loss rate for H-TCP is again higher with 1.2·BDP of buffering. We conclude that TCP SIAD induces a reasonable loss rate as its loss rate is comparable to TCP Cubic in small buffer scenarios or even lower otherwise.

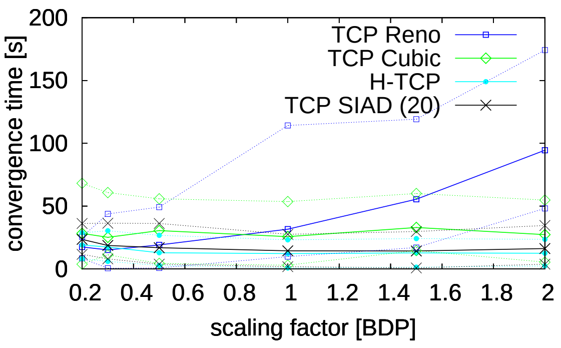
4.5.2 Evaluation of Convergence Times with Adaptive Cross Traffic

To provide a more general evaluation of the convergence properties we use a scenario on a 20 Mbit/s link with one flow that starts right at the beginning of the simulation and a second flow that starts later on during the simulation run. Both flows always use the same congestion control scheme in this investigation. We have performed 20 simulation runs for each setup with different start times at each full second between 20 s and 39 s of the simulation time. Please note, this measurement provides a better assessment than in the previous section but still does not allow for a fair comparison. E.g., while 19 s covers several congestion epochs for SIAD independent of the available bandwidth, the same setup catches only a part of one congestion epoch for TCP NewReno setups with high BDPs.

Figure 4.32a shows the mean convergence time to reach 95% of the desired equal bandwidth share (of 10 Mbit/s for each flow) as well as the absolute minimum and maximum seen for these sample set depending on the configured buffer scaling. While the convergence times for H-TCP



(a) 95% average, min, and max convergence time.



(b) 80% average, min, and max convergence time.

Figure 4.32: Convergence time of two flows on 20 Mbit/s link with different start times.

and TCP SIAD (with $Num_{RTT} = 20$) still lie in the same range, TCP Cubic, as expected from the first assessment above, has always a higher average convergence time. In contrast, TCP NewReno has even a smaller convergence time for small buffers as the decrease factor of 0.5 is in this case quite large compared to the buffer size. But with smaller buffers than 1·BDP, TCP NewReno is not able to utilize the link fully. Therefore, the buffer is often empty and thus all the buffer space can be used by the starting flow. With increasing buffer sizes, the maximum BDP increases as well and TCP NewReno needs subsequently a longer time for convergence. Both H-TCP and TCP SIAD provide, in comparison, short average convergence times and about the same average convergence time for all buffer settings. However, TCP SIAD has a higher maximum value that means that there was at least one strong outlier. As there are no outliers in Figure 4.32b anymore which shows the convergence time to reach 80% of the fair share, TCP SIAD can grab a reasonable share quickly. As TCP SIAD also operates for quite long time periods in different sharing ratio in steady state, as we have observed in previous sections, this observation is explainable.

Appendix C.5 shows the same effects for TCP SIAD with $Num_{RTT} = 40$ as well as for a the convergence time to reach 80% of the desired bandwidth. Further, selected congestion window and queue length traces are shown for the 20 Mbit/s scenario as well as for a 100 Mbit/s scenario. For 100 Mbit/s TCP SIAD's convergence times are higher but still in the same range.

The TCP Evaluation Suite [13, 63] proposes to examine convergence with starting cross traffic that has Slow Start disabled by setting the Slow Start threshold to the initial congestion window value of 10 packets. This is supposed to cover the worst case where a congestion notification is received within the first RTT, e.g., because the running flows have already filled up the queue completely at the starting time of the new flow. We covered this case by evaluating a large set of start times and the results are reflected by the maximum convergence time as given above. However, Appendix C.5 shows the same figure than above for the average, minimum, maximum convergence time to reach 80% or 95% of the equal sharing rate but with Slow Start disabled. Depending on the RTT, queue size and increase behavior disabling Slow Start adds a couple of seconds to the average convergence time for TCP NewReno and TCP Cubic. In contrast, for TCP SIAD disabling Slow Start only adds a few RTTs as in this case TCP SIAD starts in Fast Increase phase instead. This means, TCP SIAD will also start with an increase rate of 1 packet per RTT, instead of 10 packets per RTT, but still doubles its increase rate per RTT. Therefore, TCP SIAD needs five (additional) RTTs to reach an increase rate of 16 packets per RTT but also already a congestion window of 25 at that point of time.

4.5.3 Conclusion

In this section we evaluated the convergence properties of TCP SIAD in dynamic scenarios. We have shown that due to Additional Decrease TCP SIAD can quickly adapt its decrease factor when the available capacity is decreased such that the queue is emptied as desired (if the time stamp resolution is sufficient). If capacity is allocated by newly starting CBR traffic TCP SIAD might induce a rather high loss rate (similar as TCP Cubic) as Fast Increase can cause a large overshoot. However, we have shown that Fast Increase is necessary to allocate newly available capacity quickly in high-speed networks. While TCP Cubic, even though it is designed for high-speed networks, takes a long time to converge, TCP SIAD has in all scenarios a convergence time of only about 2 seconds. Regarding convergence time, H-TCP and TCP SIAD provide similar performance while clearly outperforming TCP Cubic which is currently the most widely-used high-speed congestion control scheme.

4.6 Robustness in High-Congestion Traffic Scenarios

In addition to congestion control requirements like efficiency (scalability and adaptivity), capacity sharing, and convergence we aim to assess the robustness of TCP SIAD. Robustness is especially important regarding the question if TCP SIAD is suitable to be used (for experimentation at least) in the Internet where a wide range of different scenarios can occur. However, as an evaluation of robustness is not straight-forward, our approach is to evaluate TCP SIAD in extreme scenarios that are rare in the wild but, when proven to work, do also cover those scenarios that can actually occur in the Internet.

Note, we already evaluated the robustness of TCP SIAD against delay variations in Section 4.3.2 and showed that TCP SIAD as a hybrid approach is stronger influenced than pure loss-based schemes but still provides reasonable performance and we show that TCP flows never full starve without getting any capacity anymore.

In this section we aim to evaluate TCP SIAD's robustness against loss and therefore perform simulations with additional artificial random loss as well as with short flow cross traffic that induces traffic bursts on the link and therefore also causes large (and infrequent) burst losses. Moreover, we investigate the interaction with a set of AQM mechanisms that also induce different loss patterns. This is especially important as it can be expected that AQM will be found more often in the future Internet regarding ongoing activities in standardization [18].

4.6.1 Impact of High Loss Rates

In this section we focus on the influence of high loss rates on TCP SIAD induced by two effects, either due to bursty cross traffic or, even worse, due to errors in lower layers which can cause packet drops that are not related to congestion.

Table 4.7: Link utilization and loss rate for one flow and short flow cross traffic on 10 Mbit/s link using different buffer configurations.

	buffer size [·BDP]	utilization	loss rate
TCP NewReno	0.3	43.05%	3.665%
	0.5	59.37%	2.476%
	1.0	89.44%	1.26%
TCP Cubic	0.3	77.18%	2.91%
	0.5	89.69%	1.425%
	1.0	99.19%	0.798%
H-TCP	0.3	87.88%	1.586%
	0.5	93.32%	1.579%
	1.0	90.27%	2.433%
TCP SIAD (10)	0.3	95.74%	2.834%
	0.5	94.48%	3.858%
	1.0	94.77%	3.854%
TCP SIAD (20)	0.3	94.74%	2.535%
	0.5	93.64%	3.156%
	1.0	91.62%	3.362%
TCP SIAD (40)	0.3	95.06%	2.979%
	0.5	93.06%	3.784%
	1.0	91.08%	4.223%

A long-living flow can get (severely) disturbed by (a large number of) short traffic peaks that fill up the queue and therefore signal congestion but do not last long enough to utilize the capacity that was consequently freed by the long-living flow. As soon as the queue overflows (or grows large enough to imply an AQM reaction by either drops or ECN marks) the long-living flow receives a congestion notification and reduces its sending rate. Unfortunately, if those small bursts only remain active for less than an RTT, there is already no cross traffic anymore to potentially allocated the freed bandwidth when the load of the long-living flow is finally reduced.

To further evaluate this effect, we modeled a scenario where short flows of 300.000 Bytes appear with an equally distributed Inter-Arrival Time (IAT) of 2 s to 3 s on a 10 Mbit/s link with an RTT of 100 ms for each flow. With a buffer size of one base BDP this scenario has a maximum BDP of 250.000 Bytes. This means, one short burst would not leave Slow Start (due to the exponential increase and an initial window of 10) when alone on the link but is large enough to fill the buffer when partly already allocated by long-living traffic.

As it can be seen in Table 4.7 TCP SIAD reaches for all tested buffer sizes and Num_{RTT} configurations an average link utilization of 90% to 95%. While TCP NewReno and TCP Cubic are anyway not able to utilize the link fully if the buffer size is too small, both schemes also only reach a link utilization of about 89% in those scenarios where they actually would be able to utilize the link fully (without any disruption of small traffic peaks). Without short flow cross traffic and a buffer size of one BDP TCP Cubic reaches full utilization because it induces a standing queue in this case. This means the buffer is never fully empty, and those packets in the buffer can potentially compensate for unnecessary window reductions. In contrast, TCP SIAD

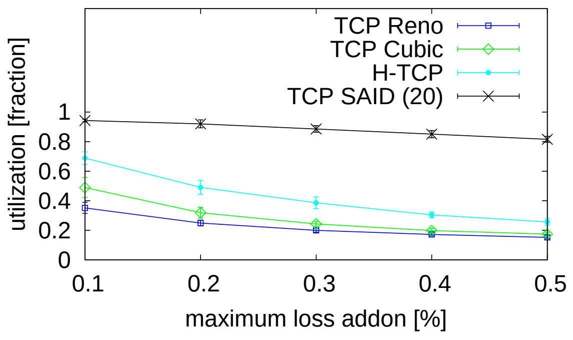
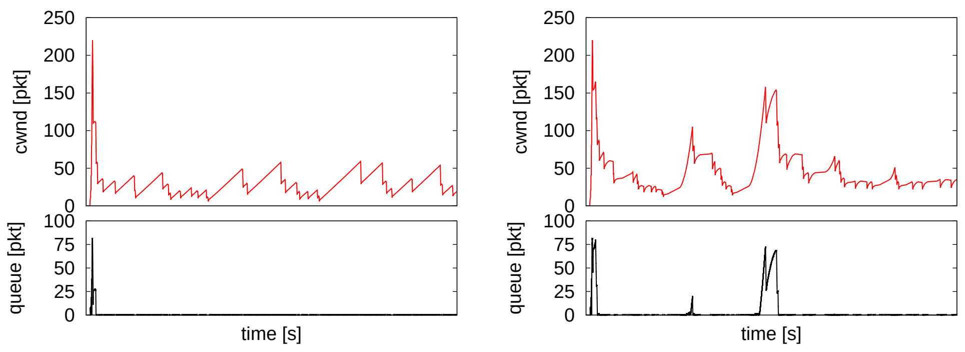


Figure 4.33: Average link utilization of one flow on 10 Mbit/s link and 0.5·BDP of buffering with random loss.

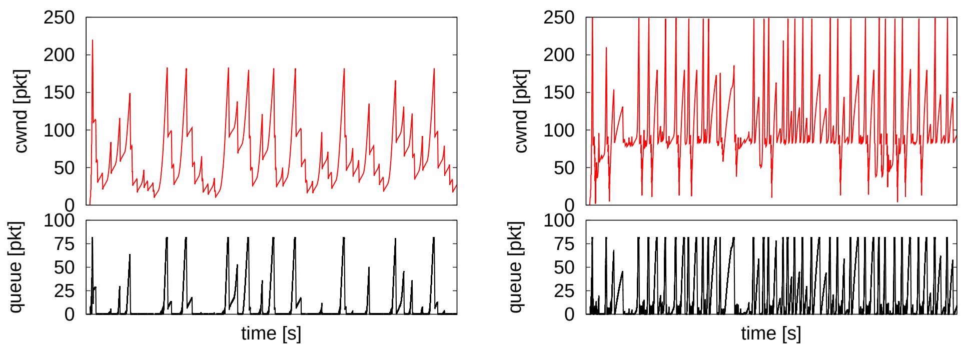
always reduces the buffer such that the link stays utilized; no matter how often congestion is indicated by the network feedback. H-TCP has a similar behavior (for a buffer range of 1·BDP to 0.5·BDP). Even though H-TCP performs better than TCP NewReno or TCP Cubic, it still gets more strongly influenced by short traffic bursts than TCP SIAD, as it stays longer at the low sending rate after a decrease even though the cross traffic already disappeared. In the scenarios with a buffer size of 1·BDP TCP SIAD performs more often Additional Decreases than in case of smaller buffer sizes. These can partly be compensated by Fast Increase, but still lead to a slightly lower utilization of 91% or more. Further, as Fast Increase is often entered TCP SIAD maintains a higher utilization than other schemes but also at the cost of higher loss rates. Both, high permanent utilization and loss, also cause slightly longer transmission times for the short flow traffic but this cannot be avoided if the high link utilization is desired and loss-based congestion control that frequently needs to fill the queue is used. Example congestion window and queue length traces are shown in Appendix C.6.

To further test TCP SIAD's robustness in high loss scenarios, we additionally evaluate a scenario with randomly distributed, artificial losses. E.g., non-congestion related loss could occur on the lower layers when a bit error causes a packet drop. Often, these losses are not randomly distributed but bundled. As TCP congestion control only reacts to congestion once per RTT, a random distribution of loss is even worse. Figure 4.33 gives the average link utilization of a 10 Mbit/s link with 1.0·BDP of buffering where the artificially induced loss rate is between 0.1% and 0.5%. It shows clearly that TCP SIAD handles this case best with a high link utilization in all cases. Also other NUM_{RTT} configurations of TCP SIAD provide similar results.

Figure 4.34 shows sample traces for the schemes in test with 0.1% artificial loss. With traditional both schemes, TCP NewReno and TCP Cubic, the queue is mostly empty as they always decrease by a fixed value even though most of the losses are not caused by congestion in this scenario. TCP NewReno and TCP Cubic can only reach a link utilization of 35.15% and 48.96%. H-TCP handles the situation better and utilizes the link by 68.85%. Only TCP SIAD reaches a high utilization of 94.3% with $Num_{RTT} = 20$ or 97.3% with $Num_{RTT} = 40$. Of course, H-TCP and TCP SIAD also decrease for each loss notification assuming congestion. However, they decrease less as the queue was not fully filled when the congestion notification was generated. Therefore, also the delay is not at its maximum value when the assumed congestion notification is received. H-TCP still decreases at least with a factor of 0.3 and therefore underutilizes the link. TCP SIAD is able to re-allocate the capacity more quickly due to Fast Increase, which



(a) TCP NewReno. (b) TCP Cubic.



(c) H-TCP. (d) TCP SIAD with $Num_{RTT} = 20$.

Figure 4.34: One flow on 10 Mbit/s link and 0.5·BDP of buffering with 0.1% additional random loss.

in return also causes higher oscillations and loss rates. However, in such a situation where non-congestion losses alter the congestion signal there is a trade-off between achieving link utilization by sending aggressively which causes overshoots and inducing low loss rates. TCP SIAD allows to maintain this trade-off by providing configurable aggressiveness.

4.6.2 Influence of the use of AQM schemes

Finally, we investigate the influence of AQM mechanisms implemented in the bottleneck queue on TCP SIAD. Even if today a majority of queues are DropTail, we expect the large-scale introduction of AQM in the Internet to move forward in the course of reducing the end-to-end latency in the Internet to better support latency-sensitive applications. To provide lower latency, two new schemes have recently been proposed, namely CoDel and PIE as introduced in Section 2.3.1. In this section we evaluate the interaction of TCP SIAD with CoDel and PIE as well as RED, the most well-known AQM scheme.

For CoDel and PIE we use the standard parameterization as both schemes are proposed to work with zero configuration. This means CoDel targets a queuing delay value of 5 ms and PIE of 20 ms. That translates for our used scenario on a 10 Mbit/s link to a queue fill length of about 4 packets in case of CoDel and 16 packets in case of PIE. With 100 ms RTT and one BDP of maximum buffering, we except a queue fill level of a about 0.05 for CoDel or 0.2 for PIE.

Table 4.8: RED parameterization.

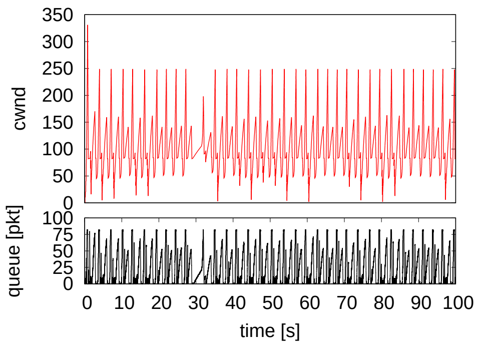
	queue size [BDP]	Min_Thresh	Max_Thresh	maxP	w
RED4	4.0	1/4	3/4	0.1	0.002
RED1	1.0	1/4	3/4	0.1	0.002
NRED4	4.0	1/4	3/4	0.1	1
NRED1	1.0	1/4	3/4	0.1	1

Further, we investigate four different parameterizations of RED, as shown in Table 4.8: the recommended parameterization [42] with two different maximum buffer sizes of 1.0·BDP (RED1) and 4.0·BDP (RED4) as well as the same settings but with a non-smoothed variant ($w = 1$) (NRED1 and NRED4). With a weighting factor w of 1 the feedback signal is not based on the average queue length but on actual current queue and therefore not smoothed but directly feed back. We investigate this non-smoothed variant as such an immediate feedback signal is currently under discussion in the IETF to reduce feedback latency. The smoothing should prevent a congestion reaction if the congestion is not permanent but, e.g., caused by a small traffic burst that disappears again after a short time. As congestion control has a feedback delay of one RTT, no congestion control scheme can react properly to congestion that is present for less than one RTT. Therefore, to avoid link underutilization, the smoothing in RED aims to not signal this kind of (short-term) congestion at all but delay the feedback signal. As we have seen above, TCP SIAD is more resistant to disruptions by short traffic burst and we expect better results with the use of TCP SIAD than for the traditional parameterization. This is because TCP SIAD will anyway only reduce as much as needed to empty the queue and thereby avoid underutilization. Therefore, it is of advantage to signal the feedback immediately without further delays.

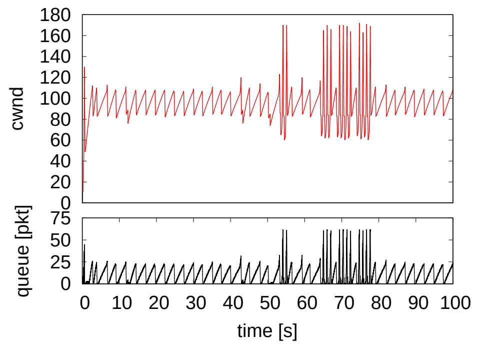
Figure 4.35 shows example traces for TCP SIAD with $NUM_{RTT} = 20$ and the use of the four AQM schemes, with and without smoothing in case of RED, for a maximum buffer size of 1.0·BDP. While TCP SIAD achieves high utilization in all cases as the queue is never empty for a long time, the use of AQM can also induce high oscillations. Only the non-smoothed variant provides a more stable behavior as a congestion situation is resolved faster while in the other cases often congestion is signaled over two subsequent RTTs.

For comparison we run the same simulation as well with TCP NewReno, TCP Cubic, and H-TCP. Note if the maximum buffer size is set to 4·BDP, the lower threshold Min_Thresh is at 1·BDP. In this case, also TCP NewReno should be able to fully utilize the link, in all other cases we expect TCP NewReno to underutilize the link due to the fixed decrease factor of 0.5. Table 4.9a, 4.9c, and 4.9b show the average link utilization, queue fill level, and loss rate of TCP NewReno, TCP Cubic, H-TCP, as well as TCP SIAD for two configuration with a Num_{RTT} value of 20 and 40 on a 10 Mbit/s bottleneck link. As expected TCP SIAD can reach respectively high link utilization of 88%-99% for all scenarios as shown in Table 4.9a. The utilization is always comparable to the utilization reached by the other schemes. While TCP Cubic in the most of the investigated cases provides the best utilization, H-TCP and TCP SIAD achieve the higher utilization in case of CoDel and RED with one BDP buffering, respectively.

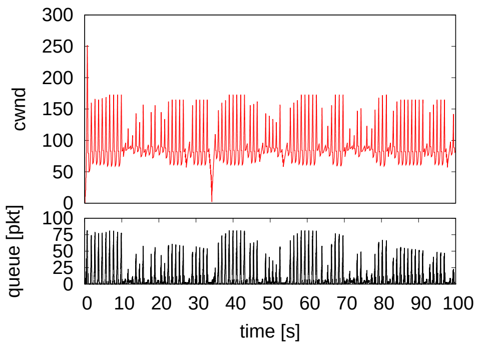
Also as expected TCP SIAD always induces the highest loss rate, as shown in Table 4.9b, due to Fast Increase and consequently high oscillations. However for the non-smoothed RED variant the loss rates are comparable low.



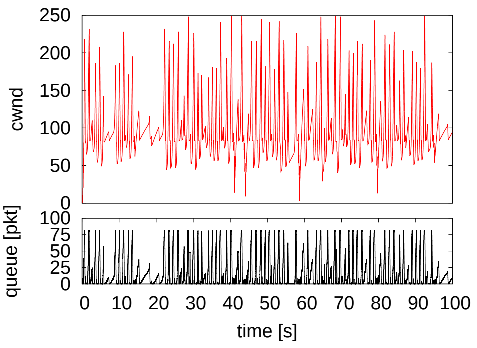
(a) RED with queue size of 1·BDP (RED1).



(b) Non-smoothed RED with queue size of 1·BDP (NRED1).



(c) CoDel.



(d) PIE.

Figure 4.35: One TCP SIAD ($Num_{RTT} = 20$) flow on 10 Mbit/s bottleneck link with different AQM schemes.

While Table 4.9c seems to indicate that the average queue fill level is also very high, TCP SIAD in fact reaches more closely the queue fill level as desired by the respective AQM parameterization. This would be in case of all RED schemes a queue fill level of about 0.25, about 0.2 for PIE (20 ms queuing and 100 ms base delay) and 0.05 for CoDel. For all cases where the link utilization is low, the average queue fill level is consequently smaller as the queue is often empty for a longer time.

4.6.3 Conclusion

We have shown that TCP SIAD is more robust in situations with higher loss rates, e.g., due to short flow and bursty cross traffic or additional losses on the lower layers. In situations with additional, not-congestion related, random loss between 0.2% and 0.5% on a 10 Mbit/s link with 0.5·BDP of buffering TCP SIAD (e.g. with $Num_{RTT} = 20$) can reach a utilization of 84.96 % – 94.47 % due to Adaptive Decrease, while all other schemes including H-TCP cannot utilize the link very well. TCP NewReno only utilizes the link with 15.25 % – 24.93 %, TCP Cubic with 17.46 % – 31.6 %, and H-TCP with 25.58 % – 47.29 %. This shows the ability of TCP SIAD to cope with a wide range of Internet scenarios where unexpected loss occurs.

