ARTICLE IN PRESS

Available online at www.sciencedirect.com

ScienceDirect

Computer Networks xxx (2007) xxx–xxx

Computer Networks

www.elsevier.com/locate/comnet

Receiver-centric congestion control with a misbehaving receiver: Vulnerabilities and end-point solutions [☆]

A. Kuzmanovic [a,*], E.W. Knightly [b]

[a] Northwestern University, Department of EECS, 2145 Sheridan Road, Evanston, IL 60208, United States
[b] Rice University, Department of ECE, 6100 South Main, Houston, TX 77005, United States

Received 23 April 2006; received in revised form 20 November 2006; accepted 21 November 2006

Responsible Editor: Vassilis Tsaoussidis

Abstract

Receiver-driven TCP protocols delegate key congestion control functions to receivers. Their goal is to exploit information available only at receivers in order to improve latency and throughput in diverse scenarios ranging from wireless access links to wireline and wireless web browsing. Unfortunately, in contrast to today's sender-driven protocols, receiver-driven congestion control introduces an incentive for misbehavior. Namely, the primary beneficiary of a flow (the receiver of data) has both the means and incentive to manipulate the congestion control algorithm in order to obtain higher throughput or reduced latency. In this paper, we study the deployability of receiver-driven TCP in environments with untrusted receivers which may tamper with the congestion control algorithm for their own benefit. Using analytical modeling and extensive simulation experiments, we show that deployment of receiver-driven TCP must strike a balance between enforcement mechanisms, which can limit performance, and complete trust of endpoints, which results in vulnerability to cheaters and even DoS attackers.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Receiver-centric TCP; Congestion control; Misbehavior; Trust; Performance

1. Introduction

Recent advances in TCP congestion control design have demonstrated the ability to significantly improve TCP performance in a variety of scenarios, ranging from high-speed (e.g., [1,2]) to mobile and wireless networks (e.g., [3,4]). However, each such advance introduces the following dilemma: if a user can obtain a significant increase in throughput via an optimized congestion control algorithm, how can the network or the other end point distinguish among (i) users with optimized protocol stacks, (ii) "cheater's" that have modified protocol stacks that maximize their own throughput without regard to fairness or network stability, and (iii) attackers that seek only to transmit at a high rate in order to deny

[☆] A subset of this work appears in the Proceedings of IEEE ICNP '04 (42).
[*] Corresponding author. Tel.: +1 847 467 5519; fax: +1 847 491 4455.
E-mail addresses: akuzma@northwestern.edu (A. Kuzmanovic), knightly@rice.edu (E.W. Knightly).

1389-1286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2006.11.021

service to others. More precisely, the question becomes how can misbehavior be detected in the presence of widely variable protocol performance profiles? And most importantly, protocol innovations often introduce novel security challenges, which, if not considered a priori, may have devastating consequences once such innovations become deployed.

TCP variants that are widely deployed today are sender-centric protocols in which the sender performs important functions such as congestion control and reliability, whereas the receiver has minimum functionality via transmission of acknowledgements to the sender. Yet, it is becoming evident that increasing the functionality of *receivers* can significantly improve TCP performance [5–10]. Indeed, a key breakthrough in this design philosophy is represented by fully receiver-centric protocols in which *all* control functions are delegated to receivers [11,12]. The benefits that are being established for this innovative design include improved TCP throughput and an array of other performance enhancements: (i) improved loss recovery; (ii) more robust congestion control; (iii) improved power management for mobile devices; (iv) a solution to the handoff problem in wireless networks; (v) improved behavior of network-specific congestion control; (vi) easy migration to a replicated server during handoffs; (vii) improved bandwidth aggregation; and (viii) improved web response times.

However, both sender- and receiver-centric protocols implicitly rely on the assumption that both endpoints cooperate in determining the proper rate at which to send data, an assumption that is increasingly invalid today. With sender-centric TCP-like congestion control, the sending endpoint may misbehave by disobeying the appropriate congestion control algorithms and send data more quickly. Fortunately, the lack of a strong incentive for selfish Internet users to do so (uploading vs. downloading) appears to be the main guard against such misbehavior. Moreover, while it has been discovered that misbehaving *receivers* can perform DoS attacks or steal bandwidth even with sender-centric protocols [13], it has been shown that it is possible to modify TCP to entirely eliminate this undesirable behavior [14,13].

On the other hand, receiver-centric congestion control presents a perfect match for a misbehaving user: the receiving endpoint performs *all* congestion control functions, and has both the incentive (faster web browsing and file downloads) and the opportunity (open source operating systems) to exploit protocol vulnerabilities. In this paper, we explore the tradeoffs and tensions between performance and trust for receiver-centric transport protocols. In particular, given the above benefits (i)–(viii), and clear vulnerabilities, our goal is to evaluate whether it is possible for HTTP, file, and streaming servers in the Internet to deploy receiver centric transport protocols while striking a balance between performance enhancements and protection against misbehavior. We focus on the class of receiver-driven protocols because their deployment introduces a set of novel security challenges that can have devastating effects on the widely-deployed HTTP, file, and streaming servers in the Internet. Moreover, we show that *none* of the existing solutions are able to efficiently protect the servers from such receiver misbehaviors.

In this paper, we first anticipate and analyze a set of possible receiver misbehaviors, ranging from classical denial-of-service attacks, e.g., receiver request flooding, to more moderate and consequently harder-to-detect misbehavior. We divide misbehaviors into two classes: the first is long-time-scale misbehaviors that manipulate the additive-increase-multiplicative-decrease (AIMD) or retransmission timeout (RTO) parameters such that flows steal bandwidth over longer time-scales; the second class is short-time-scale misbehaviors that forge parameters such as the initial congestion-window size, such that these flows improve the short-file response times at the expense of well-behaved flows. We develop an analytical model by generalizing [15] to predict the throughput that a misbehavior will obtain as a function of modified AIMD parameters. Moreover, for small files we derive an expression for the response time for file download under modifications of the initial congestion window. In both cases, we show that such modifications can lead to misbehaving flows achieving dramatically higher bandwidths and reduced latency as compared to behaving flows.

Second, we evaluate and discuss a set of state-of-the-art router- and edge-based mechanisms designed to detect and thwart denial-of-service attacks and other flow misbehaviors. Unfortunately, we find that some of the schemes, e.g., [16], are completely unable to detect any receiver misbehavior, whereas others, e.g., [17], are fundamentally limited in their ability to detect even very severe end-point misbehaviors. The key reason is the lack of knowledge of flows' round-trip times, which forces such schemes to penalize flows based on their absolute throughput, which in a heterogeneous-RTT environment typically results in punishing short-RTT flows. Moreover, even when a high-rate attack can be detected at a router, we show that protection schemes such as *pushback* [18], can have catastrophic consequences: not only that the scheme cannot prevent the attack, but it can actually significantly improve its effectiveness.

Next, we propose and evaluate a set of sender-side mechanisms designed to detect and thwart receiver misbehavior, yet without *any* help from a potentially misbehaving receiver. We initially focus on long time-scales and develop a TFRC-based scheme in which senders (i) independently estimate RTT and loss rate without any cooperation from a potentially misbehaving receiver, (ii) dynamically compute the TCP-friendly rate, and (iii) detect out-of-profile behavior. While this end-point approach at the sender-side is able to accurately detect even slight receiver misbehaviors and strictly enforce TCP-friendliness, we show that a fundamental tradeoff arises from the fact that in the absence of trust between the sender and receiver, it becomes problematic for the sender to infer whether the receiver is misbehaving or legitimately trying to optimize its performance with an enhanced protocol stack. For example, we show that a client applying receiver-driven TCP together with the congestion-control mechanisms of [3,19] can improve throughput six times in a simple wireless scenario. We therefore propose a detection methodology that protects the system against DoS attacks and severe resource stealing and ignores modest stealing that is not easily distinguishable from enhanced TCP stacks. In this way, we attempt to strike a balance between performance and trust – fostering innovation and deployment of enhanced TCP stacks while also providing counter-measures against severe abuse.

Finally, we analyze short-time-scale receiver misbehaviors, and show that the performance vs. trust tension significantly magnifies over shorter time-scales. For example, we conduct a web experiment and show that a malicious client that uses excessively long initial window size and also forges exponential backoff timers, can not only significantly improve its own response time, but can also drastically degrade the response times of the background traffic. While sender-based enforcement mechanisms (e.g., rate limiting) are again successful against DoS attacks, we show that in HTTP scenarios dominated by short-lived flows, such mechanisms can often limit receiver-driven TCP performance to a level *below* that achievable by today's sender-based TCP.

The remainder of this paper is organized as follows. In Section 2, we present background on receiver-based transport protocols. Next, in Section 3, we analyze and analytically model protocol vulnerabilities. Sections 4 and 5 propose and evaluate a set of sender-based solutions targeted to protect the system over long- and short-time-scales, respectively, while Section 6 analyzes limitations of existing solutions in detecting end-point misbehaviors. Finally, in Section 7 we conclude.

2. Background

In this section, we review transport protocols that delegate some or all control functions to receivers. For scenarios ranging from web browsing to wireless networks, the key advantages of receiver-driven protocols are improved response times and throughput due to exploitation of information available at the receiver.

2.1. Delegating control functions to receivers

One of the first transport protocols that exploits increased receiver functionality is Clark et al.'s NETBLT [5], which makes error recovery more efficient by placing the data retransmission timer at the receiver. In later work, an increased set of control functions appear at the receiver, either for performance or practical reasons (e.g., to decrease the computation burden at the sender). For example, Sinha et al.'s WTCP [8] calculates the sending rate at the receiver; Floyd et al.'s TFRC [6] maintains the loss history and computes the TCP-friendly rate at the receiver; Tsaoussidis and Zhang's TCP-Real [10] tracks loss events and determines the data delivery rate at the receiver; Spring et al. [9] and Mehra et al. [7] add functionality to the receiver to control the bandwidth shares of incoming TCP flows, i.e., by adapting the receiver's advertised window and delay in transmitting *ack* messages, the receiver is able to control the bandwidth share on the access link according to the client's needs.

2.2. Fully receiver-driven transport protocols

In contrast to the above protocols, *all* control functions are delegated to receivers in Web Transport Protocol (WebTP) [11] and Reception Control Protocol (RCP) [12]. Hsieh et al. [12] argue that the key advantage of fully receiver-centric transport protocols is that the *receiver* controls *how much data can be sent*, and *which data should be sent* by the sender. Below, we summarize some of the performance and functionality gains that a fully receiver-centric protocol can achieve in the large-scale server scenarios as well as in a scenario where the mobile host acts as the receiver for traffic from a wireline sender.