Further, TCP SIAD also reaches high utilization with the use of AQM on the bottleneck queue but it might induce high oscillations. We considered implementing further heuristics in TCP

Table 4.9: One flow on 10 Mbit/ link with use of AQM.

(a) Average link utilization.

	NewReno	TCP Cubic	H-TCP	TCP SIAD (20)	TCP SIAD (40)
RED4	99.27%	**99.96%**	90.28%	90.31%	91.32%
RED1	89.44%	89.4%	85.28%	88.55%	**92.64%**
NRED4	**100%**	**100%**	97.12%	97.07%	99.75%
NRED1	89.5%	**98.91%**	97.7%	98.2%	96.56%
CoDel	78.75%	90.45%	**94.82%**	91.74%	92.12%
PIE	87.52%	**96.06%**	93.64%	89.28%	92.16%

(b) Average loss rate [%].

	NewReno	TCP Cubic	H-TCP	TCP SIAD (20)	TCP SIAD (40)
RED4	0.006	0.062	0.347	2.246	1.924
RED1	0.015	0.094	0.532	4.23	3.807
NRED4	0.005	0.032	0.168	0.341	0.048
NRED1	0.012	0.035	0.167	0.577	1.114
CoDel	0.016	0.044	0.123	4.106	2.867
PIE	0.014	0.037	0.127	3.689	3.806

(c) Average queue fill [fraction].

	NewReno	TCP Cubic	H-TCP	TCP SIAD (20)	TCP SIAD (40)
RED4	0.1428	0.1362	0.1278	0.2926	0.2672
RED1	0.0783	0.0592	0.14	0.291	0.2721
NRED4	0.1432	0.1840	0.1125	0.1468	0.1451
NRED1	0.0665	0.1036	0.1054	0.1376	0.1392
CoDel	0.0084	0.0118	0.0178	0.106	0.1268
PIE	0.0514	0.0911	0.1012	0.2174	0.2058

SIAD to avoid permanent oscillations which would be especially optimized for this scenario. Note that this again would influence the trade-off between responsiveness and smoothness as introduced in Section 2.5.3. As TCP SIAD, however, works stable with a non-smoothed algorithm, we aim for such an AQM mechanism in the future Internet that avoids any additional feedback delay but leave the optimization to the end-host. Such an AQM scheme would support low latency best and has been proposed in the IETF [27]. The goal is to re-define the semantics of ECN such that one gets a more accurate and more frequent feedback signal and subsequently to adapt the congestion response in the end host accordingly. The result on TCP SIAD presented above supports the request to remove feedback smoothing from the network. Consequently, the combination of such an AQM scheme and the use of TCP SIAD helps to provide better support for low-latency services in the future Internet.

Further, to avoid unnecessary Additional Decreases in situations where congestion feedback is received often and the base RTT is hard to estimate due to noise, TCP SIAD could benefit from an additional signaling that indicates explicitly when the queue is empty. In the context of a re-definition of the semantics of ECN, we plan to further evaluate this approach by, e.g., using the de-facto unused ECN ECT(1) codepoint for this signaling. In general, further experimentation with the use AQM as well as with the use of ECN are needed. As the future semantics of ECN as well as AQM approaches to support low latency services are current research targets, we leave further evaluation on TCP SIAD in this context to future work.

Still we conclude that TCP SIAD shows a general robustness to high loss scenarios as well as to delay variations, shown earlier. Therefore, the scalable and adaptive approach of TCP SIAD provides benefits when used in today's Internet where mainly loss-based congestion control schemes are used (and therefore TCP SIAD need to compete with those schemes). Further, TCP SIAD provides high link utilization even with small buffers and higher congestion feedback rates which is desirable for low latency support in the future Internet.

5 Summary and Conclusion

In this work we proposed and evaluated a new congestion control scheme, called TCP SIAD, motivated by three current research challenges, namely scalability in high-speed networks, support for new emerging low latency services, and the implementation of per-user fairness in the Internet. Scalability is a well-known problem of traditional congestion control schemes where the feedback rate decreases with increasing link bandwidth. This leads to long convergence times in large BDP networks or, even worse, prohibits a full utilization of the available bandwidth resources at all. Further, more and more applications with narrow requirements on low latency are emerging which are not well addressed by existing proposals. Even though congestion control cannot solve the latency problem in today's Internet alone, it is responsible to always provide high link utilization even if buffers are configured small to prevent long queuing delays. While many of the existing approaches aim for TCP-friendliness and therefore limit their design space or introduce additional complexity, we do not require flow fairness. Flow fairness limits the instantaneous sending rate per flow which can lead to unnecessary service degradation in situations where the available resource could be distributed over time such that all services are satisfied. However, the implementation of (long-term) per-user fairness cannot be provided by congestion control (that works on a per flow basis) and therefore must be addressed by additional mechanisms in the Internet as, e.g., proposed by ingress congestion policing.

Based on these challenges, we derived general requirements for congestion control for scalability, adaptivity, capacity sharing, as well as convergence and discussed limitations of current congestion control schemes. Finally, we state concrete design goals that our congestion control scheme aims for to address these requirements: high link utilization independent of the network configuration (even with small buffers), minimization of queuing delay (by avoiding a standing queue), quick capacity allocation (if new bandwidth resources become available), implementation of a fixed feedback rate (to fully solve the scalability problem), and a configuration possibility to influence the aggressiveness (and therefore the capacity sharing). None of the current and in this work discussed schemes can achieve all of these design goals.

Therefore, we proposed a new TCP congestion control scheme. TCP SIAD is still an AIMD scheme, as most of today's proposals, but is especially targeted to cope with small buffers as well as high-speed links by implementing a *Scalable Increase Adaptive Decrease* scheme. Further, we propose three additional algorithms, namely Additional Decrease, Fast Increase and Trend calculation, to address all of our design goals.

AIMD schemes in general implement a linear increase of α packets per RTT and a multiplicative decrease with factor β on congestion notification. Scalable Increase determines the linear increase factor α dynamically for each congestion epoch such that a target congestion window value *incthresh* can be reached in a configurable number Num_{RTT} of RTTs. If the congestion window grows above the threshold *incthresh*, α is further increased exponentially within one congestion epoch to quickly allocate newly available capacity in the Fast Increase phase. If congestion occurs, Adaptive Decrease calculates the decrease factor β based on the current estimate of the queue share such that the queue just empties. This approach avoids a standing queue but at the same time also avoids underutilization of the link if the queue is small. If the queue was not emptied by this first decrease due to de-synchronization with competing flows, Additional Decreases are performed. If the sending rate would thereby be reduced too much, Scalable Increase partially compensates this with a larger increase factor to keep the link utilization high. Further, the target value is calculated based on Trend calculation where the previous maximum congestion window value is considered to estimate a trend over time to achieve convergence.

In simulative evaluation based on the integration of our TCP SIAD Linux implementation, we have shown that TCP SIAD is able to utilize a link independent of the configured bottleneck buffer size and at the same time avoid a standing queue in single- as well as in multi-flow scenarios. While H-TCP implements the same decrease scheme, it restricts its decrease factor to be between 0.3 and 0.5 and therefore still causes a standing queue or underutilizes the link in certain scenarios. Only TCP SIAD and Scalable TCP provide a constant feedback rate independent of the link bandwidth. However, as Scalable TCP is an MIMD scheme with a fixed increase factor it induces high loss and in most scenarios a large standing queue. Further, we demonstrated that TCP SIAD is able to achieve stable capacity sharing that can be impacted by the Num_{RTT} configuration parameter. We have shown that Num_{RTT} can even be configured such that equal capacity sharing with cross traffic that uses a different congestion control or operates based on different network conditions can be achieved. Moreover, if flows have different base RTTs and equal sharing is desired, the feedback frequency can be configured by Num_{ms} in milliseconds instead. We have assessed the convergence properties of TCP SIAD based on rate changes, competing CBR traffic, as well as competing TCP traffic, and have shown that TCP SIAD converges reasonably fast compared to other schemes such as H-TCP and often even much faster than TCP Cubic. Moreover, TCP SIAD provides a much higher resilience to non-congestion losses than all other schemes in test since Adaptive Decrease reduces the sending rate based on its current share of the bottleneck queue. All in all, TCP SIAD reaches the stated design goals as well as more general stated requirements for congestion control and shows high robustness that allows for further testing in the Internet.

Future Work

While we have listed and discussed further degrees of freedom in the design space of TCP SIAD in Section 3.4.1, we would like to focus in our future work on a simplified version of TCP SIAD. In fact, robustness is not trivial to evaluate. We assessed vulnerability to delay estimation errors and resilience to high loss rates for this purpose. However, one key principle for robustness is simplicity. For example using a fixed but configurable increase step instead of Scalable increase would strongly simplify convergence and capacity sharing. Of course, such a scheme would not be fully scalable anymore. However, due to Adaptive Decrease it would only scale with the configured buffer size which we expect to be independent of the BDP in future networks supporting low latency services. To evaluate such a simplified scheme would allow a comparison with TCP SIAD as proposed in this work.

In addition, we aim to experiment with TCP SIAD in combination with ECN or a future explicit signaling that could provide more accurate, immediate, and more fine-grained congestion feedback or even additional information about, e.g., if the queue is fully empty or not, as briefly discussed in 3.4.1. Based on these experiments, we aim to provide input on the currently running standardization process in the IETF.

Another area of research in the environment of congestion control is the start-up behavior. We simply adopted the start-up behavior from Slow Start. However, there are a large number of proposals to improve TCP's start up behavior. As TCP SIAD implements the same increase behavior below the Slow Start threshold and above the Linear Increment threshold *incthresh*, both could potentially be improved when further extending TCP SIAD. Especially interactive communication does have tight requirements to quickly reach a certain rate, e.g., at the beginning of an audio communication. In this respect, evaluating TCP SIAD for the use with (real-time) media transmissions that have quite different requirements as, e.g., currently under evaluation in the RTP Media Congestion Avoidance Techniques (rmcat) working group in the IETF [73] would be an interesting application and potentially lead to further extensions of the SIAD scheme, such as probing capabilities.

We have shown that it is generally desirable for high-speed congestion control schemes to implement a Fast Increase phase (not only at start up) to differentiate steady state behavior from changing network conditions and thereby achieve fast capacity allocation in future high-speed network. Therefore, it can also be evaluated how to integrate Fast Increase into other schemes. In addition, SIAD implements a new approach that provides full scalability in future high-speed networks as well as a configuration possibility that supports congestion policing. Evaluating TCP SIAD within a congestion-policed network is expected to provide more flexible capacity sharing and therefore better QoE for all applications. Moreover, with the implementation of Adaptive Decrease TCP SIAD allows network operators to configure smaller buffers and thereby reduce latency without causing network underutilization which is especially important for emerging interactive and real-time applications. Therefore, we can in summary conclude that these basic principles as introduced by TCP SIAD provide an important basis for an efficient operation in the future Internet.

A Source Code

A.1 tcp_siad.c

Listing A.1: tcp_siad.c

```c
/* Scalable Increase Adaptive Decrease (SIAD) Congestion Control Algorithm
   Author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define OFFSET 1      // in pkts (because of rounding)
#define MIN_CWND 2U   // minimum cwnd
#define NUM_RTT 20    // default number of RTT per congestion epoch
                      // between two congestion events
#define MIN_RTT 2     // minimum number of RTTs for one congestion epoch

static int num_rtt = NUM_RTT;
static int num_ms  = 0;

module_param(num_rtt, int, 0644);
MODULE_PARM_DESC(num_rtt, "desired number of RTTs between two congestion
    events (if resulting time interval is larger than configured number of
    milliseconds)");
module_param(num_ms, int, 0644);
MODULE_PARM_DESC(num_ms, "desired milliseconds between two congestion
    events (if larger than resulting time interval for the configured number
     of RTTs)");

//64 Byte=16*32bit
struct siad {
  int config_num_rtt;      // configured Num_RTT value
                           // by Socket Option TCP_SIAD_NUM_RTT
                           // (must be first variable in siad struct)
  u32 default_num_rtt;     // default Num_RTT value
                           // from module parameter or sysctl
                           // (will be set at connection start)
  u32 default_num_ms;      // default Num_ms value
                           // from module parameter or sysctl
                           // (will be set at connection start)
  u32 curr_num_rtt;        // current calculated Num_RTT
                           // based on minimum of num_rtt and num_ms
```

```
                              // or config_num_rtt or sysctl_tcp_siad_num_rtt

  u32 increase;               // = alpha * curr_num_rtt
                              // (provides sufficient resolution as minimum
                              // increase rate of 1 pkt/congestion epoch needed
                                 )
  u32 prev_max_cwnd;          // estimated maximum cwnd
                              // at previous congestion event
  u32 incthresh;              // Linear Increment threshold
                              // to enter Fast Increase phase
                              // (target value after decrease based on max.
                                 cwnd)

  u32 prior_snd_una;          // ACK number of the previously received ACK

  u32 prev_delay;             // delay value of previous sample
                              // (to filter out single outliers)
  u32 curr_delay;             // filtered current delay value
  u32 min_delay;              // absolute minimum delay
  u32 curr_min_delay;         // minimum delay since last congestion event
  u32 dec_cnt;                // number of additional decreases
                              // (for current congestion epoch)
  u8  min_delay_seen;         // state variable if the minimum delay was seen
                              // after a regular window reduction
  u8  increase_performed;     // state variable if at least one increase
                              // was performed before new decrease
  u16 prev_min_delay1,        // previous min_delay values if
      prev_min_delay2,        // monotonously increasing values
      prev_min_delay3;        // due to measurement errors
};

static void tcp_siad_init(struct sock *sk){
  struct siad *siad = inet_csk_ca(sk);
  struct tcp_sock *tp = tcp_sk(sk);

  siad->config_num_rtt = 0;
  // Set sysctl only at connection start
  if (sysctl_tcp_siad_num_rtt) {
    siad->default_num_rtt = max(MIN_RTT, sysctl_tcp_siad_num_rtt);
  } else {
    siad->default_num_rtt = num_rtt;
  }
  if (sysctl_tcp_siad_num_ms) {
    siad->default_num_ms = sysctl_tcp_siad_num_ms;
  } else {
    siad->default_num_ms = num_ms;
  }
  siad->curr_num_rtt = siad->default_num_rtt;

  siad->increase = tp->snd_cwnd*siad->curr_num_rtt;
  siad->prev_max_cwnd = tp->snd_cwnd;
  siad->incthresh = tp->snd_cwnd;

  siad->prior_snd_una=tp->snd_una;

  siad->curr_delay = 0;
  siad->min_delay = INT_MAX;
  siad->curr_min_delay = INT_MAX;
  siad->prev_delay = INT_MAX;
  siad->dec_cnt = 0;
  siad->min_delay_seen=1;
  siad->increase_performed=0;
  siad->prev_min_delay1=0;
  siad->prev_min_delay2=0;
  siad->prev_min_delay3=0;
}
EXPORT_SYMBOL_GPL(tcp_siad_init);

static void tcp_siad_cwnd_event(struct sock *sk, enum tcp_ca_event event)
{
  struct tcp_sock *tp = tcp_sk(sk);
  struct siad *siad = inet_csk_ca(sk);

  switch (event) {
  case CA_EVENT_COMPLETE_CWR:
    siad->prior_snd_una=tp->snd_una;
    siad->curr_min_delay = INT_MAX;
    siad->dec_cnt = 0;
    siad->min_delay_seen = 0;
    siad->increase_performed=0;
    break;
  default:
    break;
  }
}
EXPORT_SYMBOL_GPL(tcp_siad_cwnd_event);

void tcp_siad_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
  struct tcp_sock *tp = tcp_sk(sk);
  struct siad *siad = inet_csk_ca(sk);

  // Estimate current RTT
  u32 delay;
  if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr) {
    // current measurement sample of rtt based on TSopt
    delay = tcp_time_stamp - tp->rx_opt.rcv_tsecr;
  } else {
    //smoothed RTT based on sampled RTT measurements
    delay = tp->srtt>>3;
  }
  // filter out single outliers
  siad->curr_delay = min(delay, siad->prev_delay);
  siad->prev_delay = delay;

  // minimum delay
  if (siad->min_delay == INT_MAX || delay <= siad->min_delay ) {
    // initialize total min delay or set to smaller value
    siad->min_delay = delay;
    siad->min_delay_seen=1;
    siad->curr_min_delay = delay;
  } else if (delay <= siad->curr_min_delay) {
    // update current minimum
    siad->curr_min_delay = delay;
    if (tp->snd_cwnd > tp->snd_ssthresh+(siad->increase/siad->curr_num_rtt)
       +1) {
      // reset total minimum as same minimum was seen over several RTTs
      siad->min_delay = delay;
      siad->min_delay_seen=1;
    }
  }
  // Do not perform additional decreases in Fast Increase or Slow Start
  if (tp->snd_cwnd > siad->incthresh || tp->snd_cwnd < tp->snd_ssthresh)
    siad->min_delay_seen=1;

  // Estimate unack'ed bytes since last ACK
  u32 bytes_acked = ack-siad->prior_snd_una;
  siad->prior_snd_una = ack;

  // Do not increase or decrease if application limited
  if (!tcp_is_cwnd_limited(sk, in_flight))
    return;

  // Perform (additional) decrease or increase of congestion window
  if (tp->snd_cwnd > tp->snd_ssthresh+(siad->increase/siad->curr_num_rtt)+2
      && siad->min_delay_seen==0 && siad->dec_cnt<(siad->curr_num_rtt-1) )
         {
    // minimum delay not seen in the first RTT -> Additional Decrease
    // (perform at maximum Num_RTT-1 additional decreases)

    // count number of additional decrease
    siad->dec_cnt++;

    // Reset congestion counter at decrease
    tp->snd_cwnd_cnt=0;

    // reduce estimated cwnd from one RTT ago (=ssthresh)
    tp->snd_cwnd = (siad->min_delay * tp->snd_ssthresh / siad->curr_delay);

    if (tp->snd_cwnd > MIN_CWND+OFFSET) {
      // decrease further if large enough

      // 1. decrease by additional offset
      tp->snd_cwnd = tp->snd_cwnd-OFFSET;

      // 2. reduce at least by new alpha (=increase/Num_RTT)
            // or reach to 0 (or MIN_CWND) after Num_RTT-1 reductions

      // recalculate increase and alpha
      // (assuming already another reduction of the new alpha
            // -> siad->curr_num_rtt-siad->dec_cnt-1)
      // minimum increase rate of 1 pkt/RTT
      siad->increase = max(1*siad->curr_num_rtt,
          (siad->incthresh - tp->snd_cwnd) *
          siad->curr_num_rtt / (siad->curr_num_rtt-siad->dec_cnt-1));
      u32 alpha = siad->increase/siad->curr_num_rtt;
      // calculate reduction to reach 0 after Num_RTT-1 reductions
      u32 reduce = tp->snd_cwnd/(siad->curr_num_rtt-siad->dec_cnt);
      if (reduce < alpha) {
        // reduce at least by alpha
        if (alpha+MIN_CWND < tp->snd_cwnd) {
          tp->snd_cwnd -= alpha;
        } else {
          // set to MIN_CWND
          tp->snd_cwnd = MIN_CWND;
          // don't do any further decreases
          siad->min_delay_seen=1;
        }
      } else {
        // reduce by 'reduce' if larger than alpha
        if (reduce+MIN_CWND < tp->snd_cwnd) {
          tp->snd_cwnd -= reduce;
        } else {
          // set to MIN_CWND
          tp->snd_cwnd = MIN_CWND;
          // don't do any further decreases
          siad->min_delay_seen=1;
        }
        // recalculate increase as cwnd was reduced again
        // minimum increase rate of 1 pkt/RTT
        siad->increase = max(1*siad->curr_num_rtt,
            (siad->incthresh - tp->snd_cwnd) *
            siad->curr_num_rtt / (siad->curr_num_rtt-siad->dec_cnt));
      }
    } else {
      // set to MIN_CWND
      tp->snd_cwnd = MIN_CWND;
      // don't do any further decreases
      siad->min_delay_seen=1;
      // recalculate increase
      // minimum increase rate of 1 pkt/RTT
      siad->increase = max(1*siad->curr_num_rtt,
            (siad->incthresh - tp->snd_cwnd) *
            siad->curr_num_rtt / (siad->curr_num_rtt-siad->dec_cnt));
    }

    // reset ssthresh
    tp->snd_ssthresh = tp->snd_cwnd-1;

    // don't do any further decreases
    // as increase rate would need to be larger than doubling per RTT
    if (siad->increase > tp->snd_cwnd*siad->curr_num_rtt) {
      siad->min_delay_seen=1;
    }

  } else {
    // regular increase

    // reset num_rtt during one congestion epoch via socket option
    if (siad->config_num_rtt!=0 &&
        siad->config_num_rtt!=siad->curr_num_rtt) {
      siad->curr_num_rtt = siad->config_num_rtt;
    }

    // compensate for delayed ACK by calculation acked_pkts
    u32 acked_pkts = bytes_acked/tp->mss_cache;
    if (bytes_acked%tp->mss_cache || acked_pkts==0)
      acked_pkts++;
    tp->snd_cwnd_cnt += acked_pkts;

    // same logic as in tcp_cong_avoid_ai()
    // but also adapts increase rate (and therefore includes SS)

    // increase by more than one (N) packets
        // if several packets ack'ed and snd_cwnd_cnt>=N*next
    u32 next = max(1, tp->snd_cwnd*siad->curr_num_rtt/siad->increase);
    if (tp->snd_cwnd_cnt >= next) {
      int n = tp->snd_cwnd_cnt/next;
      if (tp->snd_cwnd < tp->snd_cwnd_clamp) {
        // actual number of increased packets
        int inc = min(acked_pkts, min(n, tp->snd_cwnd_clamp-tp->snd_cwnd));
        tp->snd_cwnd+=inc;
        siad->increase_performed=1;

        // adapt increase rate at thresholds
        // or in Slow Start/Fast Increase
        if (tp->snd_cwnd >= tp->snd_ssthresh &&
            (tp->snd_cwnd-inc) < tp->snd_ssthresh &&
            siad->incthresh > tp->snd_ssthresh)
          //recalculate increase when entering CA from SS
          siad->increase = max(1*siad->curr_num_rtt,
              siad->incthresh - tp->snd_ssthresh);
        else if ((tp->snd_cwnd >= tp->snd_ssthresh &&
            (tp->snd_cwnd-inc)<tp->snd_ssthresh &&
            siad->incthresh <= tp->snd_ssthresh) ||
            (tp->snd_cwnd >= siad->incthresh &&
            (tp->snd_cwnd-inc)<siad->incthresh))
          // reset increase rate to 1 pkt/RTT
          // 1) if we passed ssthresh
          //     but don't have information on incthresh or
          // 2) passed/reached incthresh
          siad->increase = 1*siad->curr_num_rtt;
        else if (tp->snd_cwnd > siad->incthresh &&
            siad->increase < ((tp->snd_cwnd>>1)*siad->curr_num_rtt))
          // Slow Start (below ssthresh)
          // or Fast Increase (above incthresh):
          // double increase rate per RTT
          // but limit maximum increase rate to 1.5*cwnd per RTT
          siad->increase += (inc*siad->curr_num_rtt);
        else if (tp->snd_cwnd < tp->snd_ssthresh)
          // always set alpha to cwnd in Slow Start
          siad->increase = tp->snd_cwnd*siad->curr_num_rtt;
      }
      // decrease counter (not by inc but n)
      tp->snd_cwnd_cnt -= n*next;
    }
    }
}
EXPORT_SYMBOL_GPL(tcp_siad_cong_avoid);

u32 tcp_siad_ssthresh(struct sock *sk) {
  struct siad *siad = inet_csk_ca(sk);
  struct tcp_sock *tp = tcp_sk(sk);

  // Reset congestion counter at decrease
  tp->snd_cwnd_cnt=0;

  // Estimate cwnd when congestion event occurred (about one RTT ago)
  u32 cwnd = tp->snd_cwnd;
  if (siad->increase_performed==1) {
    if (siad->increase >= tp->snd_cwnd*siad->curr_num_rtt ||
        tp->snd_cwnd <= tp->snd_ssthresh) {
      // (simply) halve cwnd
      // if increase is larger than current snd_cwnd or
      // if in Slow Start
      cwnd = tp->snd_cwnd>>1;
    } else if (tp->snd_cwnd > siad->incthresh &&
        siad->increase == (tp->snd_cwnd>>1)*siad->curr_num_rtt) {
      // reduce by 1/3 if in Fast Increase
      // but increase rate is limited to maximum already
      cwnd -= cwnd/3;
    } else if (tp->snd_cwnd >= siad->incthresh &&
        siad->incthresh > tp->snd_ssthresh &&
        siad->increase == 1*siad->curr_num_rtt) {
      // reduce by (old) alpha if Fast Increase has been just entered
      // and therefore alpha is 1
      cwnd -= (siad->incthresh - tp->snd_ssthresh)/siad->curr_num_rtt;
    } else if (tp->snd_cwnd > siad->incthresh) {
      // minus alpha/2 in Fast Increase
      cwnd -= min(tp->snd_cwnd-MIN_CWND,
          (siad->increase/siad->curr_num_rtt)>>1);
    } else {
      // minus alpha (= number of increases during last RTT
      //                 since congestion event occurred)
      cwnd -= min(tp->snd_cwnd-MIN_CWND,
          siad->increase/siad->curr_num_rtt);
    }
  }

  // detect monotonic increasing min delays and reset
  if (siad->min_delay < siad->prev_min_delay1 ||
      siad->min_delay < siad->prev_min_delay2 ||
      siad->min_delay < siad->prev_min_delay3) {
    siad->prev_min_delay1=0;
    siad->prev_min_delay2=0;
    siad->prev_min_delay3=0;
  } else if (siad->min_delay > siad->prev_min_delay1) {
    if (siad->prev_min_delay1 == 0)
      siad->prev_min_delay1 = siad->min_delay;
    else if (siad->prev_min_delay2 == 0)
      siad->prev_min_delay2 = siad->min_delay;
    else if (siad->min_delay > siad->prev_min_delay2) {
      if (siad->prev_min_delay3 == 0)
        siad->prev_min_delay3 = siad->min_delay;
      else if (siad->min_delay > siad->prev_min_delay3) {
        // reset minimum delay and remember as first value
        // (reset other two value to zero)
          siad->min_delay = siad->prev_min_delay1;
        siad->prev_min_delay2=0;
        siad->prev_min_delay3=0;
        }
      }
  }

  // calculate new ssthresh
  u32 ssthresh = cwnd;
  if (siad->min_delay!=INT_MAX && siad->curr_delay!=0) {
    // decrease proportional to delay ratio (see H-TCP)
    ssthresh = (siad->min_delay * cwnd / siad->curr_delay);
    } else {
      // halve if no information
      ssthresh = cwnd>>1;
  }
  if (ssthresh > MIN_CWND+OFFSET) {
    // decrease by additional offset
    ssthresh = ssthresh-OFFSET;
  } else {
    // at least MIN_CWND
    ssthresh = MIN_CWND;
  }

  // set current value for num_rtt
  // based on default values or configuration over socket option
  if (siad->config_num_rtt) {
    // use value of socket option TCP_SIAD_NUM_RTT
    siad->curr_num_rtt = siad->config_num_rtt;
  } else if (siad->default_num_ms && siad->min_delay!=INT_MAX &&
      siad->curr_delay!=0) {
    // calculate a Num_RTT based on current average RTT and num_ms
    // use minimum of default values (either calculated Num_RTT or num_ms)
    u32 tmp = (siad->default_num_ms<<1)/(siad->curr_delay+siad->min_delay);
    siad->curr_num_rtt = max(siad->default_num_rtt, tmp);
  } else {
    // use num_rtt in case no valid RTT measurements are available
    siad->curr_num_rtt = siad->default_num_rtt;
  }

  // calculate increase threshold/target value
  // amplify trend to speed-up convergence (but more oscillation!)
  // trend can be positive or negative
  int trend = cwnd - siad->prev_max_cwnd;
  if (siad->prev_max_cwnd < 2*cwnd)
    // increment threshold at least new cwnd after reduction (=ssthresh)
    siad->incthresh = max(cwnd + trend, ssthresh);
  else
    siad->incthresh = ssthresh;

  // calculate new increase
  // with minimum increase rate of 1 packet per RTT
  siad->increase = max(1*siad->curr_num_rtt, siad->incthresh - ssthresh);

  // remember estimated max value before reduction
  // for next trend calculation
  siad->prev_max_cwnd = cwnd;

  return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_siad_ssthresh);

```

```

u32  tcp_siad_undo_cwnd(struct sock *sk) {
  struct siad *siad = inet_csk_ca(sk);
  struct tcp_sock *tp = tcp_sk(sk);
  u32 cwnd = siad->incthresh;
  siad->incthresh = siad->prev_max_cwnd;
  siad->min_delay_seen = 1;
  return cwnd;
}
EXPORT_SYMBOL_GPL(tcp_siad_undo_cwnd);

static struct tcp_congestion_ops tcp_siad = {
  .init = tcp_siad_init,
  .name = "siad",
  .ssthresh = tcp_siad_ssthresh,
  .cong_avoid = tcp_siad_cong_avoid,
  .cwnd_event = tcp_siad_cwnd_event,
  .undo_cwnd = tcp_siad_undo_cwnd,
};

static int __init tcp_siad_register(void){
  return tcp_register_congestion_control(&tcp_siad);
}

static void __exit tcp_siad_unregister(void){
  tcp_unregister_congestion_control(&tcp_siad);
}

module_init(tcp_siad_register);
module_exit(tcp_siad_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP SIAD");
MODULE_VERSION("1.0");
```

A.2 Patches for sysctl parameters and socket option

Listing A.2: Patch for TCP SIAD incl. sysctl's and socket option

```
diff -paur linux-3.5.7/include/linux/sysctl.h linux-3.5.7_siad-sysctl/include/linux/sysctl.h
--- linux-3.5.7/include/linux/sysctl.h  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/include/linux/sysctl.h  2014-10-21 14:26:13.260651350 +0200
@@ -425,6 +425,8 @@ enum
  NET_TCP_ALLOWED_CONG_CONTROL=123,
  NET_TCP_MAX_SSTHRESH=124,
  NET_TCP_FRTO_RESPONSE=125,
+ NET_TCP_SIAD_NUM_RTT=128,
+ NET_TCP_SIAD_NUM_MS=129,
 };

 enum {
diff -paur linux-3.5.7/include/linux/tcp.h linux-3.5.7_siad-sysctl/include/linux/tcp.h
--- linux-3.5.7/include/linux/tcp.h 2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/include/linux/tcp.h 2014-03-31 18:36:50.000000000 +0200
@@ -110,6 +110,7 @@ enum {
 #define TCP_REPAIR_QUEUE 20
 #define TCP_QUEUE_SEQ    21
 #define TCP_REPAIR_OPTIONS 22
+#define TCP_SIAD_NUM_RTT 23

 struct tcp_repair_opt {
  __u32 opt_code;
diff -paur linux-3.5.7/include/net/tcp.h linux-3.5.7_siad-sysctl/include/net/tcp.h
--- linux-3.5.7/include/net/tcp.h 2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/include/net/tcp.h 2014-10-21 14:27:02.212652513 +0200
@@ -253,6 +253,9 @@ extern int sysctl_tcp_cookie_size;
 extern int sysctl_tcp_thin_linear_timeouts;
 extern int sysctl_tcp_thin_dupack;
 extern int sysctl_tcp_early_retrans;
+extern int sysctl_tcp_siad_num_rtt;
+extern int sysctl_tcp_siad_num_ms;
+

 extern atomic_long_t tcp_memory_allocated;
 extern struct percpu_counter tcp_sockets_allocated;
Only in linux-3.5.7_siad-sysctl/include/net: tcp.h~
diff -paur linux-3.5.7/kernel/sysctl_binary.c linux-3.5.7_siad-sysctl/kernel/sysctl_binary.c
--- linux-3.5.7/kernel/sysctl_binary.c  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/kernel/sysctl_binary.c  2014-10-21 14:27:33.283653252 +0200
@@ -400,6 +400,8 @@ static const struct bin_table bin_net_ip
  /* NET_TCP_AVAIL_CONG_CONTROL "tcp_available_congestion_control" no longer used */
  { CTL_STR,  NET_TCP_ALLOWED_CONG_CONTROL,   "tcp_allowed_congestion_control" },
  { CTL_INT,  NET_TCP_MAX_SSTHRESH,      "tcp_max_ssthresh" },
+ { CTL_INT,  NET_TCP_SIAD_NUM_RTT,      "tcp_siad_num_rtt" },
+ { CTL_INT,  NET_TCP_SIAD_NUM_MS,       "tcp_siad_num_ms" },

  { CTL_INT,  NET_IPV4_ICMP_ECHO_IGNORE_ALL,    "icmp_echo_ignore_all" },
  { CTL_INT,  NET_IPV4_ICMP_ECHO_IGNORE_BROADCASTS, "
     icmp_echo_ignore_broadcasts" },
diff -paur linux-3.5.7/net/ipv4/Kconfig linux-3.5.7_siad-sysctl/net/ipv4/
    Kconfig
--- linux-3.5.7/net/ipv4/Kconfig  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/net/ipv4/Kconfig  2014-10-21 14:28:55.673655210
     +0200
@@ -567,6 +567,19 @@ config TCP_CONG_ILLINOIS
  For further details see:
    http://www.ews.uiuc.edu/~shaoliu/tcpillinois/index.html

+config TCP_CONG_SIAD
+ tristate "TCP SIAD"
+ depends on EXPERIMENTAL
+ default n
+ ---help---
+ TCP SIAD is congestion control scheme that dynmaically adapt the
+ increase step and the decrease factor (to the buffer size based on
+ RTT measurements). Further a new parameter configures the
+ congestion event length and therefore the aggressivness.
+
+ For further details see:
+   http://mirja.kuehlewind.net/siad
+
 choice
  prompt "Default TCP congestion control"
  default DEFAULT_CUBIC
@@ -595,6 +608,9 @@ choice
  config DEFAULT_WESTWOOD
    bool "Westwood" if TCP_CONG_WESTWOOD=y

+ config DEFAULT_SIAD
+   bool "Siad" if TCP_CONG_SIAD=y
+
  config DEFAULT_RENO
    bool "Reno"

@@ -616,6 +632,7 @@ config DEFAULT_TCP_CONG
  default "vegas" if DEFAULT_VEGAS
  default "westwood" if DEFAULT_WESTWOOD
  default "veno" if DEFAULT_VENO
+ default "siad" if DEFAULT_SIAD
  default "reno" if DEFAULT_RENO
  default "cubic"

diff -paur linux-3.5.7/net/ipv4/Makefile linux-3.5.7_siad-sysctl/net/ipv4/
    Makefile
--- linux-3.5.7/net/ipv4/Makefile 2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/net/ipv4/Makefile 2014-10-21 14:09:22.917029849
     +0200
@@ -48,6 +48,7 @@ obj-$(CONFIG_TCP_CONG_SCALABLE) += tcp_s
 obj-$(CONFIG_TCP_CONG_LP) += tcp_lp.o
 obj-$(CONFIG_TCP_CONG_YEAH) += tcp_yeah.o
 obj-$(CONFIG_TCP_CONG_ILLINOIS) += tcp_illinois.o
+obj-$(CONFIG_TCP_CONG_SIAD) += tcp_siad.o
 obj-$(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) += tcp_memcontrol.o
 obj-$(CONFIG_NETLABEL) += cipso_ipv4.o

diff -paur linux-3.5.7/net/ipv4/sysctl_net_ipv4.c linux-3.5.7_siad-sysctl/
    net/ipv4/sysctl_net_ipv4.c
--- linux-3.5.7/net/ipv4/sysctl_net_ipv4.c  2012-10-12 22:48:25.000000000
    +0200
+++ linux-3.5.7_siad-sysctl/net/ipv4/sysctl_net_ipv4.c  2014-10-21
    14:30:06.829656901 +0200
@@ -687,6 +687,20 @@ static struct ctl_table ipv4_table[] = {
     .extra2    = &two,
   },
   {
+    .procname = "tcp_siad_num_rtt",
+    .data    = &sysctl_tcp_siad_num_rtt,
+    .maxlen    = sizeof(int),
+    .mode    = 0644,
+    .proc_handler = proc_dointvec,
+ },
+ {
+    .procname = "tcp_siad_num_ms",
+    .data    = &sysctl_tcp_siad_num_ms,
+    .maxlen    = sizeof(int),
+    .mode    = 0644,
+    .proc_handler = proc_dointvec,
+ },
+ {
     .procname = "udp_mem",
     .data    = &sysctl_udp_mem,
     .maxlen    = sizeof(sysctl_udp_mem),
diff -paur linux-3.5.7/net/ipv4/tcp.c linux-3.5.7_siad-sysctl/net/ipv4/tcp.
    c
--- linux-3.5.7/net/ipv4/tcp.c  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/net/ipv4/tcp.c  2014-10-21 14:42:13.892674181
    +0200
@@ -2638,6 +2638,15 @@ static int do_tcp_setsockopt(struct sock
     else
       icsk->icsk_user_timeout = msecs_to_jiffies(val);
     break;
+ case TCP_SIAD_NUM_RTT:
+    /* Defines the desired distance between to congestion events in number
    of RTTs
+     * if SIAD congestion control is used
+     */
+    if (!strcmp(icsk->icsk_ca_ops->name, "siad") && val > 0) {
+      int *setvalue = (int*) inet_csk_ca(sk);
+      *setvalue = val;
+    }
+    break;
   default:
     err = -ENOPROTOOPT;
     break;
diff -paur linux-3.5.7/net/ipv4/tcp_input.c linux-3.5.7_siad-sysctl/net/
    ipv4/tcp_input.c
```

```
--- linux-3.5.7/net/ipv4/tcp_input.c  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_siad-sysctl/net/ipv4/tcp_input.c  2014-10-21
    14:32:16.329659979 +0200
@@ -101,6 +101,9 @@ int sysctl_tcp_moderate_rcvbuf __read_mo
 int sysctl_tcp_abc __read_mostly;
 int sysctl_tcp_early_retrans __read_mostly = 2;

+int sysctl_tcp_siad_num_rtt __read_mostly = 0;
+int sysctl_tcp_siad_num_ms __read_mostly = 0;
+
 #define FLAG_DATA    0x01 /* Incoming frame contained data.    */
 #define FLAG_WIN_UPDATE    0x02 /* Incoming ACK was a window update. */
 #define FLAG_DATA_ACKED    0x04 /* This ACK acknowledged new data.   */
Only in linux-3.5.7_siad-sysctl/net/ipv4: tcp_siad.c
```

Listing A.3: Sysctl’s for disabling delayed ACKs and Slow Start threshold setting

```
diff -paur linux-3.5.7/include/linux/sysctl.h linux-3.5.7_sysctls/include/
    linux/sysctl.h
--- linux-3.5.7/include/linux/sysctl.h  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_sysctls/include/linux/sysctl.h  2014-10-21
    15:14:06.675719641 +0200
@@ -425,6 +425,8 @@ enum
  NET_TCP_ALLOWED_CONG_CONTROL=123,
  NET_TCP_MAX_SSTHRESH=124,
  NET_TCP_FRTO_RESPONSE=125,
+ NET_TCP_DELAYED_ACKS=126,
+ NET_TCP_INITIAL_SSTHRESH=127,
 };

 enum {
diff -paur linux-3.5.7/include/net/tcp.h linux-3.5.7_sysctls/include/net/
    tcp.h
--- linux-3.5.7/include/net/tcp.h 2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_sysctls/include/net/tcp.h 2014-10-21 15:14:42.758720498
    +0200
@@ -253,6 +253,9 @@ extern int sysctl_tcp_cookie_size;
 extern int sysctl_tcp_thin_linear_timeouts;
 extern int sysctl_tcp_thin_dupack;
 extern int sysctl_tcp_early_retrans;
+extern int sysctl_tcp_delayed_acks;
+extern int sysctl_tcp_initial_ssthresh;
+

 extern atomic_long_t tcp_memory_allocated;
 extern struct percpu_counter tcp_sockets_allocated;
diff -paur linux-3.5.7/kernel/sysctl_binary.c linux-3.5.7_sysctls/kernel/
    sysctl_binary.c
--- linux-3.5.7/kernel/sysctl_binary.c  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_sysctls/kernel/sysctl_binary.c  2014-10-21
    15:15:18.460721347 +0200
@@ -400,6 +400,8 @@ static const struct bin_table bin_net_ip
  /* NET_TCP_AVAIL_CONG_CONTROL "tcp_available_congestion_control" no
     longer used */
  { CTL_STR,  NET_TCP_ALLOWED_CONG_CONTROL,   "
     tcp_allowed_congestion_control" },
  { CTL_INT,  NET_TCP_MAX_SSTHRESH,     "tcp_max_ssthresh" },
+ { CTL_INT,  NET_TCP_DELAYED_ACKS,     "tcp_delayed_acks" },
+ { CTL_INT,  NET_TCP_INITIAL_SSTHRESH,   "tcp_initial_ssthresh" },

  { CTL_INT,  NET_IPV4_ICMP_ECHO_IGNORE_ALL,    "icmp_echo_ignore_all" },
  { CTL_INT,  NET_IPV4_ICMP_ECHO_IGNORE_BROADCASTS, "
     icmp_echo_ignore_broadcasts" },
diff -paur linux-3.5.7/net/ipv4/sysctl_net_ipv4.c linux-3.5.7_sysctls/net/
    ipv4/sysctl_net_ipv4.c
--- linux-3.5.7/net/ipv4/sysctl_net_ipv4.c  2012-10-12 22:48:25.000000000
    +0200
+++ linux-3.5.7_sysctls/net/ipv4/sysctl_net_ipv4.c  2014-10-21
    15:16:32.252723100 +0200
@@ -687,6 +687,20 @@ static struct ctl_table ipv4_table[] = {
    .extra2   = &two,
  },
  {
+   .procname = "tcp_delayed_acks",
+   .data   = &sysctl_tcp_delayed_acks,
+   .maxlen   = sizeof(int),
+   .mode   = 0644,
+   .proc_handler = proc_dointvec,
+ },
+ {
+   .procname = "tcp_initial_ssthresh",
+   .data   = &sysctl_tcp_initial_ssthresh,
+   .maxlen   = sizeof(int),
+   .mode   = 0644,
+   .proc_handler = proc_dointvec,
+ },
+ {
    .procname = "udp_mem",
    .data   = &sysctl_udp_mem,
    .maxlen   = sizeof(sysctl_udp_mem),
Only in linux-3.5.7_sysctls/net/ipv4: sysctl_net_ipv4.c~
diff -paur linux-3.5.7/net/ipv4/tcp_input.c linux-3.5.7_sysctls/net/ipv4/
    tcp_input.c
--- linux-3.5.7/net/ipv4/tcp_input.c  2012-10-12 22:48:25.000000000 +0200
+++ linux-3.5.7_sysctls/net/ipv4/tcp_input.c  2014-10-21 15:18:11.793725466
     +0200
@@ -101,6 +101,9 @@ int sysctl_tcp_moderate_rcvbuf __read_mo
 int sysctl_tcp_abc __read_mostly;
 int sysctl_tcp_early_retrans __read_mostly = 2;

+int sysctl_tcp_delayed_acks __read_mostly = 1;
+int sysctl_tcp_initial_ssthresh __read_mostly = 0;
+
 #define FLAG_DATA    0x01 /* Incoming frame contained data.    */
 #define FLAG_WIN_UPDATE    0x02 /* Incoming ACK was a window update. */
 #define FLAG_DATA_ACKED    0x04 /* This ACK acknowledged new data.   */
@@ -903,7 +906,10 @@ static void tcp_init_metrics(struct sock
    /* ssthresh may have been reduced unnecessarily during.
     * 3WHS. Restore it back to its initial default.
     */
-   tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
+   if (sysctl_tcp_initial_ssthresh > 0)
+       tp->snd_ssthresh = sysctl_tcp_initial_ssthresh;
+   else
+ 		tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
  	}
  	if (dst_metric(dst, RTAX_REORDERING) &&
  	    tp->reordering != dst_metric(dst, RTAX_REORDERING)) {
@@ -5219,7 +5225,10 @@ static void __tcp_ack_snd_check(struct s
  		tcp_send_ack(sk);
  	} else {
  		/* Else, send delayed ack. */
- 		tcp_send_delayed_ack(sk);
+ 		if (sysctl_tcp_delayed_acks)
+ 			tcp_send_delayed_ack(sk);
+ 		else
+ 			tcp_send_ack(sk);
  	}
  }
```

A.3 Variants

Listing A.4: SI_only

```
/* Scalable Increase (SI) Congestion Control Algorithm
   Author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define NUM_RTT 20

struct si {
    u32 curr_num_rtt;

    u32 increase;
    u32 incthres;
};

static void tcp_si_init(struct sock *sk){
    struct si *si = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    if (sysctl_tcp_siad_num_rtt)
        si->curr_num_rtt = sysctl_tcp_siad_num_rtt;
    else
        si->curr_num_rtt = NUM_RTT;

    si->incthres = tp->snd_cwnd;
    si->increase = tp->snd_cwnd*si->curr_num_rtt;
}

void tcp_si_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct si *si = inet_csk_ca(sk);

    if (!tcp_is_cwnd_limited(sk, in_flight))
        return;

    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh)  {
        tcp_slow_start(tp);
    /* In dangerous area, increase slowly. */
    } else {
        tcp_cong_avoid_ai(tp, min(tp->snd_cwnd,
                tp->snd_cwnd*si->curr_num_rtt/si->increase));
    }
}
EXPORT_SYMBOL_GPL(tcp_si_cong_avoid);

u32 tcp_si_ssthresh(struct sock *sk) {
    struct si *si = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    si->incthres = tp->snd_cwnd;
    u32 ssthresh = tp->snd_cwnd>>1;

    si->increase = max (1*si->curr_num_rtt, (si->incthres - ssthresh));

    return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_si_ssthresh);

static struct tcp_congestion_ops tcp_si = {
  .init = tcp_si_init,
  .name = "si",
  .ssthresh = tcp_si_ssthresh,
  .cong_avoid = tcp_si_cong_avoid,
};

static int __init tcp_si_register(void){
  return tcp_register_congestion_control(&tcp_si);
}

static void __exit tcp_si_unregister(void){
  tcp_unregister_congestion_control(&tcp_si);
}

module_init(tcp_si_register);
module_exit(tcp_si_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP SI");
MODULE_VERSION("0.1");
```