2.2.1. Performance gains

We elaborate on two of the sources of performance gains explored in [12]. The first is improved loss recovery. In particular, while TCP *ack* packets are resilient to losses due to their cumulative nature, they provide little information that the sender can use to effectively recover from losses. While TCP Sack [20] is able to recover from losses by using three "Sack blocks", the effectiveness of recovery is limited to the extend to which the sender can accurately construct the receiver buffer in a timely fashion. Hence, heavy losses on the forward path, coupled with a lossy reverse path (typical for wireless environments), may prevent the TCP Sack sender from accurately constructing the receiver's buffer state. On the contrary, the receiver has direct access to the receive buffer, and hence can always recover from losses in an effective fashion, without incurring the overhead inaccuracies of TCP Sack.

The second source of performance gains arise via wireless-aware congestion control. Namely, the wireless link typically plays a defining role in determining the characteristics of an end-to-end path. Hence, "wireless-aware" congestion control algorithms exploit information about the characteristics of the wireless link (e.g., loss classification, RTT sample filtering or reasons for non-congestion related outages (handoffs or channel blackouts)). Since the receiver is adjacent to the wireless last-hop, it has first-hand knowledge about the above information. We will show in Section 4.3 that such an approach can improve throughput for six times in a simple wireless scenario.

2.2.2. Functionality gains

The fact that receiver-centric protocol design delegates the entire protocol "intelligence" to the receiver yields a number of functionality improvements. First, sender-based TCP places all protocol state on the servers which must handle large scale workloads (e.g., web servers). On the other hand, receiver-driven transport protocols can significantly reduce the complexity of the server implementation since they distribute the state management across the large number of clients.

Second, during periods of mobility, a mobile host with heterogeneous wireless interfaces can benefit from the fact that the transport protocol functionality is concentrated at the receiver. For example, when a mobile host performs a handoff from one access network to another, it can avoid connection disruptions due to temporary link outage by using both interfaces simultaneously if the transport layer is able to use multiple interfaces without suffering from performance degradation due to persistent packet reordering. Furthermore, the mobile user can benefit from migrating to a replicated server, either because the new interface has no access to the old server, or for performance considerations.

2.3. RCP protocol

Here, we provide a brief overview of RCP, variants of which we consider for the remainder of the paper.[1]

All TCP variants provide reliable in-sequence data delivery to the application, with protocol operations consisting mainly of four mechanisms: connection management, flow control, congestion control, and reliability. Fig. 1 depicts a schematic view of the interaction between sender and receiver in TCP, together with several state variables.[2]

Observe that except for connection management, which needs to be implemented at both ends, Fig. 2 indicates that RCP delegates all other control functions to the receiver. Thus, either the sender or receiver can initiate connection setup, after which the receiver becomes fully responsible for reliability, flow control, and congestion control, using the same window-based mechanisms employed in sender-driven TCP. Since RCP shifts the control of data transfer from the sender to receiver, the *data-ack* style of message exchange in TCP is no longer applicable. Instead, to achieve the self-clocking characteristics of TCP, RCP uses *req-data* exchange for data transfer, where any data transfer from the sender is preceded with an explicit request (*req*) from the receiver. Equivalently, the RCP receiver uses incoming *data* packets to clock the requests for

[1] While we focus on RCP, similar receiver incentives and protocol vulnerabilities hold whether protocols delegate some or all control functions to receivers, e.g., TFRC [6] and WebTP [11], respectively.

[2] Figs. 1 and 2 are taken from Refs. [12].

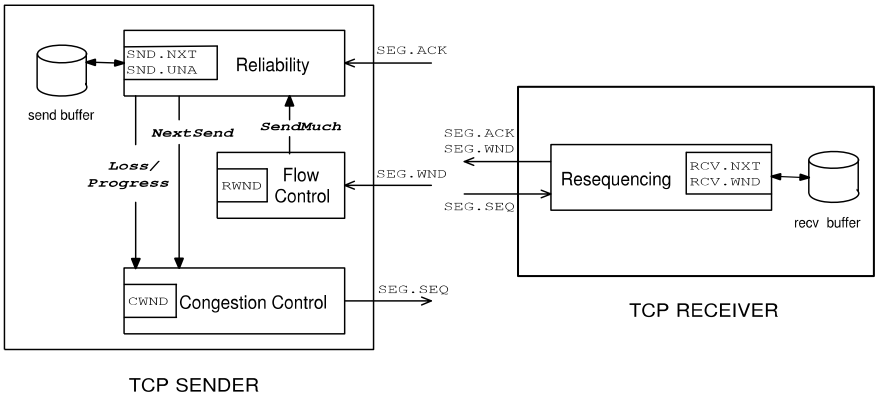

Fig. 1. TCP functionalities at the sender and receiver.

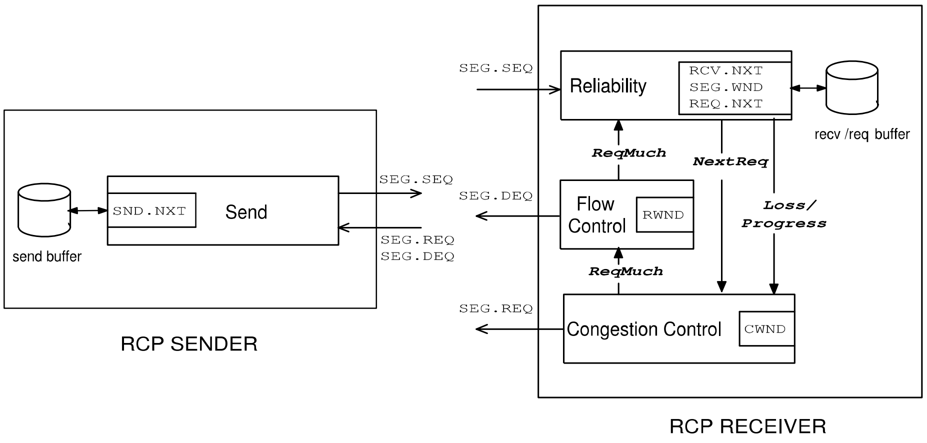

Fig. 2. RCP functionalities at the sender and receiver.

new data. In summary, RCP represents a clone of sender-side TCP which simply transfers *all* important control functionalities to the receiver. (We interchangeably use the terms RCP and receiver-driven TCP.)

However, the fact that *all* control functions are delegated to receivers raises a fundamental security concern for misbehaving receivers that will manipulate protocol parameters (all available at the receiver) and gain significant performance benefits. This concern is amplified by the fact that receivers would have the opportunity (open source operating systems requiring a minor change), and incentive (faster web browsing and file downloads) to perform such activities.

3. Vulnerabilities

In this section we analyze receiver misbehaviors which range from DoS attacks to more moderate (hence harder to detect) manipulations of congestion control parameters. We then develop an analytical model by generalizing [15] to predict the throughput that a misbehavior will obtain as a function of the modified AIMD parameters α and β, as well as the retransmission timeout RTO. Finally, for small files we derive an expression for the response time for file download under modifications of the initial congestion window.

3.1. Receiver misbehaviors

Here, we treat two classes of misbehaviors in the context of receiver-driven transport protocols: denial-of-service attacks and resource stealing. The key distinction between the two lies in the primary goal of the misbehaving client: DoS attackers aim to deny service to the background flows without necessarily achieving a particular benefit for themselves, whereas resource stealers aim to gain a performance benefit by stealing resources from the background flows (without necessarily starving them).

3.1.1. Denial of service attacks

We begin with an extreme scenario and show that an RCP sender can become an easy target of a DoS attack. Indeed, Fig. 2 shows that the RCP sender listens to the request packets from the receiver, and replies by sending data packets without *any* control, as all control functions are delegated to the receiver for performance reasons. Hence, flooding the sender with short *req* packets (the same size as the *ack* packets, ∼40 Bytes) may force the RCP sender to flood the reverse path (from the server to the client) with much longer *data* packets (typically ∼1500 Bytes), and congest the network.

To demonstrate the vulnerability of fully receiver-driven transport protocols, we use ns2 to simulate the above request-flood attack and show the result in Fig. 3. In this scenario, seven TCP Sack flows share a link, and at time 300 s, an RCP flow joins the aggregate (we provide the exact simulation parameters and topology in Section 4). However, we remove the congestion control functions from the RCP flow (by re-tuning the appropriate RCP parameters at the receiver – details are given below), such that it floods the server with requests. Consequently, the RCP flow utilizes the entire bandwidth and denies service to the background traffic by exploiting TCP's well-known vulnerability to attacks by high-rate non-responsive flows.

3.1.2. Resource stealing

In contrast, an unscrupulous receiver may moderately re-tune its parameters in an attempt to steal bandwidth from other flows in the network while eluding detection. Indeed, we will quantify the extent to which it is harder to detect flows that moderately disobey some (but not all) congestion control rules (e.g., decrease the window size upon a packet loss, but do not halve it), than it is to detect flows that dramatically violate one or more congestion control rules. While we do not underestimate the creativity of misbehaving receivers, in this paper we treat only easy-to-implement misbehaviors that can be achieved by changing protocol parameters; namely, each parameter can be modified by changing a *single* line of code.

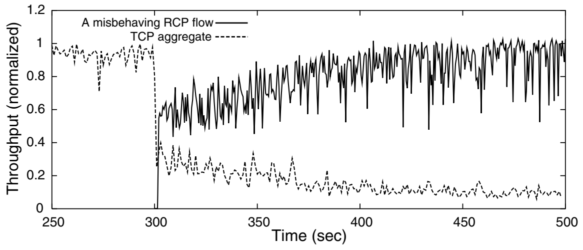

Fig. 3. RCP receiver performs a DoS attack by flooding the sender with requests.

While the space of possible receiver misbehaviors is vast, we focus on parameter-based misbehaviors simply because they are easy to implement. While receivers could clearly use other mechanisms to achieve similar rates, we demonstrate in Section 4 that this does not affect the detection problem. In other words, the proposed solutions are capable of detecting user misbehaviors independently of the method used to implement the same. Furthermore, in this paper we do *not* treat the problem of application-level misbehaviors such as parallel download (where a malicious user opens multiple transport-layer connections to parallely download different partitions of a file from a server). Nevertheless, observe that the misbehaviors analyzed in this paper are much more generic: (i) they can be simply and entirely implemented at the receivers; (ii) a malicious receiver can achieve a performance benefit even in scenarios where a single transport connection is used for download (e.g., in the HTTP 1.1 web-server scenarios or in the non-partitioned FTP-download scenarios).

The first parameter of interest is the *additive-increase parameter* α, which has a default value of one packet per round-trip time. By increasing the window size more aggressively ($\alpha > 1$), a flow can achieve higher throughput.

The second parameter is the *multiplicative-decrease parameter* β which has a default value of 0.5 such that the congestion window is halved upon the receipt of congestion indication. Again, the receiver can potentially utilize more bandwidth by decreasing the window only moderately via $\beta > 0.5$.

The third parameter is the *retransmission timeout RTO*. Both TCP and RCP use a retransmission timer to ensure data delivery in the absence of any feedback from the remote peer.[3] In both cases, this

[3] In the sender-driven TCP scenario, it is the absence of *ack* packets from the TCP receiver, while in the receiver-driven RCP scenario, it is the absence of *data* packets from the RCP sender.

value is computed using smoothed round-trip time and round-trip time variation. RFC 2988 [21] recommends to lower- and upper-bound this value to 1 and 60 s, respectively. Thus, a malicious receiver may easily change these values. For example, by setting the RTO to a small value (e.g., 100 ms), one can expect to achieve throughput improvements in high packet loss ratio environments, because the misbehaving flow would back-off significantly less aggressively than behaving flows would.

Finally, the fourth parameter of interest is *the initial window size W*. The default is two segments, whereas RFC 2414 [22] recommends increasing this parameter to a value between two and four segments (roughly 4 Kbytes) to achieve a performance improvement. A misbehaving receiver might wish to further improve its performance (without caring much about problems such as congestion collapse), and increase this parameter even more. By doing so, the receiver can maliciously jump-start the RCP flow (this is exactly what we did, among other things, in Fig. 3 by setting $W = 10$) and improve its throughput. However, this parameter is expected to be crucial in improving the short file-size response times which are typical for web browsing.

3.2. Modeling misbehaviors

Manipulations of parameters α, β, and RTO enable misbehaving receivers to steal bandwidth over longer time scales, whereas modifying the parameter W reduces latency for small files, hence over shorter time-scales. Here, we develop analytical models to predict the amount of stolen bandwidth and reduced latency over long- and short-time-scales, respectively.

3.2.1. Long time scales

We begin with the well-known TCP throughput formula (Eq. (30) in [15]) that expresses average TCP rate B as a function of the round-trip time RTT, steady-state loss event rate p, TCP retransmission timeout value RTO, and number of packets acknowledged by each ack b (typically $b = 1$ [23])

$$B \approx \frac{1}{\mathrm{RTT}\sqrt{\frac{2bp}{3}} + \mathrm{RTO}\min\left(1, 3\sqrt{\frac{3bp}{8}}\right)p(1+32p^2)}. \tag{1}$$

Using the stochastic TCP model and methodology of [15], we generalize the above result to a scenario

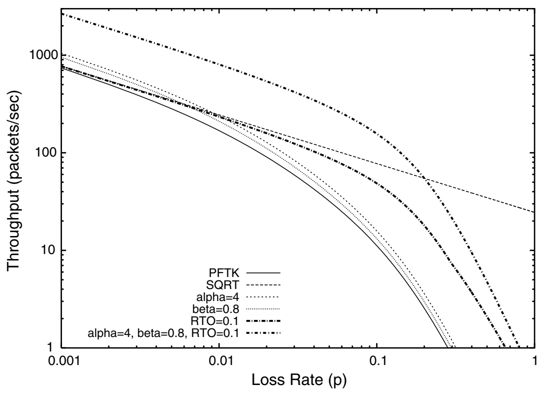

Fig. 4. Long-time-scale misbehaviors – numerical results.

with arbitrary values of α and β.[4] Denoting d as $1/\beta$, we have approximated B by

$$\frac{1}{\mathrm{RTT}\sqrt{\frac{2bp(d-1)}{\alpha(d+1)}} + \mathrm{RTO}\min\left(1, 3\sqrt{\frac{bp(1+d)(d-1)}{2\alpha d^2}}\right)p(1+32p^2)}. \tag{2}$$

We provide the key steps of the derivation in Appendix. Note the two corner cases: for $\alpha = 1$ and $\beta = 0.5$, Eqs. (1) and (2) are equivalent; when $\beta = 1$ (when $d = 1$), then $B \to \inf$, i.e., if the congestion window is never decreased upon a packet loss, the throughput will theoretically converge to infinity. We explore intermediate cases as follows.

Fig. 4 shows numerical results for TCP (and hence RCP) throughput as a function of the packet loss rate. *PFTK* denotes the formula from [15] (Eq. (1), with $b = 1$ and RTO = 1 s), while SQRT is the "square-root" formula from [26] (the same as Eq. (1), only without the RTO part). Next, we plot the throughput that a malicious receiver can achieve, according to Eq. (2), by manipulating α, β, and RTO (exact values are shown in the figure).

First, observe that by re-tuning α to four, one can double the throughput (y-axis is in logarithmic scale), while re-tuning β to 0.8 ($d = 1.25$) one can steal somewhat less bandwidth. More generally, according to Eq. (2), setting α to a value larger than one, enables a flow to achieve approximately $\sqrt{\alpha}$ higher throughput as compared to a well-behaved TCP flow and for the same packet loss rate. Second, observe that both curves ($\alpha = 4$ and $\beta = 0.8$) have a

[4] A deterministic model for TCP-friendly AIMD congestion control with arbitrary α and β could be found in [24]. Likewise, a stochastic model similar to the one proposed in Appendix could be found in [25].

shape similar to the *PFTK* curve. This indicates that the amount of stolen bandwidth (the difference between the misbehaving and the *PFTK* curve) is approximately independent of the packet loss ratio. On the other hand, notice that this is not the case for the RTO parameter (e.g., RTO = 100 ms), where the amount of stolen bandwidth increases as the packet loss ratio increases. This is because timeouts occur more frequently in higher packet loss ratio environments, and thus, disobeying the exponential backoff rules enables significant throughput gains in such environments. Furthermore, by re-tuning all parameters together ($\alpha = 4$, $\beta = 0.8$, RTO = 0.1), the model predicts significant stealing effects, where the misbehaving flow utilizes approximately 10 (for $p = 0.02$) to 20 (for $p = 0.1$) times more bandwidth than behaving flows. Finally, observe that the SQRT formula significantly overestimates the TCP-friendly rate for higher packet loss ratios (where the exponential backoffs play a key role), hence this formula is not suitable for detection purposes (to be explained in detail below).

3.2.2. Short time scales

Here, we develop an expression for the response time for file download under modifications of the initial congestion window parameter W. We model only the exponential increase phase, which is the only phase that the majority of short-lived flows ever enter [27].[5] We show in Section 5.2 that the expression accurately captures the response times of short files in a web browsing experiment.

The exponential increase phase for receiver-driven TCP is the same as in the sender-driven scenario, with the difference that the receiver has the leading role. It sends the first two *req* packets to the sender (the default initial window size is two segments), which replies with the first two *data* packet. Next, the receiver doubles the congestion window and sends four *req* packets to the sender. Denote T_r as the response time of a regular (behaving) RCP flow, N as the file (flow) size in packets, and RTT as the round-trip time. In such a scenario, the response time for a flow of size N is

$$T_{\rm r} = \max(\mathrm{RTT}, \lceil \log_2 N \rceil\, \mathrm{RTT}). \tag{3}$$

Next, denote $T_{\rm m}$ as the response time of a malicious flow that sets the initial window size W to a value larger than two. Further, denote s as the packet size in bits and C as the available bandwidth in bits/s. Then, when the file size satisfies $N \leqslant W$, we have that

$$T_{\rm m} = \max(\mathrm{RTT}, Ns/C). \tag{4}$$

In other words, if the initial window size is set to a number larger than the file size, the file will be downloaded in a "single burst", and thus the actual response time equals the burst size, lower-bounded by RTT. Otherwise, if $N > W$, we have that

$$T_{\rm m} = \max(\mathrm{RTT}, Ws/C) + \lceil \log_2 N - \log_2 W \rceil \mathrm{RTT}. \tag{5}$$

The first part of Eq. (5) is similar to Eq. (4). It says that the first W packets are downloaded in a single burst, whereas the rest are transferred in a "jump-started" exponential increase phase. A simple calculation shows that a misbehaving user can indeed significantly improve the file response time by manipulating the initial window size parameter. For example, our simple model indicates that for $C = 10$ Mb/s, a 70 kByte file can be transferred within a single RTT when the initial window size W is set to 70 or more packets: *seven* times faster than what a behaving flow achieves.

4. An end-point solution

In this section, we evaluate the potential of an end-point scheme to detect receiver misbehaviors. The key advantage of an end-point (vs. network-based) approach is the ability of the sender to estimate the round-trip time and loss rate on the path to the receiver, and hence enforce a much "tighter" TCP-friendly throughput profile. However, a fundamental problem arises from the fact that in the absence of trust between the sender and receiver, it is problematic for the sender to infer whether the receiver is misbehaving as defined in Section 3 or legitimately trying to optimize its performance.

4.1. Sender-side verification

In order to detect receiver misbehavior, the sender requires increased functionality beyond its role as a slave to the receiver's request packets (see Fig. 2). Our objective is to add the minimum functionality to the sender that will enable it to robustly detect receiver misbehavior over long-time scales (we treat the short-time-scale misbehavior detection problem in Section 5.2), yet without *any* help from a

[5] See Refs. [28–30] for more sophisticated models for the latency of *well-behaving* TCP flows.

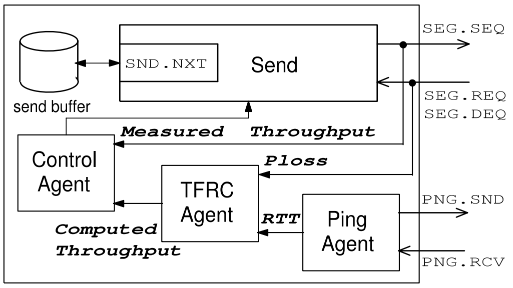

Fig. 5. Secure RCP sender.

potentially misbehaving receiver. While this new functionality inevitably increases the sender-side implementation complexity, we will demonstrate that it represents a general solution to the bandwidth-stealing receiver-induced misbehaviors.

Fig. 5 depicts the key components of such a solution. Eq. (1) indicates that knowledge of RTT and packet loss ratio is enough to compute the TCP-fair throughput, and consequently to detect out-of-profile flows. Obviously, our solution is able to measure RTT and the packet loss ratio, and hence enforce a more precise traffic profile than *any* network-based solution. Indeed, we will demonstrate in Section 6 the fundamental limitation of network-router-based schemes to accurately estimate the connection parameters (e.g., RTT), and to enforce a precise TCP-friendly rate.