Listing A.5: Fixed Increase

```
/* Fixed Increase (fixedi) Congestion Control Algorithm
  author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

struct fixedi {
  u32 increase;
};

static void tcp_fixedi_init(struct sock *sk){
    struct fixedi *fixedi = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    // misuse sysctl for configurable increase rate
    if (sysctl_tcp_siad_num_rtt)
      fixedi->increase = sysctl_tcp_siad_num_rtt;
    else
      fixedi->increase = 1;
}

void tcp_fixedi_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct fixedi *fixedi = inet_csk_ca(sk);

    if (!tcp_is_cwnd_limited(sk, in_flight))
  return;

    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh)  {
        tcp_slow_start(tp);
    /* In dangerous area, increase slowly. */
    } else {
  tcp_cong_avoid_ai(tp, min(tp->snd_cwnd, tp->snd_cwnd/fixedi->increase));
    }
}
EXPORT_SYMBOL_GPL(tcp_fixedi_cong_avoid);

static struct tcp_congestion_ops tcp_fixedi = {
  .init = tcp_fixedi_init,
  .name = "fixedi",
  .cong_avoid = tcp_fixedi_cong_avoid,
};

static int __init tcp_fixedi_register(void){
  return tcp_register_congestion_control(&tcp_fixedi);
}

static void __exit tcp_fixedi_unregister(void){
  tcp_unregister_congestion_control(&tcp_fixedi);
}

module_init(tcp_fixedi_register);
module_exit(tcp_fixedi_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP FIXED INCREASE RATE");
MODULE_VERSION("0.1");
```

Listing A.6: SI_trend

```
/*Scalable Increase with Trend (si_trend) Congestion Control Algorithm
  author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define NUM_RTT 20

struct si_trend {
    u32 curr_num_rtt;

    u32 prev_max_cwnd;
    u32 increase;
    u32 incthresh;
};
```

```

static void tcp_si_trend_init(struct sock *sk) {
    struct si_trend *si_trend = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    if (sysctl_tcp_siad_num_rtt)
      si_trend->curr_num_rtt = sysctl_tcp_siad_num_rtt;
    else
      si_trend->curr_num_rtt = NUM_RTT;

    si_trend->prev_max_cwnd = tp->snd_cwnd;
    si_trend->incthresh = tp->snd_cwnd;
    si_trend->increase = tp->snd_cwnd*si_trend->curr_num_rtt;
}

void tcp_si_trend_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct si_trend *si_trend = inet_csk_ca(sk);

    if (!tcp_is_cwnd_limited(sk, in_flight))
        return;

    // regular increase
    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh)  {
  tcp_slow_start(tp);
    /* In dangerous area, increase slowly. */
    } else {
  tcp_cong_avoid_ai(tp, min(tp->snd_cwnd,
                 tp->snd_cwnd*si_trend->curr_num_rtt/si_trend->increase));
    }
}
EXPORT_SYMBOL_GPL(tcp_si_trend_cong_avoid);

u32 tcp_si_trend_ssthresh(struct sock *sk) {
    struct si_trend *si_trend = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    si_trend->incthresh = tp->snd_cwnd;
    u32 ssthresh = tp->snd_cwnd>>1;

    int trend = ssthresh - si_trend->prev_max_cwnd;
    if (si_trend->prev_max_cwnd < 2*ssthresh)
        // increment threshold at least new cwnd after reduction (=ssthresh)
        si_trend->incthresh = max(ssthresh + trend, ssthresh);
    else
  si_trend->incthresh = ssthresh;

    // minimum increase of 1/NUM_RTT pkt/RTT
    si_trend->increase = max (1*si_trend->curr_num_rtt, (si_trend->incthresh - ssthresh));

    si_trend->prev_max_cwnd = tp->snd_cwnd;

    return ssthresh;
}
```

```
EXPORT_SYMBOL_GPL(tcp_si_trend_ssthresh);

static struct tcp_congestion_ops tcp_si_trend = {
  .init = tcp_si_trend_init,
  .name = "si_trend",
  .ssthresh = tcp_si_trend_ssthresh,
  .cong_avoid = tcp_si_trend_cong_avoid,
};

static int __init tcp_si_trend_register(void){
  return tcp_register_congestion_control(&tcp_si_trend);
}

static void __exit tcp_si_trend_unregister(void){
  tcp_unregister_congestion_control(&tcp_si_trend);
}

module_init(tcp_si_trend_register);
module_exit(tcp_si_trend_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP SI TREND");
MODULE_VERSION("0.1");
```

Listing A.7: AD_only

```
/*Adaptive Decrease (AD) Congestion Control Algorithm
  Author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define OFFSET 0
#define MIN_CWND 2U

struct ad {
  u32 min_delay;
  u32 prev_delay;
  u32 curr_delay;
};

static void tcp_ad_init(struct sock *sk) {
    struct ad *ad = inet_csk_ca(sk);
    ad->min_delay = INT_MAX;
    ad->prev_delay = 0;
    ad->curr_delay = 0;
}

void tcp_ad_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct ad *ad = inet_csk_ca(sk);

    // Estimate current RTT
    u32 delay;
    if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr) {
        // current measurement sample of rtt based on TSopt
        delay = tcp_time_stamp - tp->rx_opt.rcv_tsecr;
    } else {
        //smoothed RTT based on sampled RTT measurements
        delay = tp->srtt>>3;
    }

    // minimum delay
    if (ad->min_delay == INT_MAX || delay <= ad->min_delay )
        // initialize total min delay or set to smaller value
        ad->min_delay = delay;
    ad->curr_delay = min(delay, ad->prev_delay);
    ad->prev_delay = delay;

    if (!tcp_is_cwnd_limited(sk, in_flight))
  return;

    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh) {
  tcp_slow_start(tp);
    /* In dangerous area, increase slowly. */
    } else {
  tcp_cong_avoid_ai(tp, tp->snd_cwnd);
    }
}
EXPORT_SYMBOL_GPL(tcp_ad_cong_avoid);

u32 tcp_ad_ssthresh(struct sock *sk) {

    struct ad *ad = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    // calculate new ssthresh
    u32 cwnd = tp->snd_cwnd;
    if (tp->snd_cwnd <= tp->snd_ssthresh)
         // at least halve after slow start (see V.Jacobsone Paper)
        cwnd = tp->snd_cwnd>>1;

    u32 ssthresh;
    if (ad->min_delay!=INT_MAX && ad->curr_delay!=0)
        // decrease factor proportional to delay ratio (see H-TCP)
        ssthresh = ad->min_delay * cwnd / ad->curr_delay;
    else
  ssthresh = cwnd>>1;
    if (ssthresh > MIN_CWND+OFFSET) {
        ssthresh = ssthresh-OFFSET; // decrease by additional offset
    } else {
         ssthresh = MIN_CWND; // at least MIN_CWND
    }

    return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_ad_ssthresh);

static struct tcp_congestion_ops tcp_ad = {
  .init = tcp_ad_init,
  .name = "ad",
  .ssthresh = tcp_ad_ssthresh,
  .cong_avoid = tcp_ad_cong_avoid,
};

static int __init tcp_ad_register(void) {
  return tcp_register_congestion_control(&tcp_ad);
}

static void __exit tcp_ad_unregister(void) {
  tcp_unregister_congestion_control(&tcp_ad);
}

module_init(tcp_ad_register);
module_exit(tcp_ad_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP AD");
MODULE_VERSION("0.1");
```

Listing A.8: Fixed Decrease

```
/* Fixed Decrease (fixedd) Congestion Control Algorithm
  Author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define MIN_CWND 2U //
#define FIXED_DECREASE 42

u32 tcp_fixedd_ssthresh(struct sock *sk) {
    struct fixedd *fixedd = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    // calculate new ssthresh
    u32 ssthresh;
    if (tp->snd_cwnd <= tp->snd_ssthresh)
        // at least halve after slow start (see V.Jacobsone Paper)
        ssthresh = tp->snd_cwnd>>1;
    else if (tp->snd_cwnd > MIN_CWND+FIXED_DECREASE) {
        // decrease by fixed value (= queue size)
        ssthresh = tp->snd_cwnd-FIXED_DECREASE;
    } else {
        ssthresh = MIN_CWND; // at least MIN_CWND
    }

    return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_fixedd_ssthresh);

static struct tcp_congestion_ops tcp_fixedd = {
  .name = "fixedd",
  .ssthresh = tcp_fixedd_ssthresh,
};

static int __init tcp_fixedd_register(void){
  return tcp_register_congestion_control(&tcp_fixedd);
}

static void __exit tcp_fixedd_unregister(void){
  tcp_unregister_congestion_control(&tcp_fixedd);
}

module_init(tcp_fixedd_register);
module_exit(tcp_fixedd_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP FIXED DECREASE");
MODULE_VERSION("0.1");
```

Listing A.9: AD_addDcrease

```
/*Adaptive Decrease with Additional Decrease (ad_addDecrease) Congestion
    Control Algorithm
  Author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define OFFSET 1
#define MIN_CWND 2U

struct ad_addDecrease {
    u32 incthresh;

    u32 prev_delay;
    u32 curr_delay;
    u32 min_delay;
    u32 curr_min_delay;
    u32 dec_cnt;
    u8  min_delay_seen;
};

static void tcp_ad_addDecrease_init(struct sock *sk){
    struct ad_addDecrease *ad_addDecrease = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    ad_addDecrease->incthresh = tp->snd_cwnd;

    ad_addDecrease->curr_delay = 0;
    ad_addDecrease->min_delay = INT_MAX;
    ad_addDecrease->curr_min_delay = INT_MAX;
    ad_addDecrease->prev_delay = INT_MAX;
    ad_addDecrease->dec_cnt = 0;
    ad_addDecrease->min_delay_seen=1;
}

```

```
static void tcp_ad_addDecrease_cwnd_event(struct sock *sk, enum
  tcp_ca_event event)
{
  struct tcp_sock *tp = tcp_sk(sk);
  struct ad_addDecrease *ad_addDecrease = inet_csk_ca(sk);

  switch (event) {
  case CA_EVENT_COMPLETE_CWR:
   ad_addDecrease->curr_min_delay = INT_MAX;
   ad_addDecrease->dec_cnt = 0;
   ad_addDecrease->min_delay_seen = 0;
   break;
  default:
   break;
  }
}

void tcp_ad_addDecrease_cong_avoid(struct sock *sk, u32 ack, u32 in_flight)
  {
  struct tcp_sock *tp = tcp_sk(sk);
  struct ad_addDecrease *ad_addDecrease = inet_csk_ca(sk);

  // Estimate current RTT
  u32 delay;
  if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr) {
    // current measurement sample of rtt based on TSopt
 delay = tcp_time_stamp - tp->rx_opt.rcv_tsecr;
  } else {
 //smoothed RTT based on sampled RTT measurements
 delay = tp->srtt>>3;
  }
  // filter out single outliers
  ad_addDecrease->curr_delay = min(delay, ad_addDecrease->prev_delay);
  ad_addDecrease->prev_delay = delay;

  // minimum delay
  if (ad_addDecrease->min_delay == INT_MAX || delay <= ad_addDecrease->
    min_delay ) {
    // initialize total min delay or set to smaller value
    ad_addDecrease->min_delay = delay;
    ad_addDecrease->min_delay_seen=1;
    ad_addDecrease->curr_min_delay = delay;
  } else if (delay <= ad_addDecrease->curr_min_delay) {
    // update current minimum
 ad_addDecrease->curr_min_delay = delay;
    if (tp->snd_cwnd > tp->snd_ssthresh+1+1) {
   // reset total minimum as same minimum was seen over several RTTs
   ad_addDecrease->min_delay = delay;
   ad_addDecrease->min_delay_seen=1;
 }
  }
  if (tp->snd_cwnd > ad_addDecrease->incthresh || tp->snd_cwnd < tp->
    snd_ssthresh)
 ad_addDecrease->min_delay_seen=1;

  if (!tcp_is_cwnd_limited(sk, in_flight))
    return;

    // decrease/increase cwnd
    if (tp->snd_cwnd > tp->snd_ssthresh+1+2 && ad_addDecrease->
       min_delay_seen==0 && ad_addDecrease->dec_cnt<(ad_addDecrease->
       incthresh-tp->snd_ssthresh-1) ) {
        // minimum delay not seen in the first two RTTs -> additional
           decrease

  ad_addDecrease->dec_cnt++;

        // reduce estimated cwnd at congestion event one RTT ago (=ssthresh
           )
        // based on RTT measurements
  tp->snd_cwnd = (ad_addDecrease->min_delay * tp->snd_ssthresh /
     ad_addDecrease->curr_delay);

  if (tp->snd_cwnd > MIN_CWND+OFFSET) {
      // decrease further if large enough

      // 1. decrease by additional offset
      tp->snd_cwnd = tp->snd_cwnd-OFFSET;

      // 2. reduce to reach to 0(MIN_CWND) after Num_RTT-1 reductions
      u32 reduce = tp->snd_cwnd/(ad_addDecrease->incthresh-tp->snd_ssthresh
         -ad_addDecrease->dec_cnt);
      if (reduce+MIN_CWND < tp->snd_cwnd) {
          tp->snd_cwnd -= reduce;
      } else {
           // set to MIN_CWND
    tp->snd_cwnd = MIN_CWND;
          ad_addDecrease->min_delay_seen=1; // don't do any further
             decreases
      }
  } else {
       // set to MIN_CWND
       tp->snd_cwnd = MIN_CWND;
       ad_addDecrease->min_delay_seen=1; // don't do any further decreases
        }

        // reset ssthresh
    tp->snd_ssthresh = tp->snd_cwnd-1;
    } else {
        // regular increase
  /* In "safe" area, increase. */
  if (tp->snd_cwnd <= tp->snd_ssthresh)  {
      tcp_slow_start(tp);
  /* In dangerous area, increase slowly. */
  } else {
      tcp_cong_avoid_ai(tp, tp->snd_cwnd);
  }
    }
}
EXPORT_SYMBOL_GPL(tcp_ad_addDecrease_cong_avoid);

u32 tcp_ad_addDecrease_ssthresh(struct sock *sk) {
    struct ad_addDecrease *ad_addDecrease = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    // calculate new ssthresh
    u32 cwnd = tp->snd_cwnd;
    if (tp->snd_cwnd <= tp->snd_ssthresh)
        cwnd = tp->snd_cwnd>>1;

    u32 ssthresh;
    if (ad_addDecrease->min_delay!=INT_MAX && ad_addDecrease->curr_delay
       !=0)
        // decrease factor proportional to delay ratio (see H-TCP)
        ssthresh = ad_addDecrease->min_delay * cwnd / ad_addDecrease->
            curr_delay;
    else
        ssthresh = cwnd>>1;
    if (ssthresh > MIN_CWND+OFFSET) {
        ssthresh = ssthresh-OFFSET; // decrease by additional offset
    } else {
        ssthresh = MIN_CWND; // at least MIN_CWND
    }

    // calculate increase threshold/target value
    ad_addDecrease->incthresh = cwnd;

    return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_ad_addDecrease_ssthresh);

static struct tcp_congestion_ops tcp_ad_addDecrease = {
  .init = tcp_ad_addDecrease_init,
  .name = "ad_addD",
  .ssthresh = tcp_ad_addDecrease_ssthresh,
  .cong_avoid = tcp_ad_addDecrease_cong_avoid,
  .cwnd_event = tcp_ad_addDecrease_cwnd_event,
};

static int __init tcp_ad_addDecrease_register(void){
  return tcp_register_congestion_control(&tcp_ad_addDecrease);
}

static void __exit tcp_ad_addDecrease_unregister(void){
  tcp_unregister_congestion_control(&tcp_ad_addDecrease);
}

module_init(tcp_ad_addDecrease_register);
module_exit(tcp_ad_addDecrease_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP AD ADDITIONAL DECREASE");
MODULE_VERSION("0.1");
```

Listing A.10: SIAD_only

```
/*Scalable Increase Adaptive Decrease (siad_only) Congestion Control
    Algorithm
  Author: Mirja Kühlewind, Uni Stuttgart
*/
```

```

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define OFFSET 1
#define MIN_CWND 2U
#define NUM_RTT 20

struct siad_only {
    u32 curr_num_rtt;

    u32 min_delay;
    u32 prev_delay;
    u32 curr_delay;

    u32 increase;
    u32 incthres;
};

static void tcp_siad_only_init(struct sock *sk){
    struct siad_only *siad_only = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    if (sysctl_tcp_siad_num_rtt)
      siad_only->curr_num_rtt = sysctl_tcp_siad_num_rtt;
    else
      siad_only->curr_num_rtt = NUM_RTT;

    siad_only->incthres = tp->snd_cwnd;
    siad_only->increase = tp->snd_cwnd*siad_only->curr_num_rtt;

    siad_only->min_delay = INT_MAX;
    siad_only->prev_delay = 0;
    siad_only->curr_delay = 0;
}

void tcp_siad_only_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct siad_only *siad_only = inet_csk_ca(sk);

    // Estimate current RTT
    u32 delay;
    if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr) {
        // current measurement sample of rtt based on TSopt
  delay = tcp_time_stamp - tp->rx_opt.rcv_tsecr;
    } else {
  //smoothed RTT based on sampled RTT measurements
  delay = tp->srtt>>3;
    }

    // minimum delay
    if (siad_only->min_delay == INT_MAX || delay <= siad_only->min_delay )
  // initialize total min delay or set to smaller value
  siad_only->min_delay = delay;
    siad_only->curr_delay = min(delay, siad_only->prev_delay);
    siad_only->prev_delay = delay;

    if (!tcp_is_cwnd_limited(sk, in_flight))
  return;

    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh)  {
  tcp_slow_start(tp);
    /* In dangerous area, increase slowly. */
    } else {
  tcp_cong_avoid_ai(tp, min(tp->snd_cwnd,
                tp->snd_cwnd*siad_only->curr_num_rtt/siad_only->increase));
    }
}
EXPORT_SYMBOL_GPL(tcp_siad_only_cong_avoid);

u32 tcp_siad_only_ssthresh(struct sock *sk) {
    struct siad_only *siad_only = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    // calculate new ssthresh
    u32 cwnd = tp->snd_cwnd;
    if (tp->snd_cwnd <= tp->snd_ssthresh)
  cwnd = tp->snd_cwnd>>1;
    else
  cwnd -= min(tp->snd_cwnd-MIN_CWND, siad_only->increase/siad_only->
      curr_num_rtt);

    u32 ssthresh; // = cwnd;
    if (siad_only->min_delay!=INT_MAX && siad_only->curr_delay!=0)
  // decrease factor proportional to delay ratio (see H-TCP)
  ssthresh = siad_only->min_delay * cwnd / siad_only->curr_delay;
    else
  ssthresh = cwnd>>1;
    if (ssthresh > MIN_CWND+OFFSET) {
  ssthresh = ssthresh-OFFSET; // decrease by additional offset
    } else {
  ssthresh = MIN_CWND; // at least MIN_CWND
    }

    // calculate increase threshold/target value
    siad_only->incthres = cwnd;
    // minimum increase of 1/NUM_RTT pkt/RTT
    siad_only->increase = max (1*siad_only->curr_num_rtt, (siad_only->
        incthres - ssthresh));

    return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_siad_only_ssthresh);

static struct tcp_congestion_ops tcp_siad_only = {
  .init = tcp_siad_only_init,
  .name = "siad_only",
  .ssthresh = tcp_siad_only_ssthresh,
  .cong_avoid = tcp_siad_only_cong_avoid,
};

static int __init tcp_siad_only_register(void){
  return tcp_register_congestion_control(&tcp_siad_only);
}

static void __exit tcp_siad_only_unregister(void){
  tcp_unregister_congestion_control(&tcp_siad_only);
}

module_init(tcp_siad_only_register);
module_exit(tcp_siad_only_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP siad_only");
MODULE_VERSION("0.1");
```

Listing A.11: SIAD_trend

```
/* Scalable Increase Adaptive Decrease with Trend (siad_trend) Congestion
    Control Algorithm
  Author: Mirja Kühlewind, Uni Stuttgart
*/

#include <linux/module.h>
#include <net/tcp.h>
#include <linux/types.h>

#define OFFSET 1
#define MIN_CWND 2U
#define NUM_RTT 20

struct siad_trend {
    u32 curr_num_rtt;

    u32 min_delay;
    u32 prev_delay;
    u32 curr_delay;

    u32 increase;
    u32 incthresh;
    u32 prev_max_cwnd;
};

static void tcp_siad_trend_init(struct sock *sk) {
    struct siad_trend *siad_trend = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    if (sysctl_tcp_siad_num_rtt)
      siad_trend->curr_num_rtt = sysctl_tcp_siad_num_rtt;
    else
      siad_trend->curr_num_rtt = NUM_RTT;

    siad_trend->prev_max_cwnd = tp->snd_cwnd;
    siad_trend->incthresh = tp->snd_cwnd;
    siad_trend->increase = tp->snd_cwnd*siad_trend->curr_num_rtt;

    siad_trend->min_delay = INT_MAX;
    siad_trend->prev_delay = 0;
    siad_trend->curr_delay = 0;
}

void tcp_siad_trend_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct siad_trend *siad_trend = inet_csk_ca(sk);

    // Estimate current RTT
    u32 delay;
    if (tp->rx_opt.saw_tstamp && tp->rx_opt.rcv_tsecr) {
  // current measurement sample of rtt based on TSopt
  delay = tcp_time_stamp - tp->rx_opt.rcv_tsecr;
    } else {
  //smoothed RTT based on sampled RTT measurements
  delay = tp->srtt>>3;
    }

    // minimum delay
    if (siad_trend->min_delay == INT_MAX || delay <= siad_trend->min_delay
       )
  siad_trend->min_delay = delay; // initialize total min delay or set to
      smaller value

    siad_trend->curr_delay = min(delay, siad_trend->prev_delay);
    siad_trend->prev_delay = delay;

    if (!tcp_is_cwnd_limited(sk, in_flight))
  return;

    // regular increase
    /* In "safe" area, increase. */
    if (tp->snd_cwnd <= tp->snd_ssthresh)  {
  tcp_slow_start(tp);
    /* In dangerous area, increase slowly. */
    } else {
  tcp_cong_avoid_ai(tp, min(tp->snd_cwnd,
                tp->snd_cwnd*siad_trend->curr_num_rtt/siad_trend->increase)
                    );
    }
}
EXPORT_SYMBOL_GPL(tcp_siad_trend_cong_avoid);

u32 tcp_siad_trend_ssthresh(struct sock *sk) {
    struct siad_trend *siad_trend = inet_csk_ca(sk);
    struct tcp_sock *tp = tcp_sk(sk);

    // calculate new ssthresh
    u32 cwnd = tp->snd_cwnd;
    if (tp->snd_cwnd <= tp->snd_ssthresh)
  cwnd = tp->snd_cwnd>>1; // at least halve after slow start (see V.
      Jacobsone Paper)
    else
  cwnd -= min(tp->snd_cwnd-MIN_CWND, siad_trend->increase/siad_trend->
      curr_num_rtt);

    u32 ssthresh; //= cwnd;
    if (siad_trend->min_delay!=INT_MAX && siad_trend->curr_delay!=0)
```

```
 // decrease factor proportional to delay ratio (see H-TCP)
 ssthresh = siad_trend->min_delay * cwnd / siad_trend->curr_delay;
   else
 ssthresh = cwnd>>1;
   if (ssthresh > MIN_CWND+OFFSET) {
 ssthresh = ssthresh-OFFSET; // decrease by additional offset
   } else {
 ssthresh = MIN_CWND; // at least MIN_CWND
   }

   // calculate increase threshold/target value
   int trend = ssthresh - siad_trend->prev_max_cwnd;
   if (siad_trend->prev_max_cwnd < 2*ssthresh)
 siad_trend->incthresh = max(ssthresh + trend, ssthresh);
   else
 siad_trend->incthresh = ssthresh;

   // minimum increase of 1/NUM_RTT pkt/RTT
   siad_trend->increase = max (1*siad_trend->curr_num_rtt, (siad_trend->
       incthresh - ssthresh));

   siad_trend->prev_max_cwnd = cwnd;

   return ssthresh;
}
EXPORT_SYMBOL_GPL(tcp_siad_trend_ssthresh);

static struct tcp_congestion_ops tcp_siad_trend = {
  .init = tcp_siad_trend_init,
  .name = "siad_trend",
  .ssthresh = tcp_siad_trend_ssthresh,
  .cong_avoid = tcp_siad_trend_cong_avoid,
};

static int __init tcp_siad_trend_register(void){
  return tcp_register_congestion_control(&tcp_siad_trend);
}

static void __exit tcp_siad_trend_unregister(void){
  tcp_unregister_congestion_control(&tcp_siad_trend);
}

module_init(tcp_siad_trend_register);
module_exit(tcp_siad_trend_unregister);

MODULE_AUTHOR("Mirja Kuehlewind");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP SIAD TREND");
MODULE_VERSION("0.1");
```

B Overview of Evaluation Scenarios, Parameters, and Metrics

Table B.1 gives an overview about metrics and requirements and Table B.2 summarizes the network and traffic parameters used in Chapter 4.

Table B.1: Overview of evaluation scenarios and metrics.

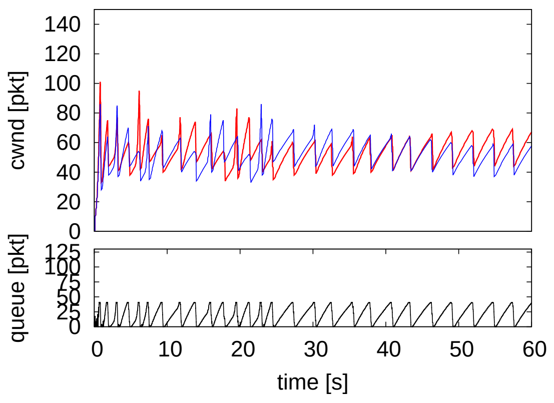
section	requirement	scenario	traffic	metrics
4.3.1	adaptivity scalability	dumbbell	1 greedy flow	avg. link utilization avg. queue fill min. queue fill avg. loss rate avg. loss event distance
4.3.2	robustness	dumbbell	1 greedy flow	avg. link utilization
4.4.1	capacity sharing	dumbbell	2 greedy flows	avg. link utilization avg. queue fill min. queue fill avg. loss rate avg. loss event distance avg. oscillation size
4.4.2	capacity sharing	dumbbell parking lot	2-10 greedy flows	avg. sending rate fairness index
4.4.3	capacity sharing	dumbbell	2 greedy flows (SIAD/non-SIAD)	avg. sending rate
4.5.1	convergence	dumbbell	1 greedy flow CBR cross traffic	convergence time
4.5.2	convergence	dumbbell	2 greedy flows	avg. loss rate avg. convergence time min. convergence time max. convergence time
4.6.1	robustness (loss)	dumbbell	1 greedy flow	avg. link utilization avg. loss rate
4.6.1	robustness (AQM)	dumbbell	1 greedy flow	avg. link utilization avg. loss rate avg. queue fill

Table B.2: Overview of evaluation network parameters and number of greedy flows per scenario.

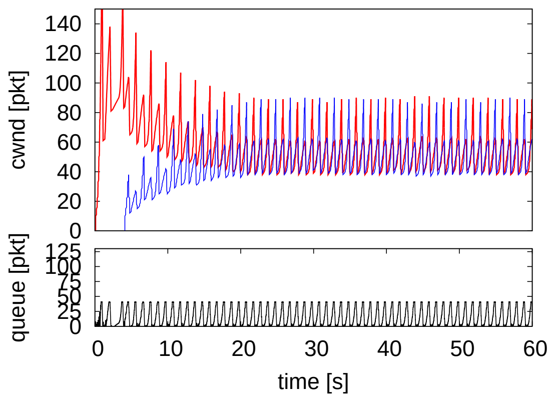
section	bandwidth	buffer size [*BDP]	base RTT	flows	comment
4.3.1	1-100 Mbit/s	0.2 - 2.0	100 ms	1	statistical evaluation
4.3.2 (1)	10 Mbit/s	0.5	100 ms / 40 ms / 140 ms	1	RTT changes at 30s and 60s sim. time
4.3.2 (2)	10-30 Mbit/s	1.0	100 ms	1	varying bandwidth
4.3.2 (3)	10 Mbit/s	0.5	100-103 ms	1	varying RTT
4.3.2 (4)	10 Mbit/s	0.5	100 ms	1	5 Mbit/s CBR cross traffic
4.3.2 (5)	10 Mbit/s	0.5	100 ms	1	TSOpt test
4.4.1 (1)	20 Mbit/s	0.5	100 ms	2	delACK and TSOpt tests
4.4.1 (2)	10-100 Mbit/s	0.2 - 2.0	100 ms	2	statistical evaluation
4.4.2 (1)	20 Mbit/s	0.5	100 ms	2	flows with diff. Num_{RTT} values
4.4.2 (2)	100 Mbit/s	0.1	100 ms	2	flows with diff. Num_{RTT} values
4.4.2 (3)	20 Mbit/s	1.0 (150 ms)	100 ms, 200 ms	2	flows with diff. base RTTs
4.4.2 (4)	10 Mbit/s	1.0	60 ms	4	one flow with multiple bottlenecks
4.4.2 (5)	100 Mbit/s	0.1	100 ms	10	
4.4.3	10 Mbit/s	1.0	100 ms	2	TCP SIAD vs. NewReno/TCP Cubic
4.5.1 (1)	10 Mit/s / 3 Mit/s / 10 Mit/s	0.5 (10 Mit/s)	100 ms	1	bandwidth changes at 30 s and 60 s
4.5.1 (2)	100 Mit/s / 25 Mit/s / 100 Mit/s	0.5 (100 Mit/s)	100 ms	1	bandwidth changes at 30 s and 60,s
4.5.1 (3)	10 Mit/s	1.0	100 ms	1	on/off 5 Mbit/s CBR cross traffic
4.5.1 (4)	100 Mit/s	0.3, 0.5, 1.0, 1.2	100 ms	1	on/off 50 Mbit/s CBR cross traffic
4.5.2	20 Mit/s	0.2 - 2.0	100 ms	2	2. flow starts later in time
4.6.1 (1)	10 Mit/s	0.3, 0.5, 1.0	100 ms	1	short flow cross traffic
4.6.1 (2)	10 Mit/s	0.5	100 ms	1	additional random loss added
4.6.2	10 Mit/s	1.0, 4.0	100 ms	1	diff. AQM schemes

C Further Results

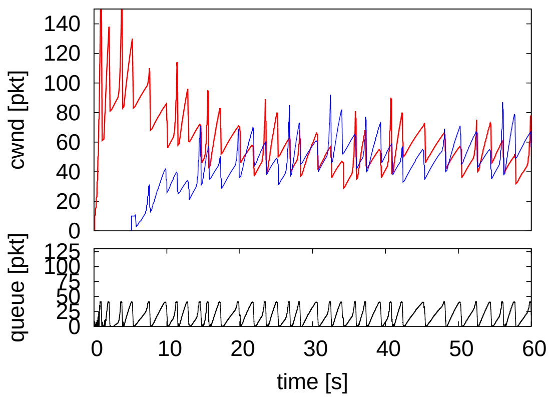
C.1 Individual Algorithm Components: SIAD_trend_fastinc



(a) Both flows start simultaneous.



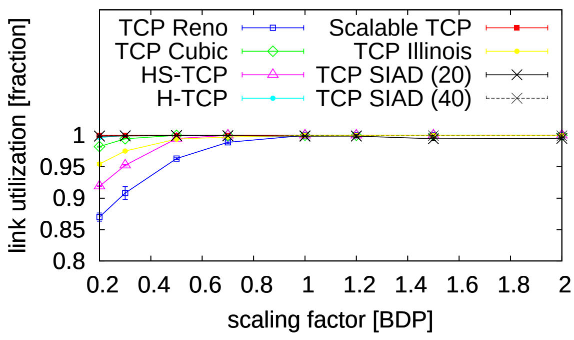
(b) Second flow starts at 4 seconds.



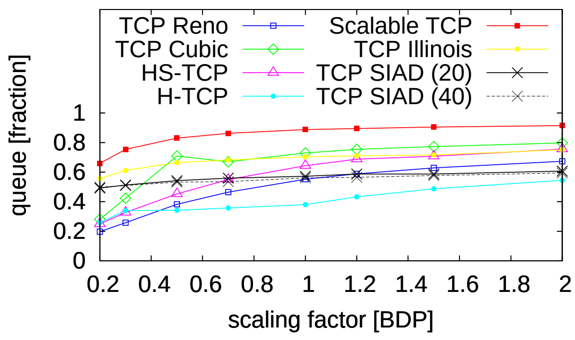
(c) Second flow starts at 5 seconds.

Figure C.1: Two competing SIAD_trend_fastinc flows with different start times.

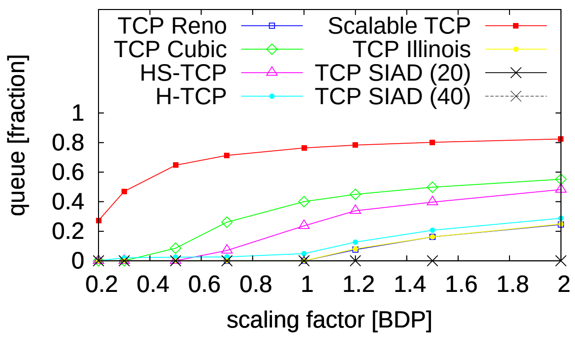
C.2 Single Flow and Two Flows Behavior



(a) Average link utilization.

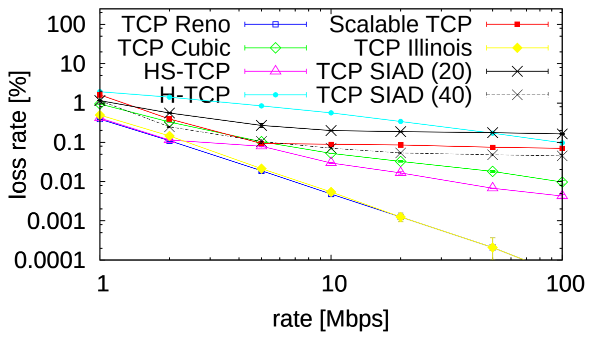


(b) Average queue fill level.

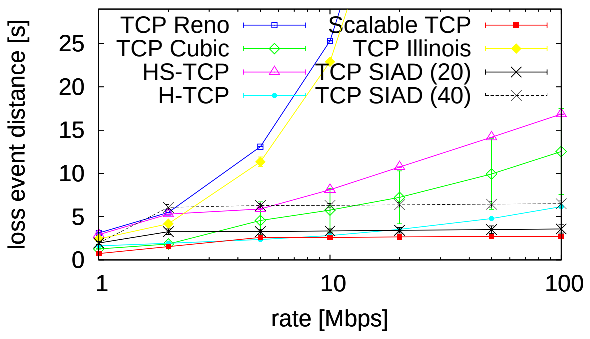


(c) Minimum queue fill level.

Figure C.2: Single flow at 20 Mbit/s.

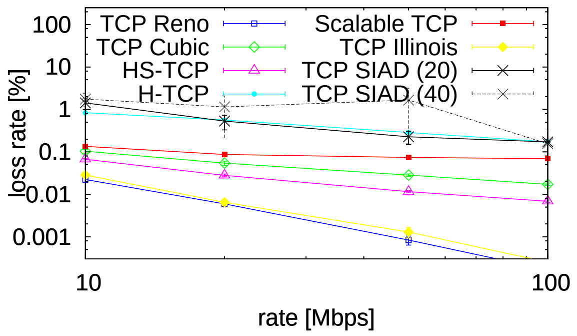


(a) Average loss rate.

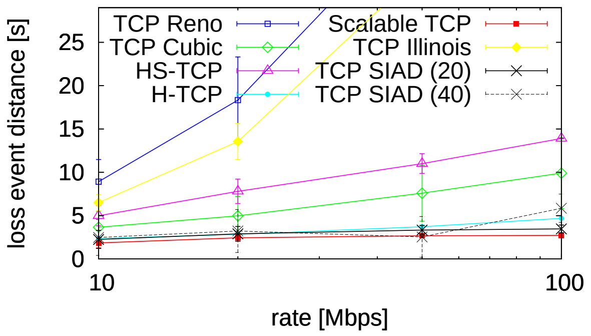


(b) Average loss event distance.

Figure C.3: Single flow with queue size of 1.0*BDP



(a) Average loss rate.



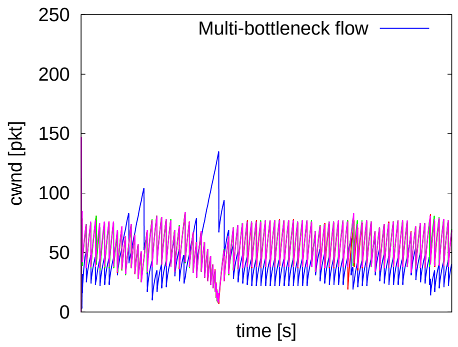
(b) Average loss event distance.

Figure C.4: Two flows with queue size of 1.0*BDP

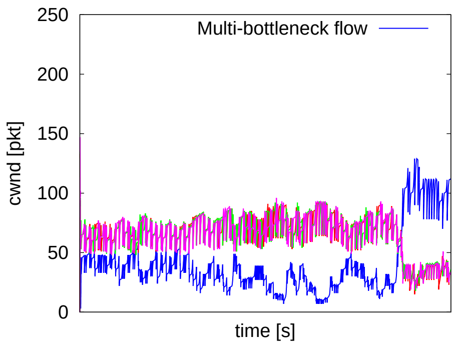
C.2.1 One of four Competing Flows crossing Multiple Bottlenecks

Table C.1: Flow 0 experiencing multiple bottlenecks with each having a link bandwidth of 20 Mbit/s.

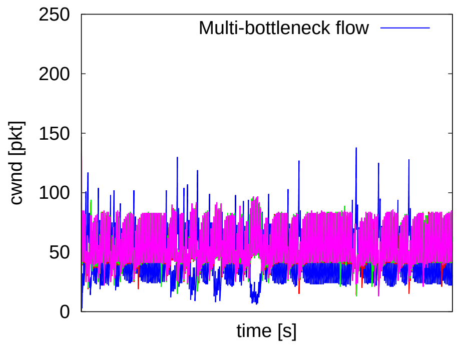
	mean rate 0	mean rate 1	mean rate 2	mean rate 3	fairness
TCP NewReno	3.32 (±0.68)	16.66 (±0.68)	16.67 (±0.68)	16.68 (±0.68)	0.6913
TCP Cubic	2.69 (±0.85)	17.30 (±0.85)	17.31 (±0.85)	17.31 (±0.85)	0.6517
H-TCP	3.86 (±0.66)	16.06 (±0.67)	16.13 (±0.66)	16.14 (±0.66)	0.7264
TCP SIAD (20)	1.43 (±0.32)	18.42 (±0.35)	18.42 (±0.35)	18.43 (±0.33)	0.5771
TCP SIAD (40)	2.63 (±0.47)	17.06 (±0.55)	17.13 (±0.52)	17.2 (±0.5)	0.6502
TCP SIAD (10/40)	6.7 (±0.72)	12.25 (±0.92)	12.68 (±0.81)	12.91 (±0.82)	0.9143
TCP SIAD (5/40)	8.62 (±0.76)	9.64 (±0.95)	10.41 (±0.84)	10.94 (±0.75)	0.9919



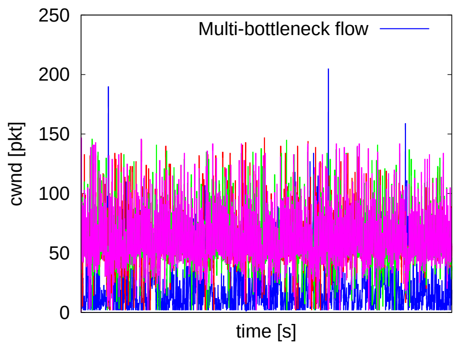
(a) TCP NewReno.



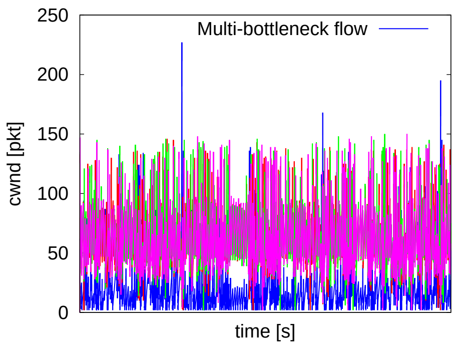
(b) TCP Cubic.



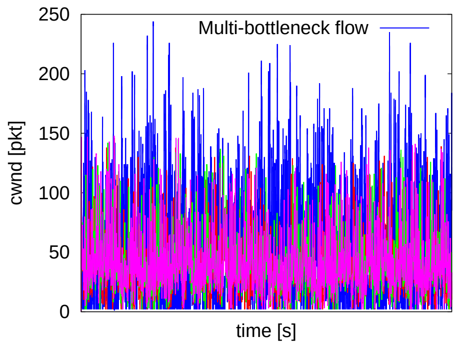
(c) H-TCP.



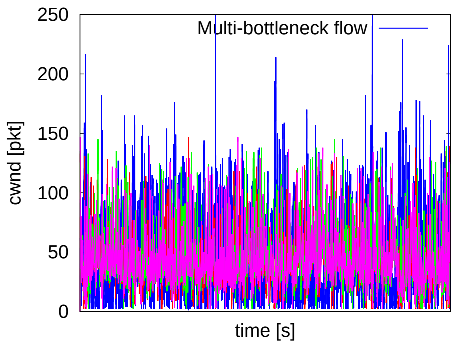
(d) TCP SIAD (Num_{RTT}=20).



(e) TCP SIAD (Num_{RTT}=40).

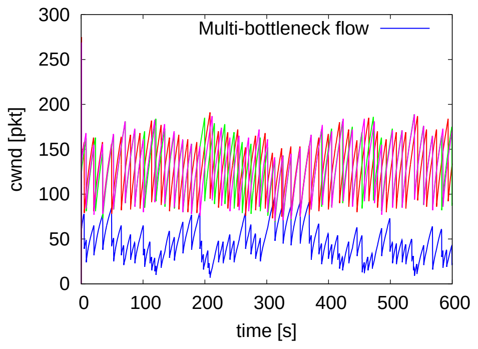


(f) TCP SIAD (Num_{RTT}=40) but multi-bottleneck flow with Num_{RTT}=5.

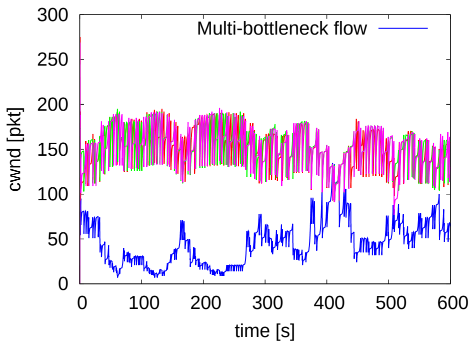


(g) TCP SIAD (Num_{RTT}=40) but multi-bottleneck flow with Num_{RTT}=10.

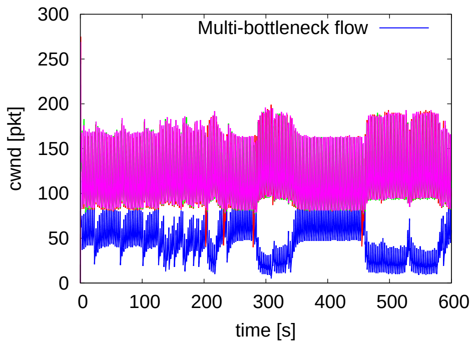
Figure C.5: Four TCP SIAD flows on 10 Mbit/s link and 0.5*BDP buffering with one flow crossing multiple bottlenecks.



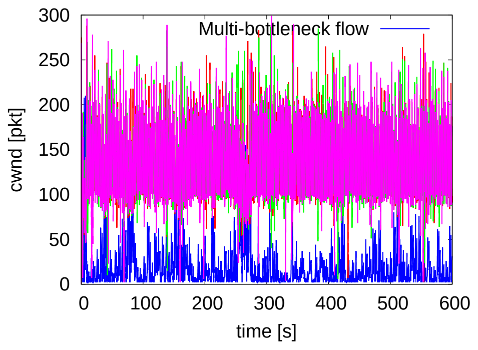
(a) TCP NewReno.



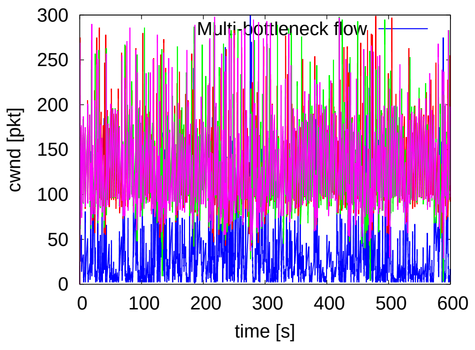
(b) TCP Cubic.



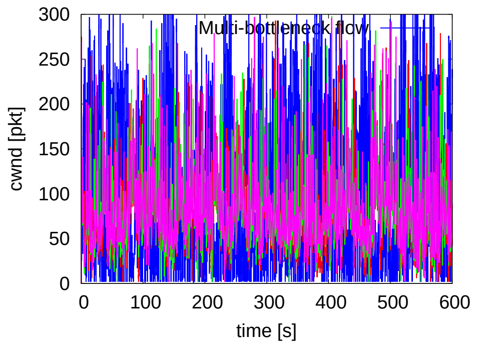
(c) H-TCP.



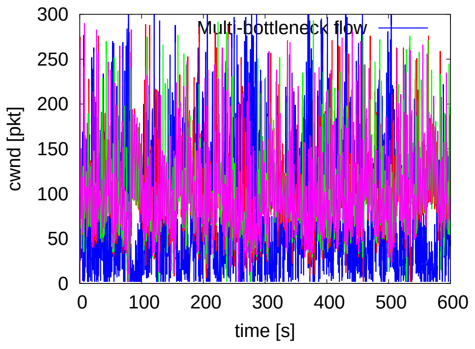
(d) TCP SIAD (Num_{RTT}=20).