Because the sender must estimate RTT and packet loss ratio without any cooperation from the untrusted receiver, the sender transmits *ping* packets that the receiver has no incentive to delay, as a larger RTT implies a lower bandwidth profile.[6] One problem with the ping approach is that ICMP (e.g., *ping*) messages may be given different treatment at routers. Moreover, because not all hosts will respond to *ping* messages (e.g., due to a system-wide policy), the sender can effectively measure RTT in the following way. By sending short TCP packets (e.g., ACK) to the receiver using its IP address, yet by setting a random port number, the sender will provoke the receiver to reply by a TCP RST packet, thus overcoming potentially restrictive system or firewall policies deployed at the receiver's network. This method has been efficiently used to measure RTT to hosts behind firewalls [31]. A problem with the "TCP ping" approach is that it might look like a port-scan attack, and hence can trigger security alarms. Anyhow, we believe that any client interested in applying a receiver-based congestion control must be required to provide a port number that would enable servers to independently control clients' behavior (e.g., measure RTT).

In our design, a sender probes the corresponding receiver approximately once per RTT. The overhead imposed by such an approach is the following. Consider a scenario in which the throughput between the two endpoints is limited by the receiver advertised window, which is typically the case in today's Internet [32].[7] Next, consider the receiver advertised window of 64 kBytes. In such a case, the overhead imposed by a single ping per RTT is 40 Bytes/64 kBytes = 0.0625%. In absolute terms, consider 100 clients hosted by a server and average RTT of 100 ms. The throughput overhead due to out-of-band pinging becomes 40 kB/s.

Likewise, the sender must estimate the packet loss ratio and detect whether the receiver is actually re-requesting data packets that are dropped. Note that a node performing a DoS attack need not re-request dropped packets, whereas receivers that are stealing bandwidth will be forced to re-request packets for a reliable service. In any case, one possible solution to the above problem is for the sender to purposely drop a packet to test if the receiver will re-request it as the absence of a repeated request for the dropped packet would indicate a potential DoS attack. Note that this is a backward-compatible technique that could be used instead of the proposed *nonce* technique [14]. Nevertheless, here we focus on bandwidth-stealing scenarios where the receivers are forced to re-request dropped packets for a reliable service.

Once the RCP sender estimates RTT and the packet loss ratio, it can compute the TCP-friendly rate. However, because these parameters can vary significantly during a flow's lifetime, we apply the methods developed for TCP-Friendly Rate Control (TFRC) [6] to estimate the TCP-friendly rate in real time. Namely, while existing use of TFRC focuses on setting the transmission rate based on RTT and loss measurements, we utilize TFRC to *verify* TCP friendliness using the actual RTT (measured

[6] To prevent the receiver to simply send a response in anticipation of a request (thus thereby simulating a smaller RTT), the sender should randomize the period between the *ping* messages.

[7] According to measurements from [32], approximately 20% of TCP flows have the advertised window parameter set to 8 kBytes, 35% to 16 kBytes, and the rest of 45% to 64 kBytes.

via the *ping agent*) and loss measurements incurred by the RCP flow itself.

Finally, by comparing the measured throughput (based on the number of packets sent) and the throughput computed by the *TFRC agent*, the *control agent* is able to detect, and eventually punish, a misbehaving receiver. We do not implement the control module in this work, as our primary goal is to explore the ability of the above scheme to accurately *detect* receiver misbehaviors. Alternatives to punish include rate-limiting and preferentially dropping packets. However, given that the scheme can indeed accurately detect misbehaving receivers (to be shown below), the sender may simply disconnect the misbehaving client, and in that way discourage potentially malicious receivers from the temptation to steal bandwidth.

4.2. Detecting misbehaviors

Here, we first evaluate the accuracy of the TFRC agent in measuring “TCP friendliness”. Next, we retune the RCP parameters at the receiver to mimic malicious behavior, and then evaluate the sender’s ability to detect such misbehaviors.

4.2.1. TFRC agent

To robustly detect misbehaving receivers, it is essential to first evaluate the TFRC agent’s accuracy in measuring TCP friendliness. Computed TFRC throughput may deviate from actual TCP throughput due to measurement errors (low RTT sampling resolution, *ping* packets sent once per second, etc.), system dynamics, and inaccuracies in the underlying TCP equation. Thus, to manage the detection scheme’s false positives (incorrect declaration of a non-malicious flow as malicious), such inaccuracies must be incorporated into the detection process.

We conduct *ns2* simulations and consider a link shared by a number of TCP Sack flows (varied from 1 to 600). The link implements RED queue management and has capacity 10 Mb/s; we set the buffer length, min_thresh, and max_thresh to 2.5, 0.25 and 1.25 times the bandwidth-delay product, respectively. The round trip time is 50 ms. Unless otherwise indicated, these parameters are used throughout the paper. We perform a number of simulations, and present average results together with 95% confidence intervals. The *ns* code and simulation scripts are available at http://www.ece.rice.edu/networks.

To establish a baseline of TFRC’s behavior, we first mount the TFRC agent on the sender side of

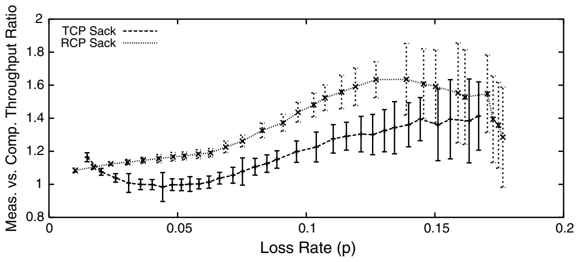

Fig. 6. TFRC agent mounted on the sender side of a well-behaved (a) TCP Sack and (b) RCP Sack.

a *sender-based* TCP Sack [20] flow and present the results in Fig. 6. The figure depicts the ratio of measured (TCP Sack) vs. computed (by the TFRC agent) throughputs as a function of the packet loss ratio. When the measured vs. computed throughput ratio is one, this indicates that the TFRC agent exactly matches the TCP Sack throughput. Observe that this is indeed the case for low packet loss ratios (for the curve labeled as “TCP Sack”). As the packet loss ratio increases, the curve moderately increases, indicating a slight conservatism of the TFRC agent as the throughput computed by the TFRC agent is slightly lower than the measured TCP Sack throughput. The problem of TFRC conservatism has been studied in depth in Ref. [33]. However, this problem is much less pronounced here than indicated in [33] as the TFRC agent in our experiment measures the actual packet loss ratio incurred by the TCP Sack flow. This ratio is much lower than the loss ratio induced by a TFRC *flow* which backs-off less conservatively than TCP Sack (see Ref. [33] for further details). In summary, the throughput computed by the *TFRC agent* deviates from the TCP Sack throughput, yet the deviation is moderate, even for high packet loss ratios. Moreover, it has been demonstrated in [45] that TFRC is capable of successfully avoiding persistent overload even in highly dynamic scenarios.

Finally, we repeat the above experiment, but now mount the TFRC agent on the RCP sender as in Fig. 5. Observe that the ratio of the measured (RCP Sack) vs. computed throughput is somewhat higher than in the above sender-based TCP Sack scenario. Indeed, RCP Sack has an improved loss recovery mechanism (see Ref. [12] for details) and consequently improves throughput. The key problem is the sender side’s difficulty in determining whether the receiver is trying to optimize its performance, or is simply stealing bandwidth. We treat this problem in detail in Section 4.3. Here, we obtained the reference measurement-based profile for a behaving RCP flow, which we will next use to demonstrate the capability of an end-point scheme to detect even moderate receiver misbehaviors.

4.2.2. Detecting misbehaving receivers

Here, we implement a misbehaving RCP node that re-tunes its congestion control parameters α, β, and RTO at the receiver. Our goal is to evaluate the sender's ability to detect these misbehaviors and to evaluate the accuracy of our modeling result from Eq. (2).

We first re-tune the additive-increase parameter α and repeat the experiment above. Fig. 7 depicts the measured vs. computed throughput ratio for misbehaving receivers (having α of 4, 9, 16 and 25), together with the same ratio for the behaving RCP flow having $\alpha = 1$. Recall that the left-most point on the curve corresponds to low loss and experiments in which the RCP flow competes with a single TCP Sack flow, whereas the right-most point on the curve corresponds to high loss and a single RCP flow competing with 600 TCP Sack flows. Observe first that the measured vs. computed throughput ratios for misbehaving flows clearly differ from the behaving flows' profile, indicating a strong potential for misbehavior detection (to be demonstrated below). Second, observe that the throughput ratio for misbehaving flows is approximately proportional to $\sqrt{\alpha}$ as predicted by the model except for extremely low aggregation regimes (e.g., $p = 0.03$ in which a single RCP flow competes with a single TCP Sack flow). In such low aggregation cases, while the misbehaving flow indeed takes significantly more bandwidth than the competing TCP Sack flow (not shown), it is unable to fully utilize the bandwidth due to frequent backoffs. Our results (not shown) indicate that similar effects occur when parameter β is re-tuned.

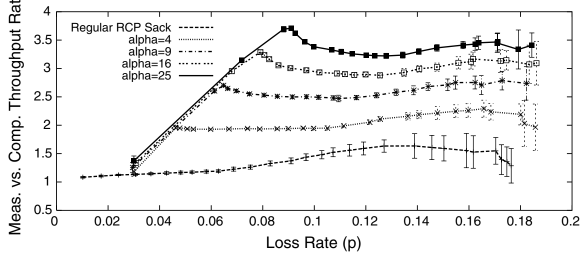

Fig. 7. Misbehaving receiver re-tunes the additive-increase parameter α.

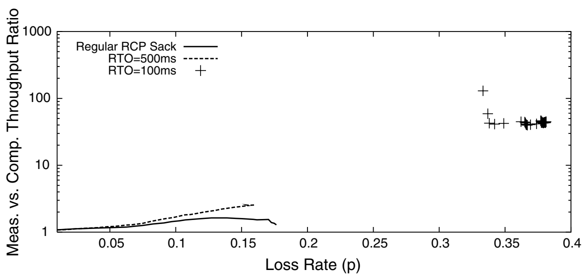

Fig. 8. Misbehaving receiver re-tunes the retransmission timeout parameter RTO.

Next, we explore misbehavers that re-tune the retransmission timeout parameter by simultaneously re-tuning both minRTO and maxRTO parameters and present the results in Fig. 8. Notice that when RTO is set to 500 ms, the receiver gradually steals more and more bandwidth as the packet loss ratio increases (as predicted by the model), since the number of time-outs increases with the packet loss ratio. However, 500 ms backoffs are sufficient to keep the system stable. On the other hand, observe that by re-tuning the RTO parameter to 100 ms (which in this scenario is smaller than the RTT), we push the system deeply into a loss regime ($p \approx 0.35$). In such a scenario, the amount of stolen bandwidth is so extreme that it may be characterized as a denial-of-service attack. Indeed, by re-tuning only a few parameters, it is possible to transform RCP (and TCP) into a powerful DoS tool (see Fig. 3).