(e) TCP SIAD (Num_{RTT}=40).



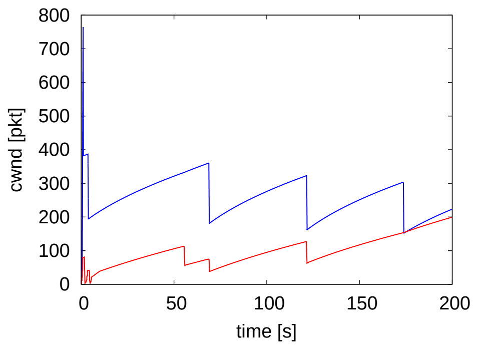
(f) TCP SIAD (Num_{RTT}=40) but multi-bottleneck flow with Num_{RTT}=5.



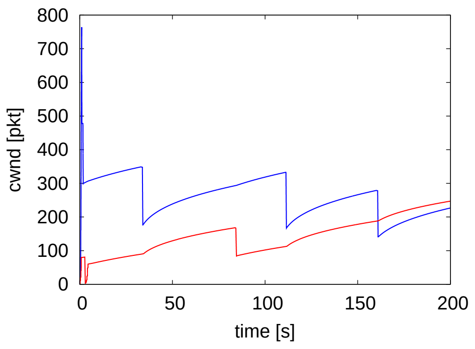
(g) TCP SIAD (Num_{RTT}=40) but multi-bottleneck flow with Num_{RTT}=10.

Figure C.6: Four TCP SIAD flows on 20 Mbit/s link and 0.5*BDP buffering with one flow crossing multiple bottlenecks.

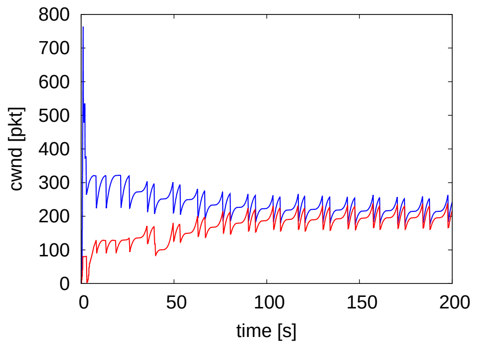
C.2.2 Two Competing Flows with different RTTs of 100 ms and 200 ms



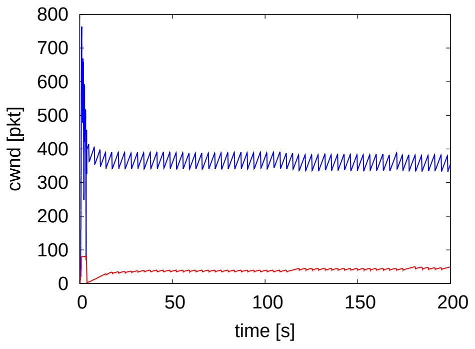
(a) TCP NewReno.



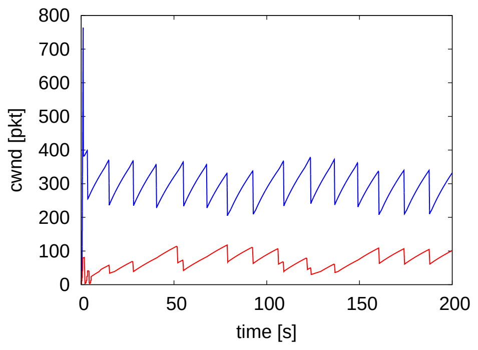
(b) TCP Illinois.



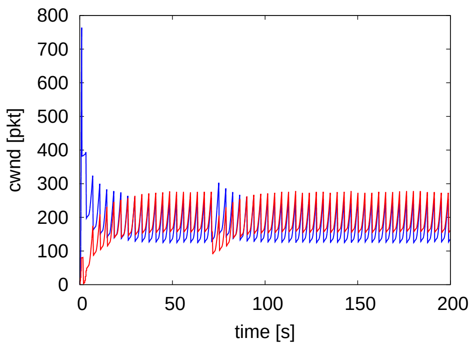
(c) TCP Cubic.



(d) Scalable TCP.

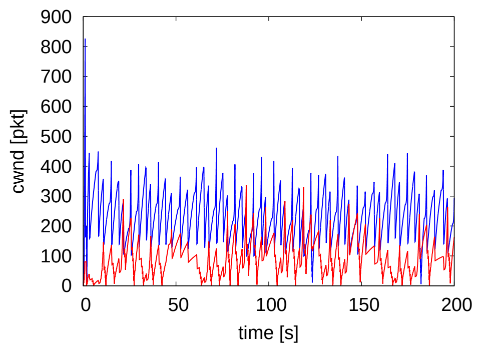


(e) High Speed TCP.

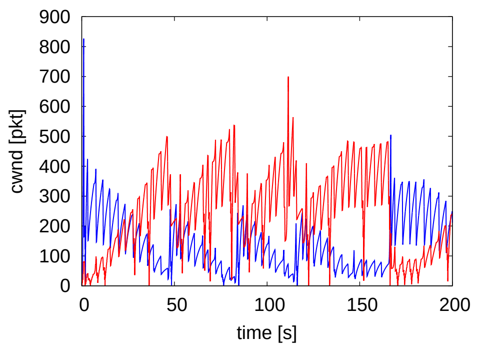


(f) H-TCP.

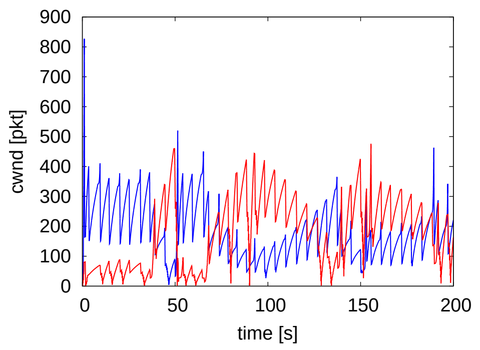
Figure C.7: Two flows on 20 Mbit/s link and 1.0*BDP buffering with different RTTs of 100 ms and 200 ms.



(a) Both flows with $Num_{RTT} = 20$.



(b) One TCP SIAD flow with $Num_{RTT} = 20$ and the other $Num_{RTT} = 10$.



(c) Both TCP SIAD flows with $Num_{MS} = 5000$.

Figure C.8: Two TCP SIAD flows on 20 Mbit/s link and 1.0*BDP buffering with different RTTs of 100 ms and 200 ms.

C.2.3 Statistiacl Evaluation of Scenario with 10 TCP SIAD Flows

Table C.2: Mean rates of 10 TCP SIAD flows with $Num_RTT = 20$.

10 Mbit/s	20 Mbit/s	50 Mbit/s	100 Mbit/s		
0.3*BDP	0.3*BDP	0.3*BDP	0.1*BDP	0.3*BDP	0.5*BDP
0.64 (±0.09)	1.69 (±0.33)	5.36 (±0.91)	10.67 (±1.63)	11.62 (±1.98)	8.76 (±1.61)
0.58 (±0.07)	1.68 (±0.24)	5.28 (±0.93)	9.42 (±1.07)	9.16 (±2.07)	8.44 (±2.4)
0.63 (±0.11)	1.58 (±0.17)	4.92 (±0.99)	9.44 (±1.67)	9.58 (±2.19)	9.95 (±1.9)
0.62 (±0.09)	1.65 (±0.17)	4.7 (±0.8)	10.08 (±1.22)	9.69 (±1.72)	10.13 (±2.49)
0.63 (±0.1)	1.93 (±0.18)	4.55 (±0.77)	9.33 (±1.01)	9.8 (±2.68)	10.03 (±1.37)
0.59 (±0.1)	1.77 (±0.24)	5.25 (±0.77)	10.01 (±1.62)	9.17 (±2.12)	9.93 (±2.23)
0.56 (±0.07)	1.63 (±0.12)	4.24 (±0.54)	9.45 (±1.17)	8.32 (±1.84)	10.79 (±1.9)
0.59 (±0.08)	1.84 (±0.2)	5.19 (±0.82)	10.02 (±1.71)	10.07 (±1.69)	8.3 (±1.61)
0.56 (±0.07)	1.65 (±0.2)	4.73 (±0.89)	9.32 (±1.43)	9.97 (±2.04)	11.69 (±2.82)
0.62 (±0.13)	1.55 (±0.15)	4.86 (±0.64)	11.04 (±1.9)	11.53 (±2.61)	10.45 (±2.18)

Table C.3: Average queue fill level (q), average link utilization (u), and average loss rate (l) of 10 competing TCP SIAD flows with $Num_RTT = 20$.

	10 Mbit/s	20 Mbit/s	50 Mbit/s	100 Mbit/s		
	0.3*BDP	0.3*BDP	0.3*BDP	0.1*BDP	0.3*BDP	0.5*BDP
q	0.64	0.51	0.44	0.46	0.41	0.42
u	99.29%	98.73%	99.24%	99.55%	99.58%	99.76%
l	7.081%	4.119%	2.797%	1.816%	2.374%	2.987%

Table C.4: Mean loss distance and standard deviation [s] of 10 competing TCP SIAD flows with $Num_RTT = 20$ and 0.3*BDP of buffering.

	10 Mbit/s	20 Mbit/s	50 Mbit/s	100 Mbit/s
total	0.41727 (±0.064093)	0.55132 (±0.11328)	0.72141 (±0.16704)	0.82051 (±0.20832)
flow 1	0.6623 (±0.22516)	0.81382 (±0.36869)	0.9617 (±0.45443)	0.9827 (±0.35799)
flow 2	0.65291 (±0.21955)	0.78287 (±0.3345)	0.94958 (±0.43428)	1.0084 (±0.41703)
flow 3	0.65079 (±0.21726)	0.8014 (±0.37064)	0.97274 (±0.46445)	1.0131 (±0.41105)
flow 4	0.65501 (±0.23017)	0.79808 (±0.37861)	0.9401 (±0.40626)	1.0026 (±0.40563)
flow 5	0.65512 (±0.23426)	0.75252 (±0.30741)	0.98071 (±0.48287)	0.99453 (±0.4027)
flow 6	0.65809 (±0.22775)	0.78227 (±0.34877)	0.96753 (±0.44723)	1.0085 (±0.39865)
flow 7	0.64306 (±0.20709)	0.79674 (±0.34767)	0.97567 (±0.44896)	1.0034 (±0.4026)
flow 8	0.6617 (±0.22544)	0.77469 (±0.31996)	0.97727 (±0.4685)	1.0005 (±0.40353)
flow 9	0.65954 (±0.21593)	0.78981 (±0.35083)	0.95275 (±0.41815)	1.0182 (±0.43484)
flow 10	0.65132 (±0.22405)	0.78562 (±0.33306)	0.98208 (±0.45528)	0.97049 (±0.37646)

C.3 Capacity Sharing with Non-SIAD Flows

Table C.5: TCP NewReno cross traffic with delayed ACKs.

Rate	Q size	Num_{RTT}	SIAD mean rate	NewReno mean rate
10 Mbit/s	0.5*BDP	10	8.05 Mbit/s	1.75 Mbit/s
		20	7.11 Mbit/s	2.82 Mbit/s
		30	6.52 Mbit/s	3.43 Mbit/s
		40	6.89 Mbit/s	3.05 Mbit/s
	1.0*BDP	10	8.25 Mbit/s	1.53 Mbit/s
		20	7.47 Mbit/s	2.46 Mbit/s
		30	6.99 Mbit/s	3.00 Mbit/s
		40	6.53 Mbit/s	3.47 Mbit/s
20 Mbit/s	0.5*BDP	10	17.99 Mbit/s	1.71 Mbit/s
		20	17.46 Mbit/s	2.49 Mbit/s
		30	16.45 Mbit/s	3.51 Mbit/s
		40	14.81 Mbit/s	5.14 Mbit/s
	1.0*BDP	10	18.37 Mbit/s	1.34 Mbit/s
		20	17.69 Mbit/s	2.18 Mbit/s
		30	16.88 Mbit/s	3.08 Mbit/s
		40	16.71 Mbit/s	3.29 Mbit/s

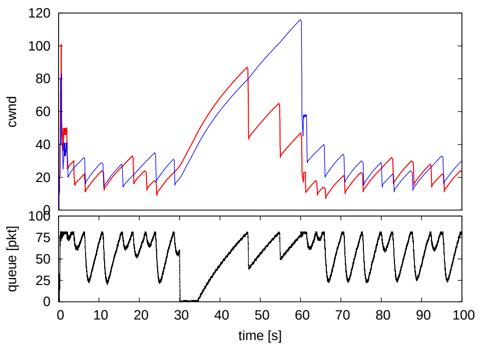
Table C.6: TCP NewReno cross traffic without delayed ACKs.

Rate	Q size	Num_{RTT}	SIAD mean rate	NewReno mean rate
10 Mbit/s	0.5*BDP	10	8.31 Mbit/s	1.63 Mbit/s
		20	6.91 Mbit/s	3.01 Mbit/s
		30	6.12 Mbit/s	3.79 Mbit/s
		40	6.11 Mbit/s	3.79 Mbit/s
		50	6.12 Mbit/s	3.78 Mbit/s
	1.0*BDP	10	8.38 Mbit/s	1.45 Mbit/s
		20	7.35 Mbit/s	2.63 Mbit/s
		30	6.22 Mbit/s	3.78 Mbit/s
		40	5.06 Mbit/s	4.93 Mbit/s
		50	4.74 Mbit/s	5.26 Mbit/s
20 Mbit/s	0.5*BDP	10	18.31 Mbit/s	1.65 Mbit/s
		20	16.82 Mbit/s	3.12 Mbit/s
		30	15.35 Mbit/s	4.57 Mbit/s
		40	13.87 Mbit/s	6.00 Mbit/s
		50	12.34 Mbit/s	7.49 Mbit/s
	1.0*BDP	10	18.29 Mbit/s	1.48 Mbit/s
		20	17.32 Mbit/s	2.64 Mbit/s
		30	16.22 Mbit/s	3.78 Mbit/s
		40	15.08 Mbit/s	4.92 Mbit/s
		50	13.81 Mbit/s	6.19 Mbit/s

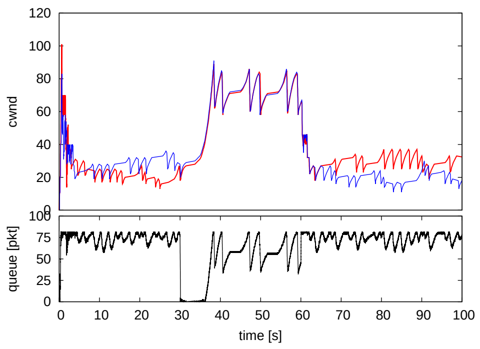
Table C.7: TCP Cubic cross traffic with delayed ACKs.

Rate	Q size	Num_{RTT}	SIAD mean rate	Cubic mean rate
10 Mbit/s	0.5*BDP	10	6.2 Mbit/s	3.69 Mbit/s
		20	5.27 Mbit/s	4.73 Mbit/s
		30	4.88 Mbit/s	5.12 Mbit/s
		40	4.67 Mbit/s	5.33 Mbit/s
	1.0*BDP	10	6.44 Mbit/s	3.43 Mbit/s
		20	4.85 Mbit/s	5.15 Mbit/s
		30	4.98 Mbit/s	5.02 Mbit/s
		40	3.77 Mbit/s	6.23 Mbit/s
20 Mbit/s	0.5*BDP	10	16.48 Mbit/s	3.2 Mbit/s
		20	14.02 Mbit/s	5.98 Mbit/s
		30	11.2 Mbit/s	8.8 Mbit/s
		40	11.74 Mbit/s	8.26 Mbit/s
	1.0*BDP	10	16.55 Mbit/s	3.06 Mbit/s
		20	10.61 Mbit/s	9.3 Mbit/s
		30	10.79 Mbit/s	9.21 Mbit/s
		40	7.63 Mbit/s	12.37 Mbit/s

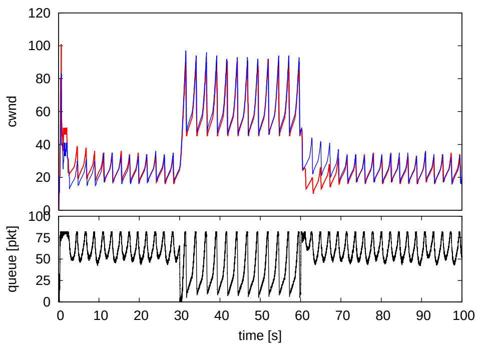
C.4 Two Flow Traces and Loss Rates with Starting CBR Traffic



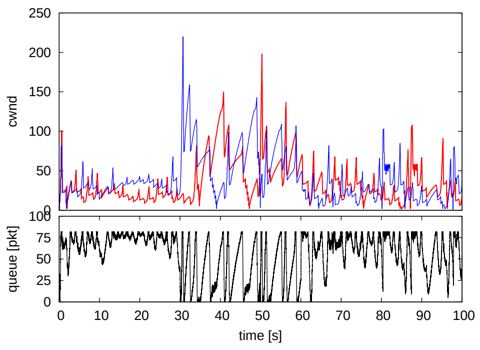
(a) TCP NewReno.



(b) TCP Cubic.

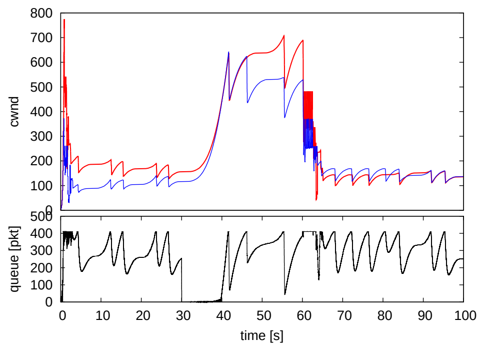


(c) H-TCP.

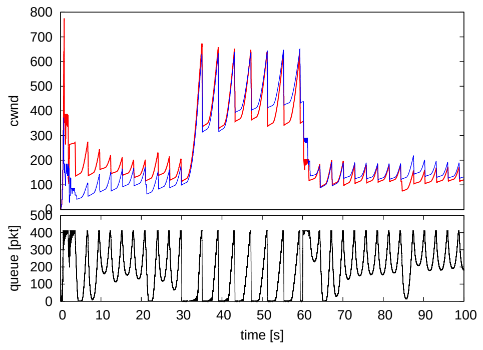


(d) TCP SIAD flow with Num_{RTT} of 20.

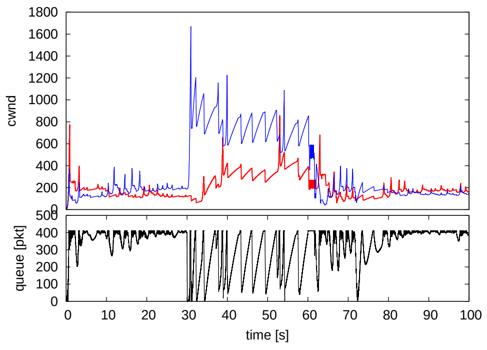
Figure C.9: Two flows at link rate 10 Mbit/s with 7 Mbit/s CBR traffic; off at 30 s on at 60 s.



(a) TCP Cubic.



(b) H-TCP.



(c) TCP SIAD with Num_{RTT} of 20.

Figure C.10: Two flows at link rate 100,Mbit/s with 75 Mbit/s CBR traffic; off at 30 s, on at 60 s.

Table C.8: Loss rate when CBR traffic starts on 10 Mbit/s link.

	*BDP	1 s	2 s	3 s	4 s	5 s
TCP NewReno	0.3	28.76%	19.36%	13.49%	11.37%	09.95%
	0.5	15.43%	8.35%	5.62%	4.26%	3.52%
	1.0	32.46%	24.02%	18.23%	13.96%	11.17%
	1.2	27.62%	20.81%	14.32%	10.88%	8.8%
TCP Cubic	0.3	28.71%	22.49%	16.89%	13.42%	10.82%
	0.5	36.43%	27.22%	20.84%	16.46%	13.73%
	1.0	35.21%	26.84%	21.3%	16.69%	14.06%
	1.2	33.98%	30.23%	23.95%	19.61%	16.39%
H-TCP	0.3	26.12%	26.02%	19.69%	16.29%	14.1%
	0.5	33.2%	27.52%	21.22%	17.64%	14.41%
	1.0	09.96%	5.61%	4.18%	3.13%	3.46%
	1.2	13.96%	8.79%	5.80%	4.95%	4.42%
TCP SIAD (20)	0.3	33.38%	30.86%	24.71%	20.78%	17.4%
	0.5	34.86%	25.58%	20.38%	16.49%	13.45%
	1.0	19.08%	11.73%	15.32%	16.52%	15.41%
	1.2	8.41%	10.64%	7.59%	5.64%	4.69%
TCP SIAD (40)	0.3	33.29%	27.07%	21.53%	17.55%	14.72%
	0.5	34.98%	25.26%	20.79%	16.75%	13.71%
	1.0	27.96%	19.53%	14.73%	14.15%	11.66%
	1.2	25.92%	20.85%	17.35%	14.3%	11.45%

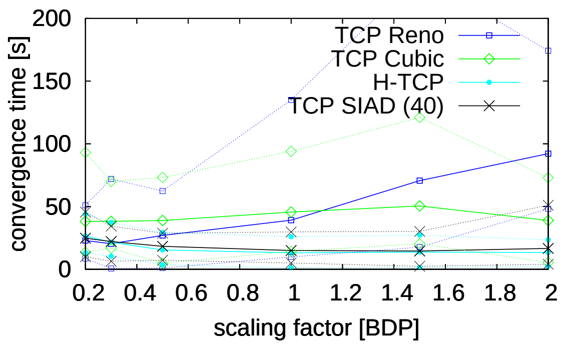
Table C.9: Number of losses when CBR traffic starts on 10 Mbit/s link.

	*BDP	1 s	2 s	3 s	4 s	5 s
TCP NewReno	0.3	32	41	41	43	43
	0.5	30	30	30	30	31
	1.0	58	73	73	73	73
	1.2	39	54	54	54	54
TCP Cubic	0.3	35	52	52	52	52
	0.5	109	132	133	133	133
	1.0	53	72	77	77	77
	1.2	29	57	60	61	61
H-TCP	0.3	67	105	105	109	109
	0.5	62	102	103	103	105
	1.0	13	13	14	14	18
	1.2	24	25	25	28	29
TCP SIAD (20)	0.3	43	83	87	87	87
	0.5	94	109	116	116	116
	1.0	40	44	59	80	88
	1.2	8	18	18	18	19
TCP SIAD (40)	0.3	65	106	110	110	112
	0.5	111	130	144	144	144
	1.0	43	50	51	53	53
	1.2	35	57	72	74	74

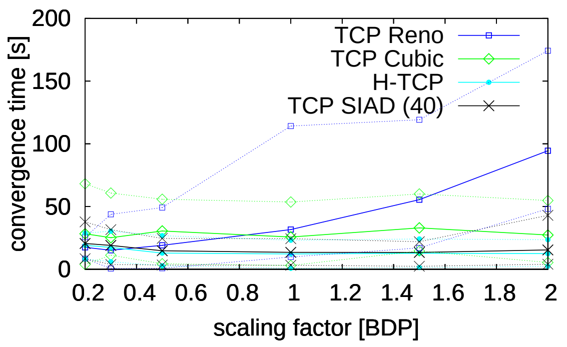
Table C.10: Number of losses when CBR traffic starts on 100 Mbit/s link.

	*BDP	1 s	2 s	3 s	4 s	5 s
TCP NewReno	0.3	21	21	21	21	21
	0.5	28	28	28	28	28
	1.0	21	38	38	38	38
	1.2	0	20	20	20	20
TCP Cubic	0.3	560	1154	1261	1261	1261
	0.5	1040	1314	1314	1314	1314
	1.0	446	713	821	877	877
	1.2	311	613	860	860	860
H-TCP	0.3	0	0	140	146	146
	0.5	670	960	990	990	990
	1.0	216	438	438	438	438
	1.2	182	285	298	298	298
TCP SIAD (20)	0.3	579	1257	1355	1391	1391
	0.5	1226	1414	1419	1476	1476
	1.0	481	641	665	697	778
	1.2	694	1120	1120	1198	1198
TCP SIAD (40)	0.3	691	1137	1198	1229	1229
	0.5	546	736	738	751	812
	1.0	13	13	13	75	75
	1.2	517	905	905	986	986

C.5 Evaluation of Convergence Time

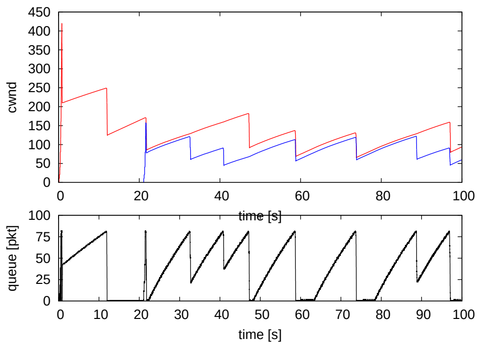


(a) 95% Average, min, and max convergence time ($Num_{RTT} = 40$).

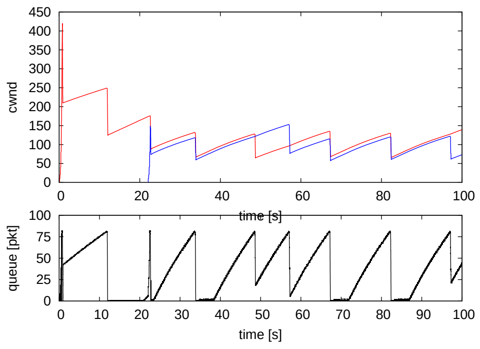


(b) 80% Average, min, and max convergence time ($Num_{RTT} = 40$).

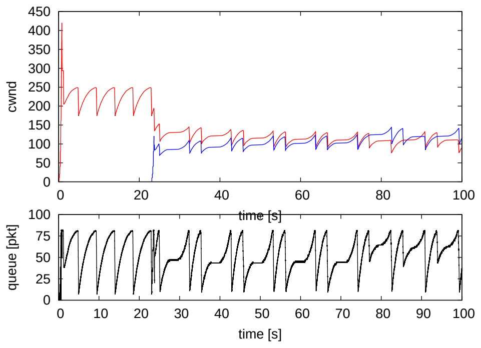
Figure C.11: Average, minimum, and maximum convergence time of two flows with different start times on 20 Mbit/s link.



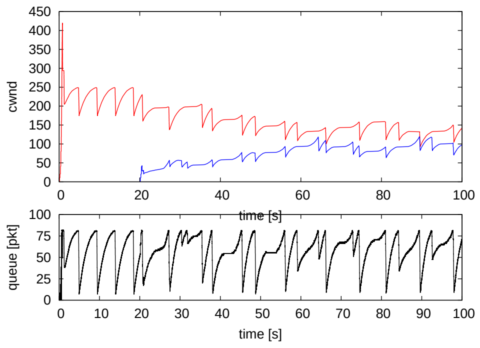
(a) TCP Reno.



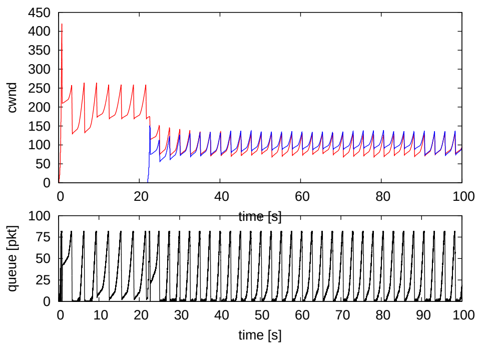
(b) TCP Reno



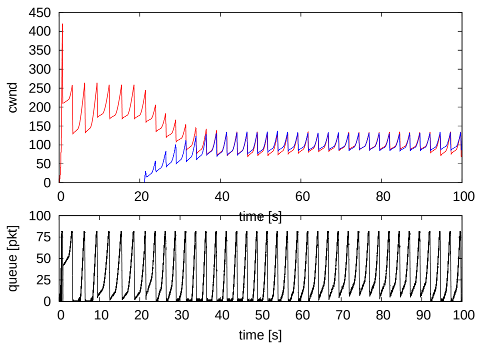
(c) TCP Cubic.



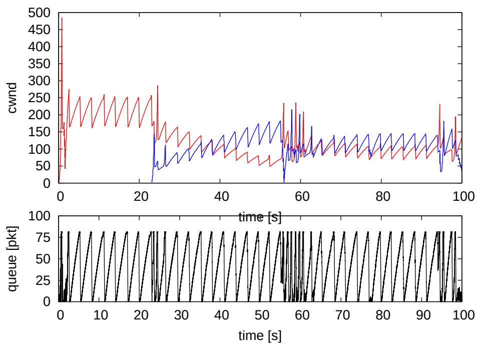
(d) TCP Cubic



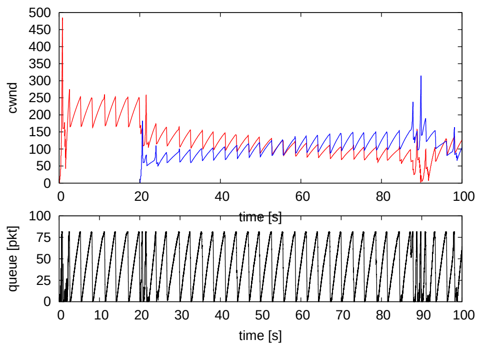
(e) H-TCP.



(f) H-TCP.

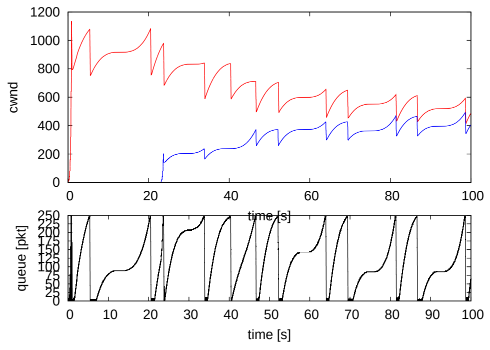


(g) TCP SIAD.

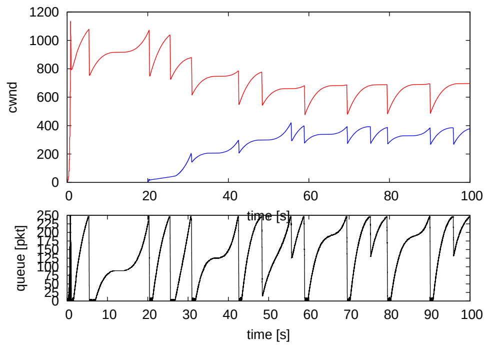


(h) TCP SIAD

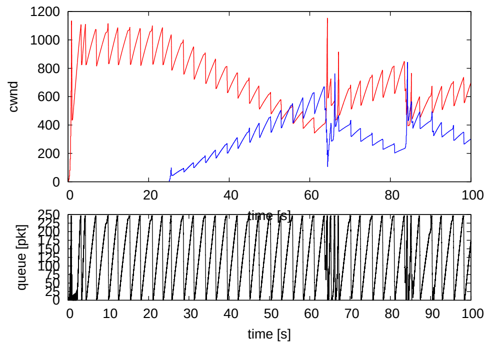
Figure C.12: Example traces of two converging flows on 20 Mbit/s link and 0.5*BDP buffering.



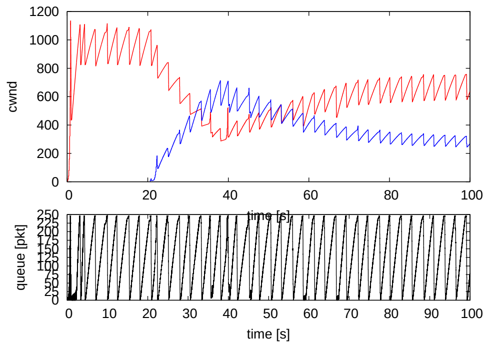
(a) TCP Cubic.



(b) TCP Cubic

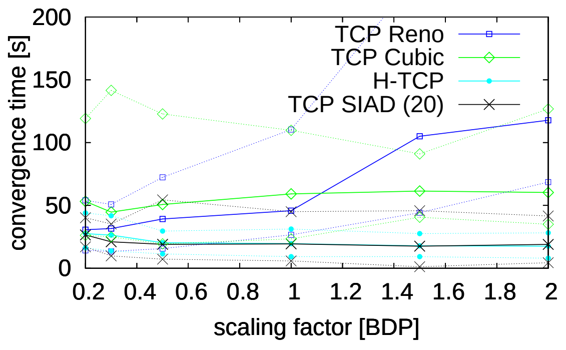


(c) TCP SIAD.

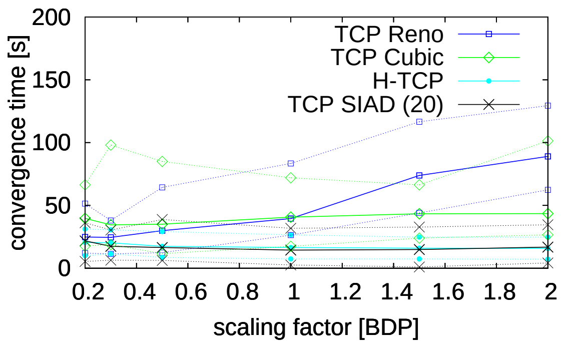


(d) TCP SIAD

Figure C.13: Example traces of two converging flows on 100 Mbit/s link and 0.3*BDP buffering.



(a) Convergence time to reach 95% of equal sharing rate without Slow Start.



(b) Convergence time to reach 80% of equal sharing rate without Slow Start.

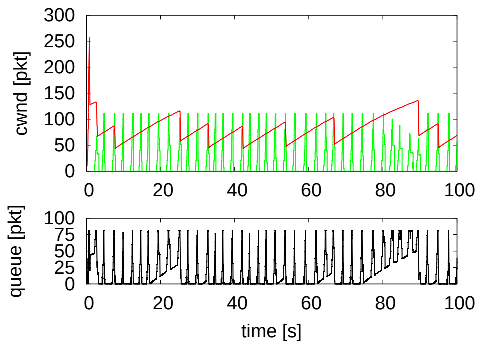
Figure C.14: Average, minimum, and maximum convergence time of two flows on 20 Mbit/s link with different start times.

Note, in case of reaching 95% of the equal share rate one TCP Cubic simulation run did not convergence within 1200 s simulation time. This simulation run was not considered in the evaluation results presented above.

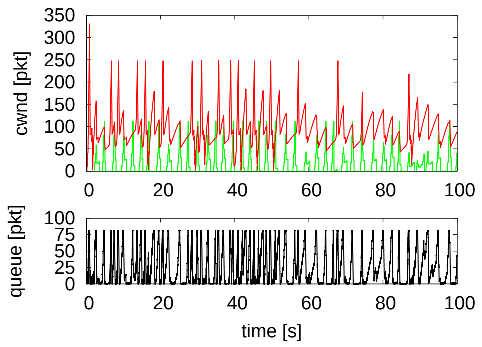
C.6 Short Flow Cross Traffic

Table C.11: Link utilization and loss rate with one flow and short flow cross traffic using TCP Cubic on 10 Mbit/s link.

	buffer size [*BDP]	utilization	loss rate
TCP NewReno	0.3	42.02%	3.65%
	0.5	55.09%	2.177%
	1.0	94.53%	0.51%
TCP Cubic	0.3	77.18%	2.91%
	0.5	89.69%	1.425%
	1.0	99.19%	0.798%
H-TCP	0.3	91.69%	1.241%
	0.5	93.93%	1.157%
	1.0	95.4%	1.364%
TCP SIAD (10)	0.3	93.13%	2.816%
	0.5	93.64%	3.194%
	1.0	95.11%	2.894%
TCP SIAD (20)	0.3	92.43%	1.964%
	0.5	91.61%	1.707%
	1.0	93.47%	2.218%
TCP SIAD (40)	0.3	91.55%	2.476%
	0.5	90.87%	3.056%
	1.0	94.19%	2.264%

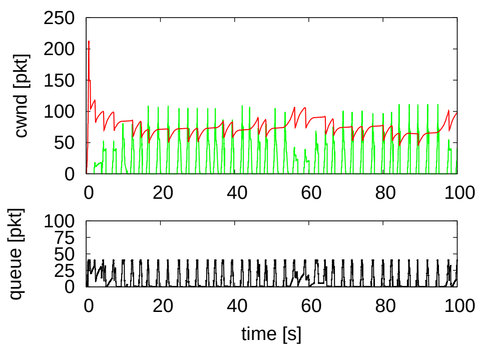


(a) TCP Reno.

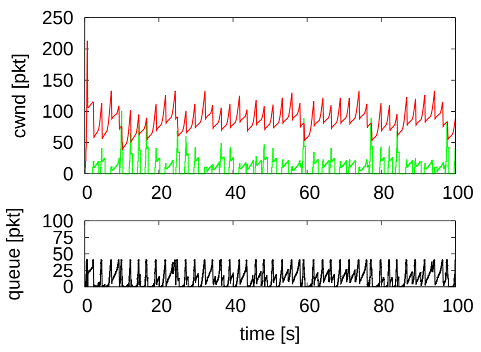


(b) TCP SIAD ($Num_{RTT} = 20$)

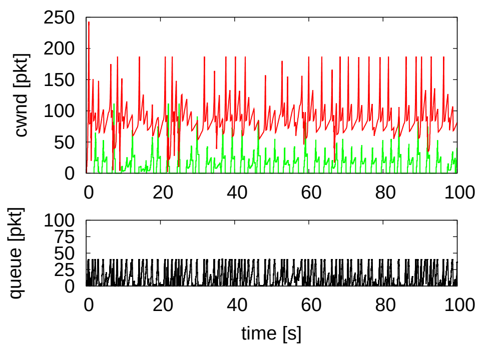
Figure C.15: One long-living flow with short flow cross traffic on 10 Mbit/s link and 1.0*BDP buffering.



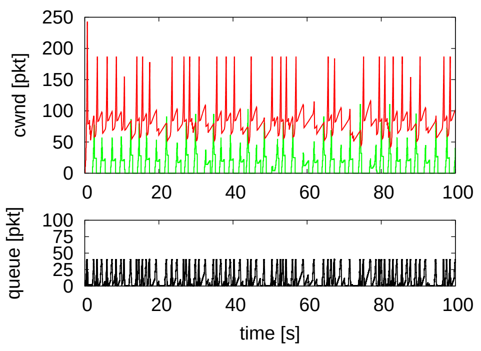
(a) TCP Cubic.



(b) H-TCP.



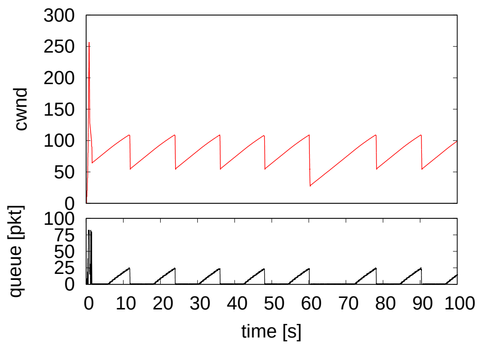
(c) TCP SIAD ($Num_{RTT} = 10$).



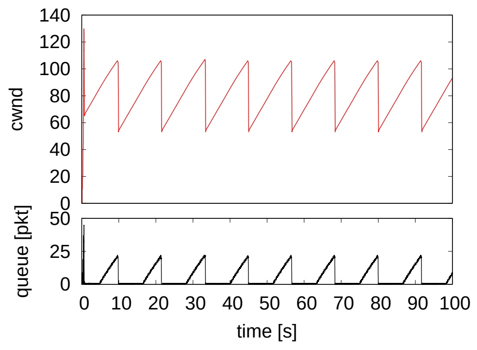
(d) TCP SIAD ($Num_{RTT} = 40$).

Figure C.16: One long-living flow with short flow cross traffic on 10 Mbit/s link and 0.5*BDP buffering.

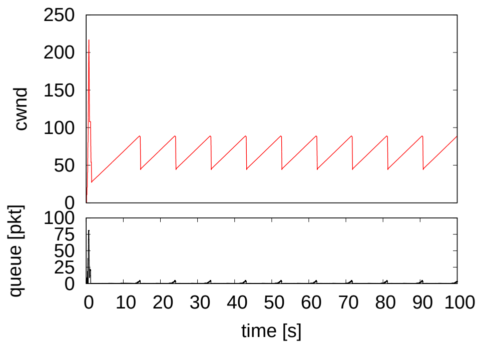
C.6.1 Example Traces with the Use of Different AQM schemes



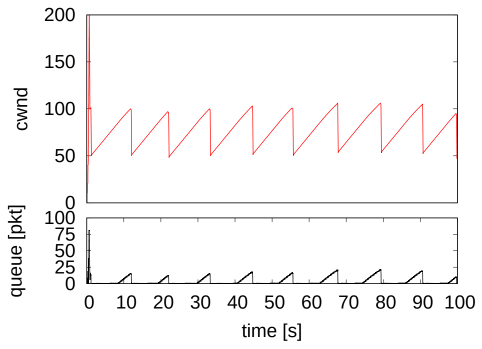
(a) RED with queue size of 1.0*BDP (RED1).



(b) Non-smoothed RED with queue size of 1.0*BDP (NRED1).

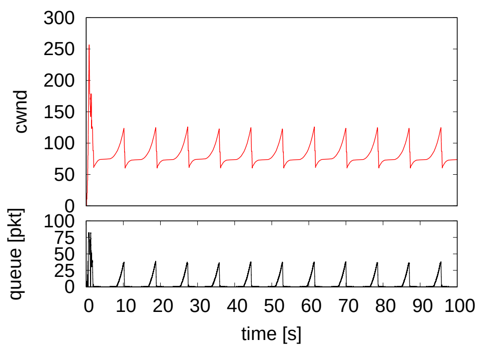


(c) CoDel.

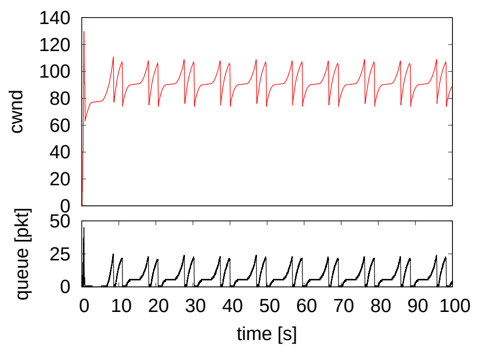


(d) PIE.

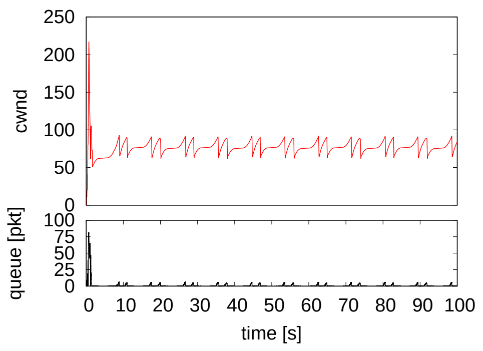
Figure C.17: TCP Reno flow on 10 Mbit/s bottleneck link with different AQM schemes.



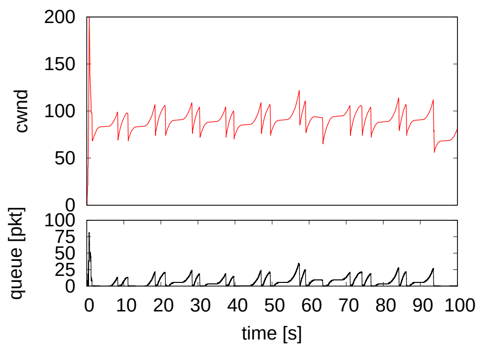
(a) RED with queue size of 1.0*BDP (RED1).



(b) Non-smoothed RED with queue size of 1.0*BDP (NRED1).

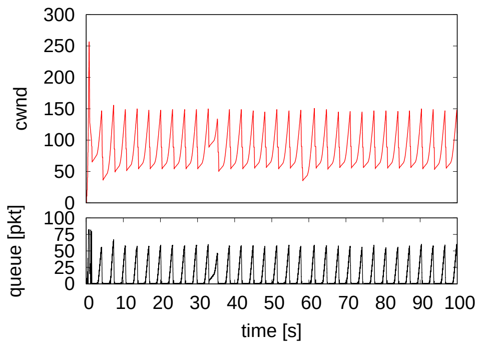


(c) CoDel.

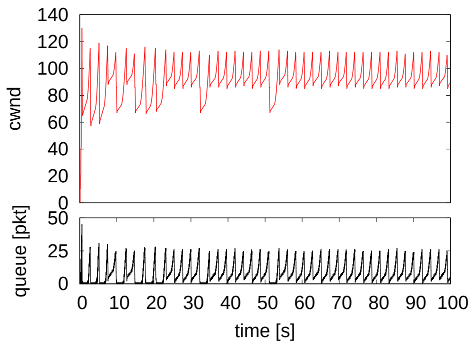


(d) PIE.

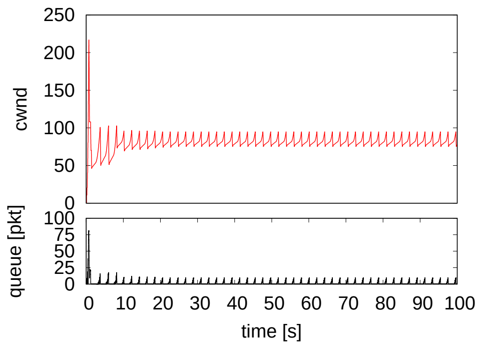
Figure C.18: TCP Cubic flow on 10 Mbit/s bottleneck link with different AQM schemes.



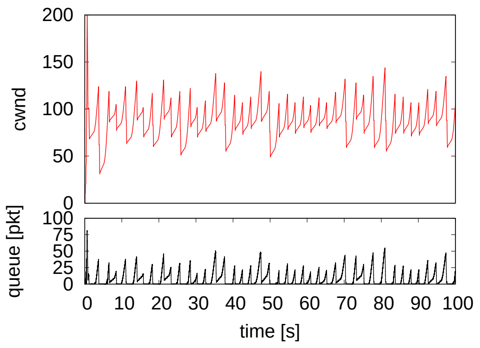
(a) RED with queue size of 1.0*BDP (RED1).



(b) Non-smoothed RED with queue size of 1.0*BDP (NRED1).