4.2.3. Detection threshold

Here we evaluate the sender's ability to detect receiver misbehaviors and study the false-alarm probability and correct misbehavior-detection probability. Denote *meas_thr* as the throughput measured by the RCP sender, and *comp_thr* as the throughput computed by the TFRC agent (as shown in Fig. 5). Next, denote k as the threshold parameter, and define $P(k)$ as

$$P(k) = \text{Prob}\left(\frac{meas\_thr}{comp\_thr} > k\right). \tag{6}$$

For example, $P(1)$ denotes the probability that the measured vs. computed throughput ratio is larger than one, whereas $P(2)$ is the probability that the measured throughput is more than twice the computed one. If the receiver is behaving, then $P(k)$ is the *false-alarm* probability (i.e., we falsely conclude that the receiver is misbehaving with probability

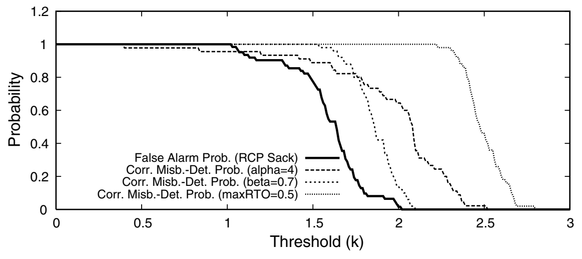

Fig. 9. Detecting out-of-profile flows.

$P(k)$). On the other hand, if the receiver is misbehaving, then $P(k)$ is the *correct misbehavior-detection* probability (i.e., we correctly conclude that the receiver is misbehaving with probability $P(k)$).

Fig. 9 plots the false alarm probability (for the behaving RCP flow), together with the correct misbehavior-detection probabilities for three moderately misbehaving receivers (exact parameters are shown in the figure). We set the packet loss ratio to 0.15 representing a scenario in which the throughput ratio deviates (approximately) the most as indicated in Fig. 6. Consequently, the false-alarm probability for the behaving RCP flow is largest, indicating that this scenario is the most challenging from the detection point of view.

The key observations from Fig. 9 are as follows. First, note the tradeoff in setting the threshold parameter k. If it is too small (e.g., $k = 1$), we are able to detect the misbehaving receivers with high probability, but the false alarm probability is also one. On the other hand, if it is set too high (e.g., $k = 3$), the false alarm probability becomes zero, but the correct misbehavior-detection probability also becomes zero. However, observe that the fact that the false-alarm probability decreases faster (for smaller k), makes it possible to set the threshold (e.g., $k = 1.8$ in this scenario), such that the false positives are acceptably small, yet we are able to detect *all* of the above cheaters with high probability. Thus, this *worst-case* scenario confirms the high precision of the end-point scheme in detecting a wide range of receiver misbehaviors. However, we will next show that setting the parameter k incurs an additional challenge when confronted with versions of TCP employing performance enhancements.

4.3. Advanced congestion control mechanisms

There is a significant body of work proposed to improve the TCP performance in wireless environments, where high channel losses may disproportionately degrade TCP Sack performance. Here, we briefly explain two well-known protocols, TCP-ELN and TCP Westwood. TCP-ELN has been proposed to distinguish wireless random losses from congestion losses. It relies on an external trigger to classify the losses, and fast retransmits lost segments due to wireless errors without decreasing down the congestion window. It has been shown in [12] that when this mechanism is applied in the *receiver-driven* protocol scenario, the throughput improvements are quite significant (we repeat this experiment and confirm the result below). This is mostly due to the fact that RCP-ELN benefits from having accurate loss classifications about all missing segments in the receive buffer.

Another protocol that improves the throughput over wireless links is TCP Westwood. It does so by using a less conservative decrease parameter β that depends on the online estimate of the available bandwidth. In this way, TCP Westwood avoids the "blind halving" of congestion window in response to a wireless error. It is expected that the same mechanism could provide further throughput improvements in receiver-driven protocols. Below, we focus on RCP-ELN and do not further consider sender- or receiver-based TCP Westwood.

We first simulate an RCP-ELN flow in a lossy wireless-like environment. Fig. 10 depicts the measured vs. computed throughput ratio as a function of loss. Observe that the RCP-ELN throughput ratio increases significantly as compared to the RCP Sack profile, indicating that RCP-ELN indeed significantly improves throughput, e.g., achieving a sixfold increase for a loss ratio of 0.17. However, the key problem is that from the sender perspective, the RCP-ELN flow is difficult to distinguish from a misbehaving flow.

Fig. 11 depicts the false-alarm probability for the behaving RCP-ELN flow for a packet loss ratio of

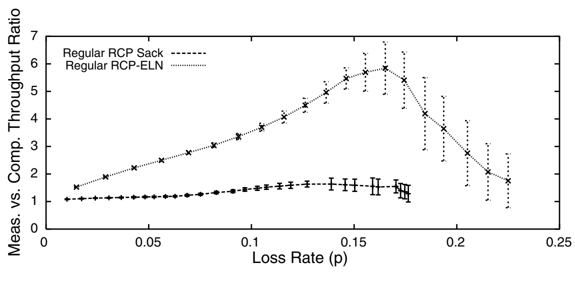

Fig. 10. RCP-ELN significantly improves throughput.

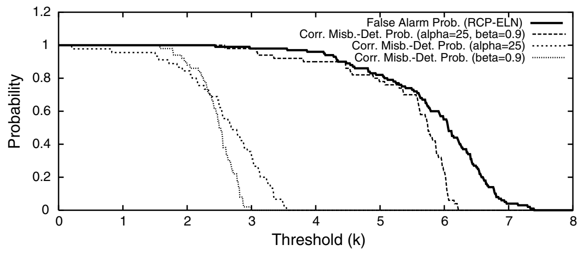

Fig. 11. From the sender's perspective, RCP-ELN looks like a misbehaving flow ($\alpha = 25$ and $\beta = 0.9$).

0.15. To emphasize the detection problem, we also plot the correct misbehavior detection probabilities (without any advanced congestion control mechanisms), with maliciously re-tuned parameters (i) $\alpha = 25$, (ii) $\beta = 0.9$, and (iii) $\alpha = 25$ and $\beta = 0.9$. Observe that using a small threshold (e.g., $k = 1$) ensures a high detection probability for any of the above misbehaviors, but we also falsely detect the RCP-ELN as malicious. However, simply increasing the threshold k does *not* eliminate the problem. For example, for $k = 4$, the false alarm probability for ELN-RCP is still one, while the probability to detect misbehaviors (i) and (ii) has already dropped to *zero*. Finally, by using a very large k (e.g., $k = 7$ in this scenario), we have an acceptably small false alarm probability for RCP-ELN, but are at the same time unable to detect any of the (quite severe) receiver misbehaviors.

Thus, these experiments illustrate a fundamental tradeoff between system performance and security (the ability to detect bandwidth stealers), as both cannot be maximized simultaneously. Ironically, while advanced congestion control mechanisms at the receiver significantly improve throughput, the resulting false-alarm probability further increases, further emphasizing the tradeoff. We believe that setting the parameter k to a larger value strikes the best balance for the file- or streaming-servers in the Internet. A large value protects servers from severe denial-of-service attacks, while enabling innovation in protocol design by preserving the performance benefits of receiver-centric transport protocols. The downside is the fact that we are unable to detect some bandwidth stealers. In contrast, strictly enforcing today's TCP-Sack throughput profile via a lower k would indeed make it possible to catch even modest bandwidth stealers. However, a small k would remove most of the RCP benefits, and indeed remove the incentive for designing and deploying enhanced TCP stacks.

5. Short time scale misbehavior

The secure RCP sender is designed to detect receiver manipulations of congestion control parameters (e.g., α, β, RTO) that would enable the receiver to steal bandwidth over longer time periods. Hence, these misbehaviors can be detected on longer timescales. In this section we explore the minimum time scale for which the sender can accurately identify receiver misbehavior. Moreover we study a receiver misbehavior targeted towards short-lived flows in which receivers begin with a large initial congestion window.

5.1. Minimum detection timescales

To explore the minimum detection time-scale, we first perform 10 experiments, and show the results in Fig. 12. In all experiments, a single RCP flow competes with 20 TCP Sack cross-traffic flows. Fig. 12 depicts the measured vs. computed throughput ratio (measured at the RCP sender) as a function of time, where the reference time zero identifies the start time of the RCP flow. In 5 of the 10 experiments, the RCP receiver behaves well (we only change the random seed for each simulation run), while in the remaining experiments we create a malicious receiver with $\alpha = 9$.

Observe that the ratios for both of the stacks (behaving and malicious) converge relatively quickly: toward one for the behaving flows, and approximately to $\sqrt{\alpha}$ for the misbehaving flows. However, note that the curves for the two stacks can be quite similar, and may overlap, over short time scales. The overlaps are due to the fact that a behaving RCP flow (just like a TCP flow) can be quite bursty over shorter time-scales (e.g., due to the exponential increase phase), and thus may

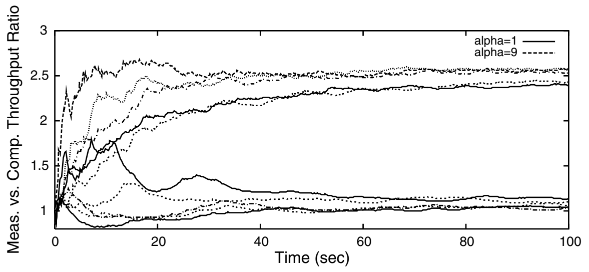

Fig. 12. Throughput ratio vs. time for (a) a well-behaving flow and (b) a misbehaving flow ($\alpha = 9$) (for five different random seeds).

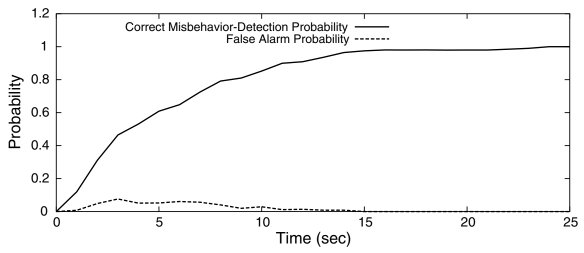

Fig. 13. Probability to detect a misbehaving flow increases as the time evolves.

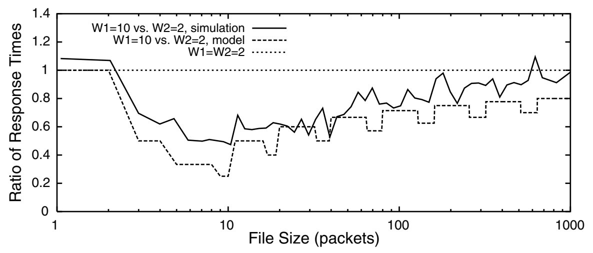

Fig. 14. Misbehaving receiver re-tunes the initial window size parameter W (link utilization 10%).

deviate from the TCP-friendly rate computed by the TFRC agent.