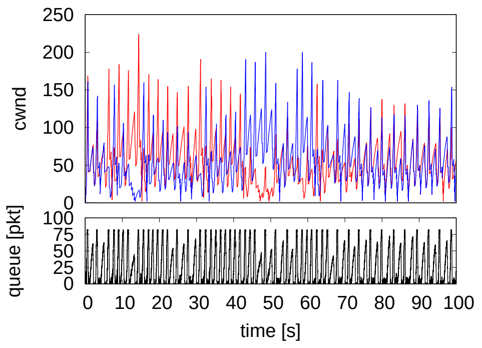


(c) CoDel.

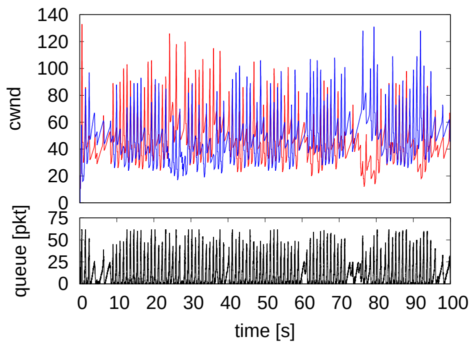


(d) PIE.

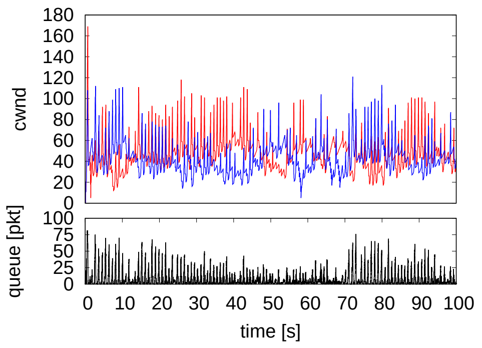
Figure C.19: H-TCP flow on 10 Mbit/s bottleneck link with different AQM schemes.



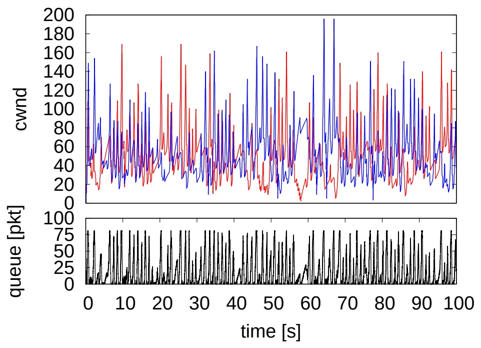
(a) RED with queue size of 1.0*BDP (RED1).



(b) Non-smoothed RED with queue size of 1.0*BDP (NRED1).



(c) CoDel.



(d) PIE.

Figure C.20: Two TCP SIAD flows ($Num_{RTT} = 20$) on 10 Mbit/s bottleneck link with different AQM schemes.

Bibliography

[1] Performance Enhancements in the Next Generation TCP/IP Stack. http://technet.microsoft.com/en-au/library/bb878127.aspx, Nov 2005.

[2] Bufferbloat. https://www.bufferbloat.net/, December 2014. Website.

[3] IKR Simulation and Emulation Library. http://www.ikr.uni-stuttgart.de/Content/IKRSimLib/, December 2014. Website.

[4] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, December 2014. Website.

[5] ns-3: a discrete-event network simulator for Internet systems. http://www.nsnam.org/, December 2014. Website.

[6] Qemu. http://www.qemu.org/, December 2014. Website.

[7] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-Host Congestion Control for TCP. *Communications Surveys Tutorials, IEEE*, 12(3):304–342, 2010.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.*, 41(4), Aug. 2010.

[9] M. Allman. TCP Congestion Control with Appropriate Byte Counting (ABC). RFC 3465 (Experimental), IETF, February 2003.

[10] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Proposed Standard), IETF, Sept. 2009.

[11] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), IETF, April 1999. Obsoleted by RFC 5681.

[12] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Packet Loss Metric for IPPM. RFC 2680 (Proposed Standard), IETF, Sept. 1999.

[13] L. Andrew, C. Marcondes, S. Floyd, L. Dunn, R. Guillier, W. Gang, L. Eggert, S. Ha, and I. Rhee. Towards a common TCP evaluation suite. In *Proc. PFLDnet*, 2008.

[14] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4):281–292, Aug. 2004.

[15] S. Athuraliya, V. H. Li, S. H. Low, and Q. Yin. REM: Active Queue Management. *IEEE Network*, 15:48–53, 2001.

[16] J. Aweya, M. Ouellette, and D. Y. Montuno. A Control Theoretic Approach to Active Queue Management. *Comput. Netw.*, 36(2-3):203–235, July 2001.

[17] A. Baiocchi, A. P. Castellani, and F. Vacirca. YeAH-TCP: yet another highspeed TCP. In *Proc. PFLDnet*, 2007.

[18] F. Baker and G. Fairhurst. IETF Recommendations Regarding Active Queue Management. Internet-Draft draft-ietf-aqm-recommendation-08, IETF, Aug. 2014.

[19] D. Bansal and H. Balakrishnan. Binomial congestion control algorithms. In *Proc. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 2, pages 631–640 vol.2, 2001.

[20] S. Bauer, D. Clark, and W. Lehr. The Evolution of Internet Congestion. In *Proc. 37th Research Conference On Communication, Information and Internet Policy (TPRC)*, 2009.

[21] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2 edition, 1992.

[22] S. Bhattacharyya, D. Towsley, and J. Kurose. The loss path multiplicity problem in multicast congestion control. In *Proc. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 2, pages 856–863 vol.2, Mar 1999.

[23] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Proposed Standard), IETF, Oct. 1989.

[24] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. In *IEEE Journal on Selected Areas in Communications*, volume 13, 1995.

[25] B. Briscoe. Flow rate fairness: dismantling a religion. *SIGCOMM Comput. Commun. Rev.*, 37(2):63–74, Mar. 2007.

[26] B. Briscoe, M. Kühlewind, and R. Scheffenegger. Accurate ECN Feedback in TCP. Internet-Draft draft-kuehlewind-tcpm-accurate-ecn-03, IETF, July 2011.

[27] B. Briscoe, M. Kuhlewind, D. Wagner, and K. D. Schepper. ECN the identifier of a new service model. http://www.ietf.org/proceedings/89/slides/slides-89-tsvarea-3.pdf, March 2014. IETF-89 Proceedings.

[28] B. Briscoe, R. Woundy, and A. Cooper. Congestion Exposure (ConEx) Concepts and Use Cases. RFC 6789 (Informational), IETF, Dec. 2012.

[29] L. Budzisz, R. Stanojević, A. Schlote, F. Baker, and R. Shorten. On the Fair Coexistence of Loss- and Delay-based TCP. *IEEE/ACM Trans. Netw.*, 19(6):1811–1824, Dec. 2011.

[30] L. Budzisz, R. Stanojevic, R. Shorten, and F. Baker. A strategy for fair coexistence of loss and delay-based congestion control algorithms. *Communications Letters, IEEE*, 13(7):555–557, July 2009.

[31] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang. TCP Westwood: End-to-end Congestion Control for Wired/Wireless Networks. *Wirel. Netw.*, 8(5):467–479, Sept. 2002.

[32] D. Cavendish, M. Gerla, and S. Mascolo. A control theoretical approach to congestion control in packet networks. *IEEE/ACM Transactions on Networking*, 12(5):893–906, Oct 2004.

[33] M. C. Chan and R. Ramjee. TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 71–82, New York, NY, USA, 2002. ACM.

[34] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.

[35] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing tcp's initial window. RFC 6928 (Experimental), IETF, April 2013.

[36] corbet. Pluggable congestion avoidance modules. http://lwn.net/Articles/128681/, March 2005.

[37] J. Crowcroft. Pricing the Internet. In *IEE Colloquium on Charging for ATM*, pages 1/1–1/4, Nov 1996.

[38] J. Crowcroft and P. Oechslin. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, July 1998.

[39] N. Dukkipati. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Dept. of Elec. Eng., Stanford University, Stanford, CA, USA, 2008.

[40] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Trans. Netw.*, 10(4):513–528, Aug. 2002.

[41] V. Firoiu and X. Zhang. Best Effort Differentiated Services: Trade-off Service Differentiation for Elastic Applications. In *Proc. of the IEEE International Conference on Telecommunications (ICT)*, 2000.

[42] S. Floyd. RED: Discussions of Setting Parameters. http://www.icir.org/floyd/REDparameters.txt, November 1997.

[43] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), IETF, December 2003.

[44] S. Floyd. Metrics for the Evaluation of Congestion Control Mechanisms. RFC 5166 (Informational), IETF, March 2008.

[45] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management. Technical report, Aug. 2001. http://www.icir.org/floyd/adaptivered/.

[46] S. Floyd, M. Handley, and J. Padhye. A Comparison of Equation-based and AIMD Congestion Control. Technical report, ACIRI, 2000. www.icir.org/tfrc/aimd.pdf.

[47] S. Floyd, M. Handley, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348 (Proposed Standard), IETF, September 2008.

[48] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), IETF, April 1999.

[49] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), IETF, April 2004.

[50] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, pages 397–413, Aug. 1993.

[51] S. Floyd, S. Ratnasamy, and S. Shenker. Modifying TCP's Congestion Control for High Speeds. Technical report, ICSI Networking Group, May 2005. www.icir.org/floyd/papers/hstcp.pdf.

[52] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40:40–40:54, Nov. 2011.

[53] R. J. Gibbens and F. P. Kelly. Resource pricing and the evolution of congestion control. In *Automatica*, volume 35, pages 1969–1985, Dec. 1999.

[54] S. Golestani and K. Sabnani. Fundamental observations on multicast congestion control in the internet. In *Proc. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 2, pages 990–1000, Mar. 1999.

[55] S. Gorinsky, M. Georg, M. Podlesny, and C. Jechlitschek. A Theory of Load Adjustments and its Implications for Congestion Control. *Journal of Internet Engineering*, 1:82–93, 2007.

[56] K.-J. Grinnemo and A. Brunstrom. A Survey of TCP-Friendly Congestion Control Mechanisms for Multimedia Traffic. Technical Report 2003:1, Karlstad University, Division for Information Technology, 2003.

[57] Y. Gu, D. Towsley, C. Hollot, and H. Zhang. Congestion Control for Small Buffer High Speed Networks. In *Proc. Twenty-sixth Annual Joint IEEE International Conference on Computer Communications (INFOCOM)*, pages 1037–1045, May 2007.

[58] L. Guo and I. Matta. The War Between Mice and Elephants. Technical report, Dept. of Comput. Sci., Boston Univ., Boston, MA, USA, 2001.

[59] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, 2008.

[60] G. Hardin. The Tragedy of the Commons. *Science*, 162(3859):1243–1248, December 1968.

[61] G. Hasegawa, K. Kurata, and M. Murata. Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet. In *International Conference on Network Protocols (ICNP)*, pages 177–186. IEEE, 2000.

[62] D. Hayes and G. Armitage. Improved coexistence and loss tolerance for delay based TCP congestion control. In *IEEE 35th Conference on Local Computer Networks (LCN)*, pages 24–31, Oct 2010.

[63] D. Hayes, D. Ros, L. Andrew, and S. Floyd. Common TCP Evaluation Suite. Internet-Draft draft-irtf-iccrg-tcpeval-00, IETF, 2014.

[64] D. A. Hayes and G. Armitage. Revisiting TCP Congestion Control Using Delay Gradients. In *10th International IFIP TC 6 Networking Conference*, volume 6641, pages 328–341. Lecture Notes in Computer Science, 2011.

[65] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, IETF, April 2012.

[66] P. Hurley, J.-Y. Le Boudec, P. Thiran, and M. Kara. ABE: providing a low-delay service within best effort. *IEEE Network*, 15(3):60–69, 2001.

[67] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, Aug. 1988.

[68] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), IETF, May 1992.

[69] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. *ACM Computer Communication Review*, 19:56–71, 1989.

[70] R. Jain. Congestion control in computer networks: issues and trends. *IEEE Network Magazine*, 4:24–30, 1990.

[71] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. Technical Report 301, DEC Research, 1984.

[72] R. Jain and K. Ramakrishnan. Congestion avoidance in computer networks with a connectionless network layer: concepts, goals and methodology. In *Proceedings of the Computer Networking Symposium*, pages 134–143, 1988.

[73] R. Jesup and Z. Sarker. Congestion Control Requirements For RMCAT. Internet-Draft draft-ietf-rmcat-cc-requirements-08, IETF, Nov. 2014.

[74] C. Jin, D. Wei, and S. Low. FAST TCP: motivation, architecture, algorithms, performance. In *Proc. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 4, pages 2490–2501, 2004.

[75] H. Jung, S. gyu Kim, H. Yeom, S. Kang, and L. Libman. Adaptive delay-based congestion control for high bandwidth-delay product networks. In *Proc. Thirtieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 2885–2893, April 2011.

[76] K. Kaneko, T. Fujikawa, Z. Su, and J. Katto. TCP-Fusion A Hybrid Congestion Control Algorithm for High-speed. In *Proc. PFLDnet*, 2007.

[77] R. Karp and C. Papadimitriou. Optimization Problems in Congestion Control. In *IEEE Symposium on Foundations of Computer Science*, pages 66–74, 2000.

[78] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. *ACM SIGCOMM Computer Communication Review*, 32(4):89–102, 2002.

[79] F. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8(1):33–37, January-February 1997.

[80] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: Shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3):237–252, Mar. 1998.

[81] T. Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, Apr. 2003.

[82] P. Key and L. Massoulié. User policies in a network implementing congestion pricing. In *Proc. Workshop on Internet Service Quality and Economics*, 1999.

[83] R. King, R. Baraniuk, and R. Riedi. Tcpafrica: An adaptive and fair rapid increase rule for scalable tcp. In *Proc. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1838–1848, 2005.

[84] V. Konda and J. Kaur. Rapid: Shrinking the congestion-control timescale. In *Proc. Twenty-ninth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1 –9, apr. 2009.

[85] R. Koodli and R. Ravikanth. One-way Loss Pattern Sample Metrics. RFC 3357 (Informational), IETF, Aug. 2002.

[86] M. Kühlewind. Adaptive and Scalable Congestion Control wanted! In *ISOC Workshop on Reducing Internet Latency*, 2013.

[87] M. Kühlewind and B. Briscoe. Chirping for Congestion Control - Implementation Feasibility. In *Proc. PFLDNeT*, 2010.

[88] M. Kühlewind, R. Scheffeneger, and B. Briscoe. Problem Statement and Requirements for a More Accurate ECN Feedback. Internet-Draft draft-ietf-tcpm-accecn-reqs-07, IETF, July 2014.

[89] N. Kuhn, P. Natarajan, D. Ros, and N. Khademi. AQM Characterization Guidelines. Internet-Draft draft-kuhn-aqm-eval-guidelines-02, IETF, August 2014.

[90] S. S. Kunniyur and R. Srikant. An Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. *IEEE/ACM Trans. Netw.*, 12(2):286–299, April 2004.

[91] A. Kuzmanovic and E. W. Knightly. TCP-LP: low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 14:739–752, August 2006.

[92] C. Lefelhocz, B. Lyles, S. Shenker, and L. Zhang. Congestion Control for Best Effort Service: why we need a new paradigm. *IEEE Network*, 10:10–19, 1996.

[93] D. Leith and R. Shorten. H-TCP: TCP for high-speed and long-distance networks. In *Proc. PFLDnet*, 2004.

[94] D. Leith, R. Shorten, G. McCullagh, J. Heffner, L. Dunn, and F. Baker. Delay-based AIMD congestion control. In *Proc. Fifth International Workshop on Protocols for FAST Long-Distance Networks (PDLFnet)*, 2007.

[95] Y.-T. Li. Evaluation of TCP Congestion Control Algorithms on the Windows Vista Platform. http://www.slac.stanford.edu/pubs/slactns/tn04/slac-tn-06-005.pdf.

[96] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A Loss and Delay-based Congestion Control Algorithm for High-speed Networks. In *Proc. First ACM International Conference on Performance Evaluation Methodolgies and Tools (valuetools)*, 2006.

[97] S. Liu, M. Vojnovic, and D. Gunawardena. Competitive and Considerate Congestion Control for Bulk Data Transfers. In *Proc. Fifteenth IEEE International Workshop on Quality of Service*, pages 1–9, June 2007.

[98] D. Loguinov and H. Radha. End-to-End Rate-Based Congestion Control: Convergence Properties and Scalability Analysis. *IEEE/ACM Trans. Netw.*, 11(4):564–577, 2003.

[99] H. Low, O. Paganini, and J. C. Doyle. Internet congestion control. *IEEE Control Systems Magazine*, 22:28–43, 2002.

[100] J. K. MacKie-Mason and H. R. Varian. Pricing the Internet. In *Public access to the Internet*, pages 269–314, 1993.

[101] L. Mamatas, T. Harks, and V. Tsaoussidis. Approaches to Congestion Control in Packet Networks. *Journal of Internet Engineering*, 1(1):22–33, January 2007.

[102] M. Mathis. Relentless Congestion Control. In *Proc. PFLDNeT*, 2009.

[103] M. Mathis, N. Dukkipati, and Y. Cheng. Proportional Rate Reduction for TCP. RFC 6937 (Experimental), IETF, May 2013.

[104] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), IETF, Oct. 1996.

[105] M. Mathis, J. Semke, and J. Mahdavi. The Rate-Halving Algorithm for TCP Congestion Control. Internet-Draft draft-mathis-tcp-ratehalving-00, IETF, August 1999.

[106] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

[107] S. Mehrotra, J. Li, S. Sengupta, M. Jain, and S. Sen. Hybrid Window and Rate Based Congestion Control for Delay Sensitive Applications. In *IEEE Global Telecommunications Conference (GLOBECOM)*, Dec 2010.

[108] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), IETF, Dec 1998.

[109] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 5:20–34, May 2012.

[110] C. Nicholson and T. Barth. Akamai Introduces Advanced Technology to Speed Downloads and Improve Online Video Quality Across its Global Platform. http://www.akamai.com/html/about/press/releases/2013/press_091313.html, September 2013.

[111] U. of North Carolina. TCP Evaluation Tmix Traces. http://hosting.riteproject.eu/tcpevaltmixtraces.tgz, 2008.

[112] T. Ott, T. V. Lakshman, and L. Wong. SRED: stabilized RED. In *Proc. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1346–1355 vol.3, Mar 1999.

[113] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, 2000.

[114] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *HPSR*, pages 148–155. IEEE, 2013.

[115] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), IETF, November 2000.

[116] R. Pletka, M. Waldvogel, and S. Mannal. PURPLE: predictive active queue management utilizing congestion information. In *Proc. 28th Annual IEEE International Conference on Local Computer Networks (LCN)*, pages 21–30, Oct 2003.

[117] M. Podlesny and S. Gorinsky. RD network services: differentiation through performance incentives. *SIGCOMM Comput. Commun. Rev.*, 38(4):255–266, Aug. 2008.

[118] O. Pomerantz. *Linux Kernel Module Programming Guide*. iUniverse, Incorporated, 2000.

[119] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[120] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), IETF, Sept. 2001.

[121] K. K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Trans. Comput. Syst.*, 8(2):158–181, May 1990.

[122] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. www4.ncsu.edu/ rhee/export/bitcp/cubic-paper.pdf, 2005.

[123] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop (PAM)*, 2003.

[124] M. Rio, M. Goutelle, T. Kelly, R. H. Jones, Jean, and Y. T. Li. A Map of the Networking Code in Linux Kernel 2.4.20. Technical Report DataTAG-2004-1, FP5/IST DataTAG Project, Mar. 2004.

[125] S. Ryu, C. Rump, and C. Qiao. Advances in internet congestion control. In *IEEE Communications Surveys*, 2001.

[126] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[127] Z. Sarker, V. Singh, X. Zhu, and M. Ramalho. Test Cases for Evaluating RMCAT Proposals. Internet-Draft draft-ietf-rmcat-eval-test-00, IETF, August 2014.

[128] P. Sarolahti and A. Kuznetsov. Congestion Control in Linux TCP. In *Proc. of the FREENIX Track: USENIX Annual Technical Conference*, pages 49–62, 2002.

[129] R. Scheffenegger, M. Kuehlewind, and B. Trammell. Additional negotiation in the TCP Timestamp Option field during the TCP handshake. Internet-Draft draft-scheffenegger-tcpm-timestamp-negotiation-05, IETF, Oct. 2012.

[130] S. Seth and M. A. Venkatesulu. *TCP/IP Architecture, Design and Implementation in Linux*. Wiley-IEEE Computer Society Press, 2008.

[131] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817 (Experimental), IETF, December 2012.

[132] S. Shenker. Fundamental Design Issues for the Future Internet. *IEEE J. Sel. A. Commun.*, 13(7):1176–1188, Sept. 2006.

[133] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in Computer Networks: Reshaping the Research Agenda. *SIGCOMM Comput. Commun. Rev.*, 26(2):19–43, Apr. 1996.

[134] H. Shimonishi, T. Hama, and T. Murase. TCP-Adaptive Reno for Improving Efficiency-Friendliness Tradeoffs of TCP Congestion Control Algorithm. In *Proc. IEEE Global Telecommunications Conference (GLOBECOM)*, 2005.

[135] R. Shorten and D. J. Leith. On queue provisioning, network efficiency and the Transmission Control Protocol. *IEEE/ACM Trans. Netw.*, 15(4):866–877, 2007.

[136] V. Singh and J. Ott. Evaluating Congestion Control for Interactive Real-time Media. Internet-Draft draft-ietf-rmcat-eval-criteria-02, IETF, July 2014.

[137] N. Spring, D. Wetherall, and D. Ely. Robust Explicit Congestion Notification (ECN) Signaling with Nonces. RFC 3540 (Experimental), IETF, June 2003.

[138] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001 (Proposed Standard), IETF, January 1997.

[139] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. TechReport MSR-TR-2005-86, Microsoft Research publications, July 2005.

[140] R. B. Technologies. GENI Exploring Networks of the Future. http://www.geni.net/, October 2014.

[141] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, Dec. 2002.

[142] R. Wang, M. Valla, M. Sanadidi, B. K. F. Ng, and M. Gerla. Efficiency/Friendliness Tradeoffs in TCP Westwood. In *Proc. Seventh IEEE Symposium on Computers and Communications*, July 2002.

[143] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). *ACM Computer Communication Review*, 21:32–43, 1991.

[144] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. *Linux Network Architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[145] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F. D. Smith. Tmix: A Tool for Generating Realistic TCP Application Workloads in Ns-2. *SIGCOMM Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[146] T. Werthmann, M. Kaschub, M. Kühlewind, S. Scholz, and D. Wagner. VMSimInt: A Network Simulation Tool Supporting Integration of Arbitrary Kernels and Applications. In *Proc. 7th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2014.

[147] J. Widmer, R. Denda, and M. Mauve. A survey on TCP-friendly congestion control. *IEEE Network*, 15(3):28–37, 2001.

[148] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proc. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 4, pages 2514–2524, March 2004.

[149] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP Congestion Avoidance Algorithm Identification. In *Proc. 31st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 310–321, 2011.

[150] Y. Yang, M. S. Kim, and S. Lam. Transient behaviors of TCP-friendly congestion control protocols. In *Proc. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1716–1725 vol.3, 2001.