Next, we perform an extensive set of simulations to statistically quantify the above observations. Fig. 13 depicts the correct misbehavior detection probability (for the misbehaving flow with $\alpha = 9$), together with the false alarm probability (for the behaving flows) as a function of time and for $k = 2$. Observe first that the correct misbehavior-detection probability converges to one as time evolves, indicating that it becomes more and more certain that the receiver is misbehaving. On the other hand, observe that the false-alarm probability for the same scenario is quite low (only several percent up to 10 s) and approaches zero beyond 10 s. Thus, beyond this time scale, it is possible to detect the receiver misbehavior with high confidence, and the sender can freely punish the receiver given that the probability to falsely detect a behaving flow drops to near zero beyond 10 s.

However, very short-lived flows transmitting up to tens or hundreds of packets are common in today's Internet due to web traffic. The file transmission times typically last for only several ms to several hundreds of ms, and the above scheme (targeted to detect bandwidth stealers in file- or streaming-server scenarios) is not designed to detect very short time-scale misbehaviors. Below, we first explore additional short time-scale receiver misbehaviors targeted for web-browsing and short files, and then analyze appropriate protection mechanisms.

5.2. Initial congestion window

Here, we consider web RCP flows that increase their initial congestion window in order to obtain decreased response time. We show that it is possible for a malicious receiver to not only significantly improve its own response time, but to also severely degrade the response times for the background traffic.

We adopt the model developed in [34] in which clients initiate sessions from randomly chosen web sites (the server pool) with several web pages downloaded from each site. Each page consists of several objects, which are downloaded by either TCP or RCP, depending on the client (all the servers in the pool support both options). There is a single misbehaving client in the client pool, which uses a mis-configured RCP (details are given below), while the other clients from the pool behave and use unmodified TCP Sack.

Fig. 14 depicts the average file response time for the RCP flow (normalized by the response times for the same flow when the RCP client is well behaving) as a function of file size. Because of the normalization, the curve labeled as "$W1 = W2 = 2$" is a straight line with a value of one. On the other hand, notice that the misbehaving RCP client is able to significantly improve its response times by increasing the initial window size parameter W to 10. Observe next that the malicious receiver achieves the maximum improvement exactly for the files that are 10-packets long, and this is because such files are downloaded in a single burst (files with size less than 10 packets are also downloaded in a single burst, but the improvement is most prominent for the longest files in this single-burst-category). On the other hand, files longer than 10 packets also improve their response times, simply due to the fact that their congestion windows are jump-started with $W = 10$. Next, observe that the modeling result from Section 3.2.2 accurately tracks the simulation results.[8] The non-monotonic and alternating quasi-periodic

[8] We set $C = 10$ Mb/s in Eqs. (4) and (5), which due to the low average utilization of 10%, is close to the flow's available bandwidth.

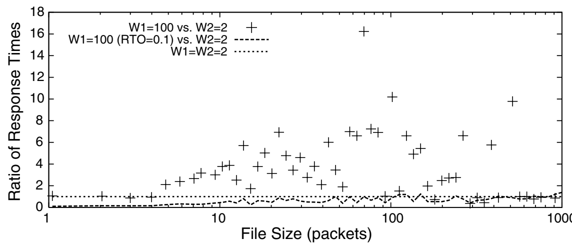

Fig. 15. A greedy receiver ($W = 100$) may degrade its own response times; but "turning off" the backoff timers ($W = 100$, $\mathrm{RTO} = 0.1$) "improves" the response times (link utilization 90%).

shape of the modeling curve is due to the use of the ceiling function ($\lceil \cdot \rceil$) in Eqs. (3) and (5).

While it may appear attractive for a malicious client to maximally increase the initial window size parameter W in order to steal more and more bandwidth, this is not necessarily a good option, especially in more congested environments. This is illustrated in Fig. 15, where we increase the link utilization to 90%, and the malicious clients sets the initial window size parameter W to 100 packets. Here, this greedy user significantly degrades not only the background traffic,[9] but also degrades its *own* response times (shown in the figure) by an order of magnitude. This degradation is due to the fact that when the malicious user sends large bursts of requests, it forces the web server to reply with large bursts of data packets, many of which are themselves lost in the congestion. These packet losses force even the RCP user to enter the exponential backoff phase and degrades its response time. To overcome the above problem, the malicious user needs to "turn off" the exponential backoff timers. We do this by re-tuning the RTO parameter to 100 ms. In this way, the malicious user is able both to "push-out" and significantly degrade the background traffic, and at the same time improve its own response times, as also shown in the figure.

5.3. Solutions

Here, we explore two possible solutions to the above short-time-scale misbehaviors. One is to rate-limit flows, which while effective in thwarting cheaters, is a non-work conserving solution in which it is problematic to determine the appropriate rate. The second solution is to have a "smart" RCP client at the sender side that would enforce a "TCP-friendly" exponential window increase. It would estimate the RTT to the client, and release the *data* packets accordingly. While also effective in thwarting cheaters, this approach unfortunately mitigates some of the benefits of RCP.

To study the performance of the above solutions, we compute and plot in Fig. 16 the file-response times in three different scenarios for the RCP flow with the available bandwidth of 10 Mb/s and RTT of 50 ms: (i) when a malicious user sets the initial window W to 100 packets and the sender does not rate limit (labeled as "Rcv. misbehaving – Snd. unprotected"); (ii) the receiver sets $W = 100$, but the sender rate limits to 200 Kb/s (labeled as "Rcv. misbehaving – Snd. rate-limited") and (iii) the receiver is well behaving and is not rate-limited (labeled as "Rcv. well-behaving – Snd. unprotected"). Fig. 16 illustrates problems in setting the rate-limit value. Setting it to 200 Kb/s degrades the file response times significantly, as shown in Fig. 16.

But the key insight from the above experiment is that using a large initial window sizes can significantly (up to 10 times in the above scenario – and much more in larger-bandwidth networks) improve file response times. Such methodologies have been studied in depth in [35–38], but in the context of sender-based TCP, where the web-server increases the initial window size in an attempt to improve system performance. However, in the receiver-driven RCP scenario, it is hard to distinguish whether the receiver is jump-starting the TCP flow or is simply malicious. Thus, applying rate limiting or the "smart" RCP client methodology may indeed protect the system against receiver misbehavior, but at the same time prevents attempts as in [35–38] to improve performance. This illustrates the tradeoff between

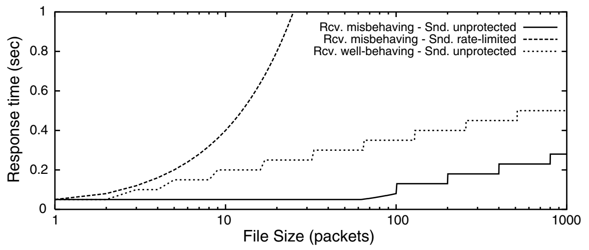

Fig. 16. Protecting against short-time-scale misbehaviors.

[9] We do not show the impact of misbehaving clients on the background traffic in the figure because the effects are similar to those shown in Fig. 3; likewise, mechanisms invoking vulnerabilities in this scenario are essentially the same as the ones discussed in Section 3.

system security and performance in that strict enforcement of protocol rules would not only reduce performance, but would also inhibit protocol innovation.

However, unlike in the RCP-ELN scenario (in which we concluded that using larger values for the detection threshold parameter k can protect against DoS attacks, but not from relatively moderate bandwidth stealing), here, we conclude that either rate-limiting or a "smart" RCP client has to be *strictly* applied, because a receiver with an excessively large W in combination with manipulated exponential backoff timers can significantly degrade the legitimate background traffic (Fig. 15). Yet, applying any of the short-time-scale protection methodologies inevitably reduces the incentive for receivers to use RCP for short-lived flows, as sender-based TCP enhanced with jump-starting methodologies is able to achieve the best response-time curve from Fig. 16 without any security considerations.

6. Network- vs. end-point-based solutions: analysis and overview

Here, we review several state-of-the-art core-router-, edge-router-, and end-point-based solutions designed to detect and thwart malicious flows.

6.1. Network-based solutions

Out of many router-assisted schemes designed to detect malicious flows in the network, we consider several representative schemes. First, we analyze RED-PD (RED with Preferential Dropping) in detail, and then briefly discuss variants of Fair Queuing (FQ). In addition, we explain how *push-back*, a router-based protection scheme designed to protect the network against Distributed DoS (DDoS) attacks, can be exploited by misbehaving receivers to launch DoS attacks against public Internet servers.

6.1.1. Detection: a simulation analysis

In [17], Mahajan et al. develop *RED-PD*, a scheme that uses the packet drop history at a router to detect high-bandwidth flows in times of congestion, and preferentially drop packets from these flows. In order to detect high-bandwidth flows, RED-PD sets a *target bandwidth* above which a flow is identified as malicious. The target bandwidth is defined as the bandwidth obtained by a *reference* TCP flow with the *target RTT* (default is 40 ms), and the current drop rate measured at the output router queue. The targeted bandwidth is computed using the square-root TCP-friendly formula. In other words, in the absence of per-flow RTT measurements, RED-PD sets the target RTT to 40 ms as a bound for distinguishing in- vs. out-of-profile flows.

While RED-PD can protect the system against certain misbehaviors, the lack of exact knowledge of the flow's RTT fundamentally limits its ability to detect severe end-point misbehaviors as demonstrated in Fig. 17. We perform *ns* experiments with nine flows sharing a RED-PD router. We vary the round-trip times of the flows from 20 to 350 ms (as shown on the x-axis), and plot the bandwidth of a single flow on the y-axis. When all flows are well-behaved, the bandwidth share is fair (the straight line in the figure). However, when one of the flows (whose normalized throughput is shown on y-axis) re-tunes α to 25, it can potentially steal up to five times more bandwidth than its fair share according to Eq. (2). Observe that RED-PD successfully limits the malicious flow to its fair-share, but only when the RTT is less than or equal to 40 ms (recall that this is the RTT of the *reference* flow). However, as the flows' RTT increases, the malicious flow is able to steal more and more bandwidth, up to five times more than its fair share (the maximum for this scenario) when the RTT is 350 ms.

RED-PD's limitations in detecting misbehaving flows are more general than indicated in the above example. First, it is important to notice that a misbehaving flow can steal bandwidth not only in homogeneous-RTT scenarios as in the above experiments, but also in heterogeneous-RTT environments, since the amount of stolen bandwidth depends on the RTT of a misbehaving flow. Second, while in this paper we focus on receiver-driven

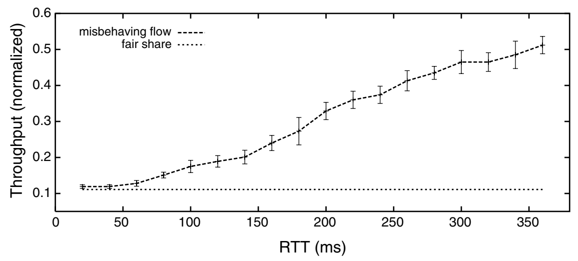

Fig. 17. RED-PD is unable to detect a malicious flow.

transport protocols, observe that the above RED-PD limitations apply equally to sender-based TCP stacks. Another problem arises from the fact that RED-PD uses a simple (and less accurate) square-root formula, which significantly overestimates the TCP-friendly rate for higher packet loss ratios because it does not account for retransmissions [15]. Hence, malicious TCP or RCP flows have the opportunity to steal dramatically more bandwidth as the packet loss ratio increases, e.g., 100 times more when $p = 0.3$, as indicated in Fig. 4.

While it may appear attractive to apply some version of *fair queuing* (including the preferential-dropping schemes developed to enforce fairness among adaptive and non-adaptive flows, e.g., Flow Random Early Detection (FRED) [39], CHOKe [40], or Stochastic Fair Blue (SFB) [41]) to solve the above problem, observe that such schemes are also unable to detect end-point misbehaviors and to enforce the proportional fairness targeted by TCP. Moreover, in a heterogeneous RTT environment, such schemes will significantly deviate from the proportional bandwidth share, and even magnify the bandwidth-stealing effects. Below, we provide a simple, yet illustrative example. While not representative of an actual or realistic scenario, our main goal is to illustrate the difference between proportional (RTT-dependent) and max-min fairness as enforced by FQ.

Consider a link shared by three congestion-controlled flows, such that the proportional fair share is (0.9, 0.05, 0.05), due to flow 1 having a shorter path length (smaller RTT) than flows 2 and 3. Next, assume that flow 2 is malicious. It re-tunes its parameters and utilizes more bandwidth by stealing from flow number one, such that the bandwidth share is now (0.7, 0.25, 0.05). However, if FQ is used, all flows get their “fair-share”, and the bandwidth share is now (0.33, 0.33, 0.33). Thus, FQ provides even more bandwidth to flow 2 than it could have stolen without it.

6.1.2. Protection

In [18], Mahajan et al. develop both a local mechanism for detecting and controlling an aggregate at a single router, and a cooperative *pushback* mechanism in which a router can ask upstream routers to control an aggregate (e.g., preferentially drop its packets). Here, we focus on the *pushback* mechanism. While this cooperative effort among network routers indeed seems to be a reasonable approach in preventing high-rate DoS attacks, it can actually become a tool misused by DoS attackers in a scenario with receiver-driven TCP stacks. The key problem is that *pushback* assumes that the endpoints that are sending packets at high rates are malicious, which is not necessarily true in the receiver-based congestion control case. For example, consider the request-flood scenario of Fig. 3, in which a malicious receiver provokes the sender (e.g., a public web server) to flood the network. In such a scenario, the *pushback*-enabled routers would coordinate an action *against* the web-server traffic aggregate in the upstream direction, from the congestion point toward the non-malicious web server, thereby degrading its service.

6.2. Edge-router-based solutions

In [16], Mirkovic et al. develop *D-WARD*, an edge-router based protection scheme for detecting DoS activity. For each traffic type, they establish a baseline traffic model. For a TCP session, they measure both outgoing (*data*) and incoming (*ack*) traffic and define the maximum allowable ratio of the two. When the ratio of the number of data vs. ack packets goes over a certain threshold, they conclude that the flow is out of profile and rate-limit it. While the above scheme may indeed protect against TCP-based denial-of-service attacks (where the sender floods the network with data packets independent of the feedback from the receiver), like in the above *pushback* example, this model clearly does not apply to the *receiver-driven* TCP scenario. Recall that in the receiver-based scenario, the number of requests and data packets is the same in both directions, even in the most severe denial-of-service scenarios. Moreover, the fact that the number of packets in the forward (*data*) and reverse (*req*) directions is the same is actually the core idea of the request-flood attack: the receiver floods the sender with requests, and the sender replies by transmitting the same number of data packets, yet with significantly larger size thereby congesting the network.

In [42], Paxson presents *tcpanaly*, a tool whose initial goal was to work in *one pass* over a packet trace by recognizing *generic* TCP actions. The goal of executing only one pass stemmed from the objective that `tcpanaly` might later evolve into a tool that could monitor an Internet link in real-time and detect misbehaving TCP sessions on the link. Unfortunately, the author was forced to abandon both of the goals. Among many obstacles, the key one is that one-pass analysis proved difficult due to vantage point issues (see Ref. [42] for details), in which it was often hard to tell whether a TCP flow's actions were due to the most recently received packet, or one received in the distant past.

6.3. End-point-based solutions

In [13], Savage et al. demonstrate that there exist simple attacks in *sender-based* TCP scenarios that allow a misbehaving receiver to drive a standard TCP sender arbitrarily fast, without losing end-to-end reliability. Fortunately, the authors show that it is possible to modify the design of TCP to eliminate this behavior, without requiring that the receiver be trusted in any manner. Unfortunately, these modifications are specific for the set of attacks explored in [13], and specific to sender-based TCP. Hence, they do not solve the problem in the receiver-based congestion control scenario.

The above paper also proposed the use of *nonce* fields in the TCP packet format. For each segment, the sender fills the *nonce* field with a unique random number. When a receiver generates an ACK in response to a data segment, it echoes the nonce value, and thus prove that it has received a packet. While the requirement for the less-trusted communication party to prove that it has received a packet is certainly beneficial, note that it does not address the bandwidth stealing problem analyzed here (in Section 3.1.2): the receiver may regularly echo-back all nonces to the sender, yet voluntarily manipulate the congestion control parameters (all available at the receiver) and request as many packets as it wants.

Finally, in [43], Patel et al. designed an end-point scheme whose goal is to verify TCP friendliness in the context of untrusted mobile code. The key difference between our scheme, initially presented in [44], and the one from [43] is that our scheme aims to thwart possible *receiver* misbehaviors, and hence does not require any cooperation from a potentially malicious receiver. Moreover, in contrast to the scheme from [43], which compares the TCP sending rate to the TCP-friendly equation rate [15], our scheme applies the TFRC protocol to estimate the TCP-friendly rate in real time. This is particularly important in the presence of highly dynamic background traffic; while being an equation-based scheme, TFRC manages to adapt to relatively short time-scale available-bandwidth fluctuations [45].

7. Conclusions

Receiver-driven transport protocols delegate key control functions to receivers. While this radically new protocol design achieves significant performance and functionality gains in a variety of wireless and wireline scenarios, we showed that a high concentration of control functions available at the receiver leads to an extreme vulnerability. Namely, receivers would have both the means and incentive to tamper with the congestion control algorithm for their own benefits. We analyzed a set of easy-to-implement receiver misbehaviors and analytically quantified the substantial benefits that a malicious client can achieve in terms of stolen bandwidth over long time-scales (e.g., in file- or streaming-server scenarios) and response time improvements for short-files in HTTP scenarios.

We evaluated a set of state-of-the-art network-based solutions, and proposed and analyzed a set of end-point solutions. Our findings are as follows: (1) Network-based solutions are fundamentally limited in their ability to detect and punish even severe endpoint misbehaviors. (2) End-point solution can accurately detect long-time-scale receiver misbehaviors and strictly enforce the TCP-friendly rate, but such enforcement entirely *removes* the performance benefits of receiver-driven protocols. (3) In the file- and streaming-server scenarios, it is possible to strike an acceptable balance between protocol performance on one hand, and vulnerability to misbehavers on the other, due to the fact that moderate bandwidth stealers do not represent a critical threat to the system security. (4) On the contrary, short time-scale receiver misbehaviors can extremely degrade the response times of well-behaving clients in the HTTP-server scenarios; hence, such servers have to *strictly* apply sender-based short-time-scale protection mechanisms; unfortunately, such mechanisms can often limit the receiver-driven TCP performance to a level which is *below* the level achievable by sender-based TCP.

Appendix A. Computing the throughput of an RCP (TCP) flow with misconfigured parameters

We use exactly the same assumptions, methodology, and notation as in [15]. Due to space constraints, we only present the key steps of the derivation. Consequently, a reader interested in following the derivation below needs to use Ref. [15] in parallel.

A.1. Loss indications are exclusively "triple-duplicate" ACKs

Assume that a user increases window size by α packets, and that it decreases it by a factor of β. Denote d as $1/\beta$. Then, Eq. (7) from [15] becomes

$$W_i = \frac{W_{i-1}}{d} + \frac{X_i}{b/\alpha}, \quad i = 1, 2, \ldots, \tag{7}$$

where X_i is the number of increase rounds in the ith tripple-duplicate period (TDP$_i$). Equation indicates that during TDP$_i$, the window size increases between W_{i-1}/d and W_i, and the increase is linear with slope α/b. Consequently, the number of packets transmitted in TDP$_i$ is expressed by

$$Y_i = \frac{X_i}{2}\left(\frac{W_{i-1}}{d} + W_i - \alpha\right) + \beta_i, \tag{8}$$

where β_i is the number of packets sent in the last round. Next, assuming that X_i and W_i are mutually independent sequences of i.i.d. random variables, it follows from the above two equations and Eq. (5) from [15] that

$$E[W] = \frac{\alpha}{b}\frac{d}{d-1}E[x], \tag{9}$$

and

$$\frac{1-p}{p} + E[W] = \frac{E[X]}{2}\left(\frac{E[W]}{d} + E[W] - \alpha\right) + \frac{E[W]}{2}. \tag{10}$$

From Eqs. (9) and (10), we have

$$E[W] = \frac{d\alpha}{1+d}\frac{b(d-1)+d}{2b(d-1)} + \sqrt{\left(\frac{b(d-1)+d}{2b(d-1)}\right)^2\left(\frac{d\alpha}{1+d}\right)^2 + 2\frac{\alpha d^2}{b(1+d)(d-1)}\frac{1-p}{p}}. \tag{11}$$

Observe that,

$$E[W] = \sqrt{\frac{d^2}{(1+d)(d-1)}\frac{2\alpha}{bp}} + \mathrm{o}(1/\sqrt{p}). \tag{12}$$

i.e., $E[W]$ converges to the first term of Eq. (12) for small values of p. From Eqs. (9), (14) as well as Eq. (6) from [15], we derive the expressions for the expected number of rounds ($E[X]$) in the TD period, as well as the expected duration $E[A]$ of the same period.

A simplified expression for $E[X]$ is

$$E[X] = \sqrt{\frac{2b(d-1)}{(1+d)p\alpha}} + \mathrm{o}(1/\sqrt{p}), \tag{13}$$

and

$$E[A] = \mathrm{RTT}\left(\frac{b(d+1)+d}{2(1+d)} + \sqrt{\left(\frac{b(d+1)+d}{2(1+d)}\right)^2 + \frac{1-p}{p}\frac{2b}{1+d}\frac{d-1}{\alpha} + 1}\right). \tag{14}$$

Finally, based on Eq. (18) from [15], as well as $E[X]$ and $E[A]$ derived here, it could be shown that the RCP (TCP) throughput $B(p)$ is

$$B(p) = \frac{1}{\mathrm{RTT}}\frac{1}{\sqrt{\frac{2bp(d-1)}{\alpha(d+1)}}} + \mathrm{o}(1/\sqrt{p}). \tag{15}$$

A.2. Loss indications are "triple-duplicate" ACKs and timeouts

Using Eqs. (25) and (28) from [15], as well as Eqs. (12) and (13) from above, it could be shown that Eq. (2) follows.

References

[1] S. Floyd, Highspeed TCP for large congestion windows, Internet RFC 3649, December 2003.

[2] C. Jin, D. Wei, S. Low, FAST TCP: motivation, architecture, algorithms, performance, in: Proceedings of IEEE INFOCOM '04, Hong Kong, China, 2004.

[3] H. Balakrishnan, R. Katz, Explicit loss notification and wireless Web performance, in: Proceedings of IEEE GLOBECOM '98, Sydney, Australia, 1998.

[4] C. Casetti, M. Gerla, S. Mascolo, M.Y. Sanadidi, R. Wang, TCP Westwood: bandwidth estimation for enhanced transport over wireless links, in: Proceedings of ACM MOBICOM '01, Rome, Italy, 2001.

[5] D. Clark, M. Lambert, L. Zhang, NETBLT: a high throughput transport protocol, in: Proceedings of ACM SIGCOMM '87, Stowe, VM, 1987.

[6] S. Floyd, M. Handley, J. Padhye, J. Widmer, Equation-based congestion control for unicast applications, in: Proceedings of ACM SIGCOMM '00, Stockholm, Sweden, 2000.

[7] P. Mehra, A. Zakhor, C.D. Vleeschouwer, Receiver-driven bandwidth sharing for TCP, in: Proceedings of IEEE INFOCOM '03, San Francisco, CA, 2003.

[8] P. Sinha, N. Venkitaraman, R. Sivakumar, V. Bharghavan, WTCP: a reliable transport protocol for wireless wide-area networks, in: Proceedings of ACM MOBICOM '99, Seattle, WA, 1999.

[9] N. Spring, M. Chesire, M. Berryman, V. Sahasranaman, T. Anderson, B. Bershad, Receiver based management of low bandwidth access links, in: Proceedings of IEEE INFOCOM '00, Tel Aviv, Israel, 2000.

[10] V. Tsaoussidis, C. Zhang, TCP-real: receiver-oriented congestion control, The Journal of Computer Networks 40 (4) (2002) 477–497.

[11] R. Gupta, M. Chen, S. McCanne, J. Walrand, A receiver-driven transport protocol for the Web, in: Proceedings of INFORMS '00, San Antonio, TX, 2000.
[12] H.-Y. Hsieh, K.-H. Kim, Y. Zhu, R. Sivakumar, A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces, in: Proceedings of ACM MOBICOM '03, San Diego, CA, 2003.
[13] S. Savage, N. Cardwell, D. Wetherall, T. Anderson, TCP congestion control with a misbehaving receiver, ACM Comput. Commun. Rev. 29 (5) (1999) 71–78.
[14] D. Ely, N. Spring, D. Wetherall, S. Savage, T. Anderson, Robust congestion signaling, in: Proceedings of IEEE ICNP '01, Riverside, CA, 2001.
[15] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Modeling TCP Reno performance: a simple model and its empirical validation, IEEE/ACM Trans. Network. 8 (2) (2000) 133–145.
[16] J. Mirkovic, G. Prier, P. Reiher, Attacking DDoS at the source, in: Proceedings of IEEE ICNP '02, Paris, France, 2002.
[17] R. Mahajan, S. Floyd, D. Wetherall, Controlling high-bandwidth flows at the congested router, in: Proceedings of IEEE ICNP '01, Riverside, CA, 2001.
[18] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, S. Shenker, Controlling high bandwidth aggregates in the network, ACM Comput. Commun. Rev. 32 (3) (2002) 62–73.
[19] H. Balakrishnan, V. Padmanabhan, S. Seshana, R. Katz, A comparison of mechanisms for improving TCP performance over wireless links, IEEE/ACM Trans. Network. 5 (6) (1997) 756–769.
[20] S. Floyd, J. Madhavi, M. Mathis, M. Podolsky, An extension to the selective acknowledgement (SACK) option for TCP, Internet RFC 288, July 2000.
[21] V. Paxson, M. Allman, Computing TCP's retransmission timer, Internet RFC 2988, November 2000.
[22] M. Allman, S. Floyd, C. Partridge, Increasing TCP's initial window, Internet RFC 2414, September 1998.
[23] M. Handley, J. Padhye, S. Floyd, J. Widmer, TCP friendly rate control, IETF Internet draft, July 2001.
[24] S. Floyd, M. Handley, J. Padhye, A comparison of equation-based and AIMD congestion control, Technical Report.
[25] Y. Yang, S. Lam, General AIMD congestion control, Technical Report TR-00-09, Department of Computer Science, UT Austin, May 2000.
[26] M. Mathis, J. Semke, J. Madhavi, T. Ott, The macroscopic behavior of the TCP congestion avoidance, ACM Comput. Commun. Rev. 27 (3) (1997) 67–82.
[27] L. Guo, I. Matta, The war between mice and elephants, in: Proceedings of IEEE ICNP '01, Riverside, CA, 2001.
[28] N. Cardwell, S. Savage, T. Anderson, Modeling TCP latency, in: Proceedings of IEEE INFOCOM '00, Tel Aviv, Israel, 2000.
[29] J. Heidemann, K. Obraczka, J. Touch, Modeling the performance of HTTP over several transport protocols, IEEE/ACM Trans. Network. 5 (5) (1997) 616–630.
[30] C. Patridge, T. Shepard, TCP/IP performance over satellite links, IEEE Network 11 (5) (1997) 44–49.
[31] K. Gummadi, H. Madhyastha, S. Gribble, H. Levy, D. Wetherall, Improving the reliability of Internet paths with one-hop source routing, in: Proceedings of OSDI '04, San Francisco, CA, 2004.
[32] A. Medina, J. Padhye, S. Floyd, Measuring the evaluation of transport protocols in the Internet, Technical Report, 2004.
[33] M. Vojnovic, J.-Y.L. Boudec, On the long-run behavior of equation-based rate control, in: Proceedings of ACM SIGCOMM '02, Pittsburgh, PA, 2002.
[34] A. Feldmann, A. Gilbert, P. Huang, W. Willinger, Dynamics of IP traffic: a study of the role of variability and the impact of control, in: Proceedings of ACM SIGCOMM '99, Vancouver, British Columbia, 1999.
[35] J. Hoe, Improving the start-up behavior of a congestion control scheme for TCP, in: Proceedings of ACM SIGCOMM '96, Stanford University, CA, 1996.
[36] V. Padmanabhan, R. Katz, TCP fast start: a technique for speeding up Web transfers, in: Proceedings of IEEE GLOBECOM '98, Sydney, Australia, 1998.
[37] R. Wang, G. Pau, K. Yamada, M. Sanadidi, M. Gerla, TCP start up performance in large bandwidth delay networks, in: Proceedings of IEEE INFOCOM '04, Hong Kong, China, 2004.
[38] Y. Zhang, L. Qiu, S. Keshav, Speeding up short data transfers: theory, architectural support, and simulations, in: Proceedings of NOSSDAV '00, Chapel Hill, NC, 2000.
[39] D. Lin, R. Morris, Dynamics of random early detection, in: Proceedings of ACM SIGCOMM '97, Cannes, France, 1997.
[40] R. Pain, B. Prabhakar, K. Psounis, CHOKe, a stateless active queue management scheme for approximating fair bandwidth allocation, in: Proceedings of IEEE INFOCOM '00, Tel Aviv, Israel, 2000.
[41] W. Feng, D. Kandlur, D. Saha, K. Shin, Stochastic fair BLUE: a queue management algorithm for enforcing fairness, in: Proceedings of IEEE INFOCOM '01, Anchorage, Alaska, 2001.
[42] V. Paxson, Automated packet trace analysis of TCP implementations, in: Proceedings of ACM SIGCOMM '97, Cannes, France, 1997.
[43] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, T. Stack, Upgrading transport protocols with untrusted mobile code, in: Proceedings of ACM SOSP '03, Bolton Landing, NY, 2003.
[44] A. Kuzmanovic, E. Knightly, A performance vs. trust perspective in the design of end-point congestion control protocols, in: Proceedings of IEEE ICNP '04, Berlin, Germany, 2004.
[45] D. Bansal, H. Balakrishnan, S. Floyd, S. Shenker, Dynamic behavior of slowly-responsive congestion control algorithms, in: Proceedings of ACM SIGCOMM '01, San Diego, CA, 2001.

Aleksandar Kuzmanovic is an assistant professor in the Department of Electrical Engineering and Computer Science at Northwestern University. He received his B.S. and M.S. degrees from the University of Belgrade, Serbia, in 1996 and 1999 respectively. He received his Ph.D. degree from the Rice University in 2004. His research interests are in the area of computer networking with emphasis on design, security, analysis, theory, and prototype implementation of protocols and algorithms for the wired and wireless Internet.

Edward W. Knightly is a professor of Electrical and Computer Engineering at Rice University. He received his B.S. degree from Auburn University in 1991 and the M.S. and Ph.D. degrees from the University of California at Berkeley in 1992 and 1996, respectively. He is an associate editor of IEEE/ACM Transactions on Networking. He served as technical co-chair of IEEE IWQoS 1998 and IEEE INFOCOM 2005 and served on the program committee for numerous networking conferences including ICNP, INFOCOM, IWQoS, MobiCom, and SIGMETRICS. He received the National Science Foundation CAREER Award in 1997 and the Sloan Fellowship in 2001. His research interests are in the areas of mobile and wireless networks and high-performance and denial-of-service resilient protocol design.